Program Development in Action Systems

Ian Collier

The Queen's College

Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford

Michaelmas Term, 1994

Program Development in Action Systems

Ian Collier, The Queen's College

Thesis submitted for the degree of Doctor of Philosophy
Oxford University, Michaelmas 1994

Abstract

The action-system formalism, introduced by Back and Kurki-Suonio [2], is a state-based approach to distributed computing. An action system may be viewed as a sequential program or as a collection of processes which communicate by means of shared variables; it therefore provides a flexible way in which to develop programs on a variety of architectures. This thesis describes a model in which such developments may take place and shows several example developments.

Input and output events are introduced into the notation together with probes similar to those of Alain Martin [15]. These may be used by 'reactive' action systems to communicate with the environment; such action systems usually go on forever producing results rather than terminating and giving the result of a computation as part of the final state.

The basis of the development model is a definition of the correctness of an action system, based on a specification which details the required behaviour of the system and also the constraints which it may place upon communications with the environment. Such a specification is equally suitable for both terminating and reactive action systems. The model makes it possible for the input and output of an action system to be changed; this is known as data refinement.

A parallel operator is introduced, which allows a system to be divided explicitly into two or more processes which communicate internally via input and output events. The operator is first defined by stating the semantics of the resulting system in terms of the semantics of the two operands, and later by stating the syntax of the resulting system in terms of the syntax of the two operands. These definitions are shown to be equivalent under certain circumstances.

Simulation is a proof technique for showing refinement between action systems. Simulation is shown to be sound within our formalism and forms the basis of several of the example developments.

Acknowledgements

I am indebted to my supervisor, Jeff Sanders, who pointed me in the right direction, provided many ideas for development and was a source of encouragement. Thanks also to Tony Hoare, who took on this role during Jeff's visit to MIT.

Thanks to Oxford University and to the Queen's College for providing a comfortable and stimulating environment in which to work. Thanks also to various friends in Oxford and to my parents and brothers for their support.

The TEXbook by Donald Knuth has been an invaluable aid in the design and production of this thesis.

This work was supported by SERC grant number 90807341.

Contents

1

Introduction

1.1 Action Systems

The action system of Back and Kurki-Suonio [2] provides a state-based approach to concurrency which will be used throughout this thesis. In such a system a state is shared by several actions, each of which consists of a guard and a command. The guard is a predicate upon the state of the system which determines the conditions necessary for the action to be executed. When the guard is true, the command may be executed. The command may change the state by assigning to one or more variables or it may cause input from or output to an environment. An action system is executed by repeatedly choosing an action whose guard is true and executing it. Execution continues until a state is reached which does not satisfy any of the guards. If this eventually happens then the action system terminates, giving a final state which is often useful as the result of a computation. Many action systems are designed in such a way that at all times at least one action may be executed. Such systems do not terminate and are often called *reactive systems*. A reactive system is not designed to give the result of a computation but instead is intended to accept an ongoing stream of inputs from its environment and react by providing an ongoing stream of outputs in return. The UNITY formalism of Misra and Chandy [6] has essentially the same structure, but lacks explicit guards in favour of a fairness constraint upon the execution of the actions.

An action system may be viewed as an implementation of an algorithm or as a datatype or module, depending upon the context. The view that is taken will usually influence the way in which the order of execution of the actions is chosen. If the action system is to be viewed as an implementation, then the order of execution will normally be chosen internally without influence from the environment. The system will progress autonomously until a computation has been performed, at which point the result will be made available either by output or in the final state at termination. Most of the examples in this thesis are of this type (although chapter 5 shows how action systems can influence the order of execution of other systems running in parallel). An action system which is a module will usually be a reactive system, each of whose actions performs a specific function. The environment will decide which action will be executed and when, under the constraints given by the guards of the actions. The environment will observe the result of each action by examining the state or by accepting output from the system. Chapter 4 deals with an action system of this form.

1.2 Examples

1.2.1 An action system viewed as an implementation

For an example of an action system viewed as an implementation, consider the problem from [6] of scheduling a meeting. Three people, F, G and H, wish to find the earliest time for a meeting which is acceptable to all three. Time is integer-valued and non-negative, with zero being the present. Associated with F, G and H are functions f, g and h respectively which map times to times. For any time t, the value of $f(t)$ is the earliest time of value no less than t at which F is available for a meeting. The definitions of g and h are similar. A common meeting time z is a time satisfying $z = f(z) = g(z) = h(z)$. Given that such a z exists, the following action system implements an algorithm for computing the least one:

$$
\begin{array}{l}
\mathbf{var}\ z : \mathbb{N} \bullet z = 0; \\
\mathbf{do}\ z \neq f(z) \rightarrow z := f(z) \\
[\!]\ z \neq g(z) \rightarrow z := g(z) \\
[\!]\ z \neq h(z) \rightarrow z := h(z) \\
\mathbf{od}.
\end{array}
$$

The system contains one guard for each person whose guard is the assertion that a satisfactory meeting time has not been arranged and whose command assigns a new value to z for the next approximation to a meeting time. These actions are executed repeatedly starting from $z = 0$ until a result is reached. Since f, g and h are monotonic functions and a least solution exists, we can see that the value of z is always bounded above by this solution and that z increases each time an action is executed. Therefore the system must terminate after a finite number of steps with z satisfying $z = f(z) = g(z) = h(z)$.

1.2.2 An action system viewed as a module

For an example of an action system viewed as a module, consider a simple mean calculator which inputs a set of numbers and can compute the mean value (this has also been used as an example of a module by Morgan[17]).

$$
\begin{array}{ll}
 & \mathbf{var}\ s : \mathbf{bag}\,\mathbb{R} \bullet s = \emptyset; \\
 & \mathbf{do} \\
\mathit{clear}: & \quad \mathit{true} \to s := \emptyset \\
\mathit{add}: & [\!]\ \mathit{true} \to \mathit{in}?x;\ s := s \cup \{x\} \\
\mathit{mean}: & [\!]\ s \neq \emptyset \to \mathit{out}! \sum s / \# s \\
 & \mathbf{od}
\end{array}
$$

The system contains three labelled actions: *clear*, *add* and *mean*. The environment may execute *clear* to empty the set s or *add* to insert an element into s at any time. It may execute *mean* to compute the mean of s only if s is nonempty. The environment chooses the order of execution of the actions by communicating with the action-selection mechanism. One way in which this can be made explicit is mentioned in section 2.4.

Since the main concern of this thesis is to provide techniques for developing algorithms involving concurrency, the former view of action systems will be predominant.

1.3 Program Development

The aim of a developer is to develop an algorithm in steps, starting from a specification and progressing through a simple, or *abstract*, implementation into a complex but efficient, or *concrete*, implementation. The advantage of such a method is that the proof obligation at each step is small, whereas the result of all the steps is an implementation which would have been difficult or impossible to prove correct in a single step. The abstract implementation should provide a suitable starting place for the development which is simple enough that its correctness is acceptable.

At each development step, the system will be transformed in a way which makes it easier to implement on the target architecture, or which makes the system more efficient. The transformation should be shown to preserve the correctness of the system so that the final result of the development will be a working implementation. For a system which terminates, correctness means that the derived implementation terminates on all inputs for which the original implementation does—this is known as a liveness condition—and the results given by the derived implementation must also be allowed by the original—this is known as a safety condition. A terminating system may be modelled by a relation A between states which relates each state to all terminating states which the system may reach from that state, and to $\perp$ if it is possible for the action system not to terminate starting from that state. If the derived implementation is represented by a similar relation C, then it is correct if and only if $C \subseteq A$.

For systems which do not terminate, safety and liveness conditions also exist, but can depend on the system. Liveness conditions usually imply that the derived system is free from deadlock, and safety conditions imply that some relationship is preserved between the inputs and outputs of the system. Chapter 3 discusses the correctness of such systems.

Methods of development for terminating algorithms in procedural programming languages are well known (for example, [8]). Back has also developed methods for the stepwise development of action systems which terminate [3]. It is the intention of this thesis to explore methods of developing reactive action systems.

1.4 Refinement

When a system A is transformed into a system C as part of a development, we say that C *refines* A, or $A \sqsubseteq C$. The relation $\sqsubseteq$ is defined in a way which embodies one of the definitions of correctness, so that by proving the statement $A \sqsubseteq C$ we show that the transformation preserved correctness and was thus a valid step. The various refinement relations arising from different definitions of correctness are distinguished by modifying the symbol $\sqsubseteq$ or by adding a letter, and are discussed in chapter 3. The models of correctness used here are based on the possible sequences of states and communications which may be produced by executing an action system. For example, if every such sequence generated by C may also be generated by A, then $A \sqsubseteq C$. We may also perform refinements which change the state space of an action system. The $\sqsubseteq_R$ relation of chapter 3 is an example of such a relation.

Several techniques are available to prove refinement between two action systems A and C without going directly to the definitions of the refinement relations. Simulation, discussed in chapter 7, is the most important of these. It may be used to perform data refinements and refinements of atomicity.

1.4.1 Data refinement

A data refinement may be applied to a system by changing the state space of an action system and amending each action of the system in a manner consistent with this. Such a refinement may make the system easier or less costly to implement on some architecture. For example, in our statistical calculator we may replace the bag s by the two values $\sum s$ and $\#s$, thus reducing the amount of space required:

$$\begin{array}{ll}
 & \mathbf{var}\ t:\mathbb{R};\ n:\mathbb{N} \bullet t = 0 \land n = 0; \\
 & \mathbf{do} \\
\mathit{clear}: & \quad \mathit{true} \rightarrow t := 0;\ n := 0 \\
\mathit{add}: & \ [\!]\ \mathit{true} \rightarrow \mathit{in}?x;\ t := t + x;\ n := n + 1 \\
\mathit{mean}: & \ [\!]\ n \neq 0 \rightarrow \mathit{out}!t/n \\
 & \mathbf{od}.
\end{array}$$

Note that this system is indistinguishable from the earlier system if only the choice of actions, the values input and the outputs are examined. This kind of data refinement of a module is well known [9].

Such a data refinement may also be called a trace refinement, because the new system has actions which correspond to those of the original system and which are executed in the same order. A trace of a system is a record of the order in which its actions were executed. Some systems of notation such as CSP [11] have definitions of refinement which involve trace refinement directly. However, as refinement in this thesis is based on the states reached by an action system, it must be proved that trace refinement is a valid form of refinement in this model; this is done in chapter 7.

1.4.2 Refinement of atomicity

An important class of program transformation is provided by the refinement of atomicity. This involves increasing the number of actions in a system, introducing intermediate states which were previously hidden within an action or actions. Usually, a smaller number of coarse actions is split into a larger number of finer actions by refinement of atomicity. The importance of this is that it can allow the actions to be interleaved more, thus allowing greater freedom of implementation and more potential concurrency. It can also reduce data sharing in the actions of the system and allow the variables to be more distributed among the actions.

For example, an unordered buffer (as described in [5]) collects inputs and may output them in any order.

$$
\begin{array}{l}
\mathbf{var}\ s : \mathbf{bag}\ Msg \bullet s = \emptyset; \\
\mathbf{do}\ \overline{in} \rightarrow [\![\mathbf{var}\ m : Msg \bullet in?m;\ s := s \cup \{m\}]\!] \\
[\!]\ \overline{out} \wedge s \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in s;\ s := s - \{m\};\ out!m]\!] \\
\mathbf{od}
\end{array}
$$

The constructions $\overline{in}$ and $\overline{out}$ are *probes* [15], described in section 2.5.

By replacing the bag s with two bags t and u such that $t \cup u = s$, it is possible to distribute the variables of the system in such a way that the input action and the output action do not share variables and therefore may be implemented in different physical locations. However, an intermediate action must be added which transfers elements from t to u. Intermediate actions are introduced by adding the **null** action of section 2.7. Once this has been added, a simulation of the three actions may be written with the data refinement relation $t \cup u = s$:

$$
\begin{array}{l}
\mathbf{var}\ t, u : \mathbf{bag}\ Msg \bullet t = u = \emptyset; \\
\mathbf{do}\ \overline{in} \rightarrow [\![\mathbf{var}\ m : Msg \bullet in?m;\ t := t \cup \{m\}]\!] \\
[\!]\ t \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in t;\ t := t - \{m\};\ u := u \cup \{m\}]\!] \\
[\!]\ \overline{out} \wedge u \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in u;\ u := u - \{m\};\ out!m]\!] \\
\mathbf{od}.
\end{array}
$$

The middle action, which is the simulation of the **null** action, represents a communication link which will pass the messages from the processor containing the input action to that containing the output action. In chapter 5 this system will be developed into one which contains an actual communication link in the form of a channel between two processes. It will become apparent that each of these processes is an unordered buffer similar to that with which this development started. Also, in chapter 7, this development will be shown to satisfy the conditions for the simulation to be valid.

1.5 Outline of Thesis

Chapter 2 describes the notation and the semantics of action systems used in this thesis. Chapter 3 defines the notion of correctness and introduces the refinement relations. Chapter 4 contains a short example of the data refinement of a module. Chapter 5 defines the result of placing systems in parallel and lays the foundation for internal communication between action systems. Chapter 6 shows a proof in the notation of action systems of the correctness of a routing protocol, using the parallel operator defined in the previous chapter. Simulation is described and shown correct in chapter 7, and the following chapters contain examples of developments which use a combination of all the elements of theory so far presented. Finally, chapter 11 contains some theory on refinement of atomicity, based on a theorem of Back [1],[3]. This work is largely independent of the rest of the thesis. Back has already provided examples of its use, so it is presented here without applications.

A summary of this work, its conclusions and its relation to other work may be found in chapter 12.

2

Action Systems

2.1 Actions

An action system consists of a variable declaration, an initialisation and a set of actions, usually written:

$$\begin{array}{l}
\textbf{var } v : T \bullet \\
init(v); \\
\textbf{do} \\
\qquad \left(\underset{i \in I}{[\!]} A_i \right) \\
\textbf{od}
\end{array}$$

where *init* is a satisfiable predicate on the set of variables which specifies the initial condition, and each A_i for i in some index set I is an *action*. A finalisation is occasionally written after the set of actions, but this has been omitted here because its sole task is usually to delete some local variables from the system.

An action contains a *guard* and a *command*, which are together written: $g \rightarrow S$. The guard g is a predicate on the states of the action system, and the command S is a program which implements a transformation on the states of the action system. The command is written in the language of guarded commands as in Dijkstra[8] with extensions for input and output. The constructions used are the atoms (**skip**, **abort** and **magic**), assignment ($x := E$), input ($c?x$), output ($c!E$), sequential composition ($P;\ Q$), alternation (**if** ($[\!]_{j \in J}\, g_j \rightarrow S_j$) **fi**) and iteration (**do** ($[\!]_{j \in J}\, g_j \rightarrow S_j$) **od**). The communication actions will be described more fully later in this chapter.

An action is said to be *enabled* whenever the guard is satisfied, and in this case the command may be executed. The execution of an action is atomic; that is, the state may be observed before and after execution but no observation may be made while the command is in progress. The command may, however, cause input or output events which may be observed when execution is complete and which may provide information about the intermediate states of the command.

2.2 Semantics

An action system is executed by first choosing an initial state which satisfies the predicate for initialisation of the action system and then repeatedly choosing and executing an enabled action. The choice of which action among several enabled actions to execute at any instant is made nondeterministically. If at some point none of the actions is enabled then execution ceases and the state of the system is called a termination state of the system. Otherwise execution continues forever.

The result of executing an action system is a finite or infinite sequence of states. The first element in this sequence is the initial state. For each pair of adjacent states s and s' in the sequence, there exists an action whose guard is satisfied by s and whose command may transform s to s' in the usual semantics for the language of guarded commands. If execution reached a termination state then the state sequence is finite with the termination state as its last element. Otherwise the sequence is infinite. This state sequence will be called a *computation* of the action system. A state change at a particular point in the computation is defined by a pair of two adjacent states in the computation; this will be called a *transition*. By the definition of a computation, each transition is caused by the execution of an action (or, if the $\sharp$ operator from section 5.1 has been applied to the computation, a transition may be part of an action). An action which led to each transition (or list of transitions) may therefore be inferred; the sequence of actions which may be inferred from the list of all transitions of a computation will be called a *trace* of the action system. Several such traces may be possible inferences from a computation; whenever this occurs it does not matter which one is chosen. A computation may be built up by adding transitions together: adding a transition (s, s') to a computation c requires that the last element of c is s and consists of appending s' to c. Similarly, a computation may be broken into transitions: the first transition of a computation c is its first two elements, and it is removed by removing the first element of c.

The set of all possible computations of an action system P will be denoted by $\hat{P}$, and will be called the semantics of P.

An action system P may be given an extra liveness condition L, which is a predicate on computations which may be written in a notation similar to that of the liveness condition in the specification (section 3.1) or stated in terms of the order of execution of the actions. The semantics of such a system is $\hat{P} \cap L$; that is, the set of all possible computations of P satisfying L. The liveness condition may be used in several ways:

(a) to specify a fairness constraint or another property of action selection which cannot be written concisely in the notation of an action system, for example:

$$\lim_{n\to\infty} \Big(\#(\langle out\rangle_n \upharpoonright 1) \Big/ \#\langle out\rangle_n\Big) = \frac{1}{2}$$

where $\langle c\rangle$ represents the sequence of values communicated on channel c (see section 2.6), and $\langle c\rangle_n$ represents this value in the nth state of the computation (so that the above means that the number of outputs having value 1 is asymptotically equal to half the total number of outputs);

(b) to specify a constraint upon the values which are obtained from the environment, for example:

$$\forall n : \mathbb{N}\ \forall i \bullet i \in \langle in \rangle_n \to i = 0 \lor i = 1$$

(that is, every input is either 0 or 1), or

(c) to specify any other constraint which will be implemented at a later stage during the development.

If an action system is to be considered as 'code'—that is, a final system from a development—then it should contain only liveness conditions which belong to category (b) or which are fairness conditions from category (a). Chapter 3 will discuss the conditions under which a constraint upon the environment is justified.

Notice that using the liveness condition a program may be written in such a way that its computations are precisely the members of any given set S, provided S is nonempty. If the system P is defined to be $\mathbf{var}\ v : T \bullet \mathbf{do}\ \mathit{true} \to v :\in T\ \mathbf{od}$ and the liveness condition L is defined for each computation c to be $c \in S$ then $\hat{P}$ is the set of all possible computations and $\hat{P} \cap L$ equals S.

A predicate I on the states of an action system P with liveness condition L is an *invariant* of P if and only if each state in each computation from $\hat{P} \cap L$ satisfies I. The predicate I will be called a *strong invariant* of P if and only if I is satisfied by all possible initial states of P and each action $g \to S$ of P satisfies $(I \land g) \Rightarrow [S]I$ (where $[S]I$ is the weakest precondition for S to establish I). It is easily shown that any strong invariant of P is also an invariant of P.

In future sections, the semantics of a general action system P incorporating a liveness constraint will also be written $\hat{P}$, with the liveness condition being omitted for brevity.

2.3 Implementation

Back[3] and Misra and Chandy[6] illustrate how action-based systems can be implemented on different architectures. The semantics of an action system make possible a straightforward implementation in a sequential, procedural programming language. However, it may also be possible to divide an action system into several sets of actions or 'processes' such that each process has exclusive use of a subset of the system's local variables; the resulting processes may then be run on a network of processors, with one or more processes on each processor. Thus one goal of developing an action system is to partition the actions into processes which share as few variables as possible. Section 8.3 contains an example of such partitioning.

The fact that several different mappings from action systems to architectures exist means that one need not be concerned with the implementation details at the start of a development. By developing action systems in several refinement steps the developer can delay the introduction of a particular architecture until the later stages of the development. Then an action system may be transformed into one which is suitable for the target architecture.

2.4 Action Systems as Data Types

If the selection of which action to execute at any given time is influenced by the environment rather than being made internally by some nondeterministic mechanism then an action system can model a data type.

A data type is a set of states together with an initialisation and a set of named actions which act upon the state. An example is given in section 1.2.2. The selection of which action to execute is done by the environment and may be modelled with probes and I/O, for example (see the following sections for descriptions of probes and I/O). The parameters and results of the actions may be modelled either by the use of global variables or by communications, both of which may be observed in a computation of the system. The communications model of the example from section 1.2.2 may be written as follows.

$$
\begin{array}{l}
\mathbf{var}\ S : \mathbf{bag}\,\mathbb{R};\ x : \mathbb{R} \bullet \\
S = \emptyset; \\
\mathbf{do}\ \overline{clear} \rightarrow clear?x;\ S := \emptyset \\
\ [\!]\ \overline{add} \rightarrow add?x;\ S := S \cup \{x\} \\
\ [\!]\ \overline{mean} \wedge S \neq \emptyset \rightarrow mean! \sum S / \# S \\
\mathbf{od}
\end{array}
$$

The choice of which action the system will execute is made depending on the communications provided by the environment. In the case of the 'clear' action, the input from the environment is discarded; its existence is what directs the system to perform the action. This kind of system, which accepts inputs and provides outputs, is a *reactive* action system.

This reactive action system may be executed by a nondeterministic selection mechanism, but the communications within the system force the mechanism to follow the selections made by the environment. Moreover, internal actions may be added to the system to aid computation. They will not be guarded by communications, so they may be executed at any time. Provided that the resulting action system is a correct refinement of the original one, the environment will not be able to detect the extra actions.

2.5 An I/O Model

Input and output will be via synchronous communication events along channels, written $c?x$ and $c!x$ for input and output of x along channel c, respectively. An informal description of the instruction $c?x$ is that it will wait until another process is ready to execute an output instruction $c!E$ and then the two will take place simultaneously, resulting in the assignment $x := E$. The instruction $c!E$ behaves similarly. Chapter 5 will give a more formal description of these instructions when they are executed in action systems which run in parallel.

If selection of actions to execute is made nondeterministically then it may be possible for an I/O instruction to be encountered at a time when no other process is ready to communicate; the action system will then wait forever. In order to avoid this, communications may be guarded by probes [15]. A probe on the channel c is written $\overline{c?}$ or $\overline{c!}$, but this may be shortened to $\overline{c}$ when the context makes clear whether an input or an output is required. It is a boolean which is true if and only if the instruction $c?x$ or $c!E$ as appropriate is guaranteed to be completed without waiting. We may therefore write

$$g \wedge \overline{in} \rightarrow in?x$$

to denote an input event with guard g which will only be selected for execution at a time when it will be completed without waiting.

The addition of probes to the guard of an action system may make the system terminate prematurely, so an amendment is made to the semantics of an action system. If all guards of an action system should become false at a time when a change in the value of one or more probes would make one or more guards of the system true then the action system will not terminate but wait until a guard becomes true. That is, the set of actions

$$\{guard_i \wedge \overline{probe_i} \rightarrow command_i \mid i \in I\}$$

behaves as if augmented by the action

$$\Big(\bigwedge_{i \in I} \neg guard_i \vee \neg\overline{probe_i}\Big) \wedge \Big(\bigvee_{i \in I} guard_i\Big) \rightarrow \mathbf{skip}$$

except that this new action is not recorded in the computations of the action system. If the system should wait forever in this manner, then the resulting computation will be finite, and is indistinguishable from a terminating computation.

A probe gives in explicit notation a way for the environment to influence the selection of actions in an action system. As previously mentioned, this may be necessary in a system with internal selection in order to prevent the system from terminating unexpectedly. However, as chapter 5 will show, it is possible to execute an action system in such a way that the environment has an influence upon the selection of actions. In such a system the environment will never choose to execute an action containing a communication which will deadlock. In this case any probe in the guard of an action has no effect, but it serves as information, indicating that the action will not be executed unless a particular channel or set of channels is ready for communication. This method of selection is more suited to writing systems which behave like CSP processes when run in parallel, whereas internal action selection is more suited to writing systems which run in true concurrency with the environment.

If a communication is executed before its probe is true, then the communication will be delayed until the environment is ready to accept the communication. If this does eventually happen, then the probe will have become true just before the communication occurred. Since actions are executed atomically, no effects of the action are visible to the environment at the time when the probe changed, so the probe change affects the computation of the system by being recorded in the most recent state (similarly, if an action never terminates then any probe changes which occur will be recorded in the last element of the computation of the system).

2.6 Channel History

Whenever a communication occurs, it is possible for the action system to store details of the communication in a component of the state space of the system. However, since such a variable is often useful in specifying the behaviour of an action system, this will be done automatically as part of the semantics of the communication. The state of an action system at any instant therefore contains the special element $\langle comms \rangle$ of type $\mathbf{seq}(channel \times value)$ which contains the (finite) sequence of all past communication events in the order in which they occurred, with the earliest event first. An element of $\langle comms \rangle$ can be written $c.x$, meaning that the value x was communicated along channel c. In addition, it is also useful to have a list of all values communicated along a particular channel, so that for any channel c the symbol $\langle c \rangle$ will denote such a sequence, which can be written

$$\left\langle x \mid c.x \in \langle comms \rangle \right\rangle$$

(the notation $\langle x \mid y \in z \wedge p \rangle$ represents a *list comprehension*, and the order of the elements in the resulting sequence is that in which the elements appear in z).

The sequence $\langle comms \rangle$, as well as the values of all the probes, form part of the state space which is stored as elements in the computations of an action system.

It will sometimes be convenient to treat each channel history $\langle c \rangle$ as a separate element of the state and to dispense with $\langle comms \rangle$. This will be useful for systems in which only the order of communication events within a channel is constrained, and the interaction between channels is unspecified.

2.7 The Null Action

As chapter 7 will show, it is often useful to add to a system actions which do nothing. Such an action will be denoted **null**. The effect of this action is the same as that of the action $true \rightarrow \mathbf{skip}$; that is, to leave the state unchanged. However, the intent of the **null** action is to add only a finite number of repetitions to the computations of a system, so that the system remains live. Hence, all **null** actions in a system have the property that at any point in time they will eventually all cease to be enabled until an action is executed that is not **null**.

3

Refinement

3.1 Specification

A specification defines the possible computations of an action system by placing restrictions on the allowable computations and specifying the conditions under which the restrictions should be met. It will be written as a triple (r, I, l) in which r is called the *rely condition*[1], I is the *invariant* and l is the *liveness condition*.

The rely condition is a predicate on computations whose free variables are the global variables, the channel probes and the elements of the communication history variable $\langle comms \rangle$ which correspond to inputs. It may not involve any of the output events or local variables from the state space. It may involve global variables in order to place constraints upon the ways in which these variables may change under the influence of some external process. However, since it is difficult to distinguish such a change from one caused by the action system itself, we usually abandon global variables in favour of communication actions (see section 5.1). The rely condition specifies the allowable behaviours of the environment such that the action system may be reasonably expected to behave correctly. For instance, if the action system is required to continue processing inputs and producing results forever then the rely condition will specify that the environment is prepared to provide inputs forever, which means that in any computation the input probe is required to be true in infinitely many of the computation's states. This is such a common condition that the abbreviation Inf will be defined such that for any predicate p on states, $\mathrm{Inf}(p)$ is a predicate on computations which is true for a computation c if and only if either p is true on infinitely many elements of c or c is finite and p is true on the last element of c. This definition allows any system with a rely condition such as $\mathrm{Inf}(\overline{in})$ to assume that an input communication will eventually be completed even if the probe is false when the communication is attempted. This is because if c is finite with the last element satisfying p then the system failed to produce results forever because of some fault in the system rather than because the environment was unwilling to provide an input (note that section 2.5 defines that

[1] The reader may be reminded of work by C. B. Jones concerning rely and guarantee conditions; however, the words 'rely condition' are used here in a way not directly connected with that work.

any probe changes that occur during an action which never finishes are recorded in the last state of the computation).

The invariant is a predicate on states which may be lifted to computations by defining that a computation satisfies the invariant if and only if each of its elements satisfies the invariant. The invariant specifies a formula which must be an invariant of the action system whenever the rely condition is satisfied.

The liveness condition is a predicate on computations which may involve any state elements except for the channel probes and the values of any inputs (it may, however, involve the number of inputs which have occurred). The liveness condition specifies requirements which the action system must meet provided the rely condition is satisfied, such as the requirement that the system must continue producing outputs forever or in other words the size of $\langle out \rangle_n$ must tend to infinity as n tends to infinity (where $\langle out \rangle_n$ is the sequence $\langle out \rangle$ taken from the nth element of the computation).

An action system satisfies a specification (r, I, l) if and only if each of its computations satisfying r also satisfies I and l. The specification $A = (r, I, l)$ gives rise to a set of computations $\hat{A} \mathrel{\hat{=}} \{c \mid r(c) \to (I(c) \wedge l(c))\}$ such that an action system P satisfies A if and only if $\hat{P} \subseteq \hat{A}$.

Notice that for any nonempty set S it is possible to write a specification A such that $\hat{A} = S$—namely, $A = (true, true, l)$ where $l(c) \mathrel{\hat{=}} c \in S$.

The rely condition of an action system can be likened to the precondition of a program or instruction, and the invariant and liveness together may be likened to the postcondition. Given an action system, an invariant and a liveness condition, it is possible to find the weakest rely condition which guarantees that the action system satisfies the invariant and the liveness condition. An action system satisfies a specification (r, I, l) if and only if its weakest rely condition for I and l is implied by r.

3.2 Operation Refinement

A *refinement* relation is a relation between specifications, action systems or both such that if x is related to y and x is 'correct' then so is y. By *operation refinement* we mean that the state spaces of any pair of related entities are the same. The symbol for an operation refinement relation will be $\sqsubseteq$, and if $x \sqsubseteq y$ then we will say that x is refined by y. The process of obtaining x and/or y such that $x \sqsubseteq y$ will loosely be referred to as a refinement.

If A is a specification and P is an action system, then $A \sqsubseteq P$ will be defined to mean that A is satisfied by P; that is, $\hat{P} \subseteq \hat{A}$. If A and B are specifications then we will say that $A \sqsubseteq B$ if and only if $\hat{B} \subseteq \hat{A}$.

Proposition: $A \sqsubseteq B$ if and only if for any program P, $B \sqsubseteq P$ implies $A \sqsubseteq P$.

Proof: The proposition may be written: $\hat{B} \subseteq \hat{A}$ if and only if for any set $\hat{P}$, $\hat{P} \subseteq \hat{B}$ implies $\hat{P} \subseteq \hat{A}$, which is clear from set theory.

This shows that refinement is sound: if we refine a specification A by B then any program correctly derived from B will also satisfy A. It is also complete in that any B with this property does refine A.

Proposition: If $r \to r'$, $I' \to I$ and $r \wedge l' \to l$ then $(r, I, l) \sqsubseteq (r', I', l')$.

Proof:

$$
\begin{aligned}
& (r, I, l) \not\sqsubseteq (r', I', l') \\
= \;& \{c \mid r'(c) \to I'(c) \wedge l'(c)\} \not\subseteq \{c \mid r(c) \to I(c) \wedge l(c)\} \\
= \;& \exists c \bullet \neg[(r'(c) \to I'(c) \wedge l'(c)) \to (r(c) \to I(c) \wedge l(c))] \\
= \;& \exists c \bullet [\neg r'(c) \vee (I'(c) \wedge l'(c)] \wedge r(c) \wedge \neg(I(c) \wedge l(c)) \\
= \;& \exists c \bullet [r(c) \wedge \neg(I(c) \wedge l(c)) \wedge \neg r'(c)] \vee [I'(c) \wedge l'(c) \wedge \neg I(c) \wedge r(c)] \\
& \qquad \vee [I'(c) \wedge l'(c) \wedge \neg l(c) \wedge r(c)] \\
\Rightarrow \;& \exists c \bullet \neg[r(c) \to r'(c)] \vee \neg[I'(c) \to I(c)] \vee \neg[r(c) \wedge l'(c) \to l(c)] \\
= \;& \neg(r \to r' \wedge I' \to I \wedge (r \wedge l') \to l).
\end{aligned}
$$

In a similar manner we may prove that $(r, I, l) \sqsubseteq (r', I', l')$ if and only if $r \wedge \neg(I \wedge l) \to r'$, $I' \wedge l' \wedge r \to I$ and $I' \wedge l' \wedge r \to l$. These results may be used to give a proof that $A \sqsubseteq B$ for any two given specifications A and B between which refinement holds.

If P and Q are action systems, then we will say that $P \sqsubseteq Q$ if and only if $\hat{Q} \subseteq \hat{P}$. A soundness and completeness result applies to refinement between action systems similar to that for refinement between specifications.

Proposition: $P \sqsubseteq Q$ if and only if for any specification A, $A \sqsubseteq P$ implies $A \sqsubseteq Q$.

Proof: The proposition may be written: $\hat{Q} \subseteq \hat{P}$ if and only if for any set $\hat{A}$, $\hat{P} \subseteq \hat{A}$ implies $\hat{Q} \subseteq \hat{A}$, which is clear from set theory.

In some cases we may require to refine an action system P by another Q in such a way that some of the actions of P are hidden; for instance Q may contain an action which simulates two of the actions of P executed in sequence. Such a refinement will be valid if the environment of P never inspects variables whose changes are hidden. All communications made by P will still be visible to the environment even if a communication action is hidden, because the communications are all accessible through the special variable $\langle comms \rangle$. It may appear, however, that several communications have taken place during one action of Q, where they would have occurred separately in P.

A second type of refinement will be defined thus: $P \underset{\sim}{\sqsubset} Q$ if and only if for each computation c in $\hat{Q}$ there exists a computation a in $\hat{P}$ such that c is a subsequence of a which is infinite if a is infinite. This definition can be extended to the cases $A \underset{\sim}{\sqsubset} B$ and $A \underset{\sim}{\sqsubset} P$ where A and B are specifications. Effectively, $A \underset{\sim}{\sqsubset} P$ if each computation of P can be extended in a consistent manner to a computation which satisfies A, where extending in a consistent manner means that, for example, $\langle comms \rangle$ must not change between successive states except by having communications appended to it, and a channel probe must be true before a communication occurs.

3.3 Parallelism in Operation Refinement

As future chapters will show, it is often useful to allow finite repetition and parallelism in the computations of an action system. For example, finite repetition in an action system may be introduced with the **null** action of section 2.7, and parallelism is introduced by the parallel operator of section 5.1. Introducing parallelism into a computation means that collections of two or more adjacent transitions in the computation are added together into a single transition, thus making it appear that the transitions were executed concurrently.

Definition: For any computation c, $\natural\, c$ is c with adjacent repetitions removed. It is defined as the limit of the following applied to ever increasing prefixes of c:

$$\natural\langle\rangle = \langle\rangle$$
$$\natural\langle x\rangle = \langle x\rangle$$
$$\natural(x_0 : x_1 : xs) = \begin{cases} \natural(x_0 : xs) & \text{if } x_0 = x_1 \\ x_0 : \natural(x_1 : xs) & \text{if } x_0 \neq x_1 \end{cases}$$

where $a : as$ is the sequence as with the element a prepended.

This 'flatten' operator (which appears in Lamport, for instance in [13]) will be used to define refinement relations which ignore finite repetitions within computations. The definitions will exclude infinite repetitions because when the flatten operator is applied to such a computation the result will usually be a finite computation which does not end in a termination state, and this will not usually be a valid computation of the system being refined.

The definition of $\natural$ leaves open the possibility that one or more probes will change during an invisible action, and this will result in two adjacent states which are equal except in the probes. Since probe changes should be able to commute with such actions, it is intended that this situation should not occur, and so all the probe changes which occur during a sequence of repeated states will be moved to the next element of the computation before the repetitions are deleted. The probe changes thus appear to have occurred between the last repeated state and the state following it. Probe changes will later affect other kinds of refinement relation in a similar manner.

In order to deal with parallelism and repetition together, a relation will be defined which determines whether two computations are equal, modulo parallelism and repetition. The relation uses a bag E of state spaces which is such that any transition falling entirely within one state space from E may be executed concurrently with another transition which falls entirely within another state space from E. A transition is tested using the function *diffs*, where for any two states σ and τ, $diffs(\sigma, \tau)$ is the set of state elements, excluding probes, which differ between σ and τ. In the following definition, s_i denotes the ith element of the sequence s, $s \downarrow i$ denotes s with the first i elements removed, and s' denotes $s \downarrow 1$ ($s \downarrow i$ is the empty sequence whenever $i \geq \#s$).

Definition: For any bag E of state spaces, $=^E$ is the smallest commutative relation containing $\approx^E$. The relation $\approx^E$ relates the empty sequence to itself but no other sequence. It is defined on finite nonempty computations c and d as follows:

$$
\begin{array}{l}
c_0 = d_0 \\
\wedge \\
\quad c' =^E d' \\
\quad \vee \\
\quad \#c > 1 \wedge c_0 = c_1 \wedge c' =^E d \\
\quad \vee \\
\quad \exists D : \mathbf{seq}(\mathit{state\ space}) \bullet \\
\qquad \mathsf{bag}\, D \subseteq E \wedge 1 < \#D < \#c \wedge (c_0 : (c \downarrow \#D)) =^E d \\
\qquad \wedge \\
\qquad (\forall i : 0 \mathinner{..} \#D \bullet \mathit{diffs}(c_i, c_{i+1}) \subseteq D_i)
\end{array}
$$

and on infinite computations c and d as follows:

$$\forall M : \mathbb{N}\ \exists c', d' \bullet c' \leq c \wedge d' \leq d \wedge \#c' > M \wedge c' =^E d'$$

where $x \leq y$ means that the sequence x is a prefix of y.

This definition states that if $c =^E d$ then it is possible to find a sequence e which is a subsequence of each of $\natural\, c$ and $\natural\, d$ such that the states which appear in $\natural\, c$ but not in e are states which are allowed to have been omitted because of parallelism, and similarly for d.

In the case where E is empty, the relation $=^E$ relates two computations c and d if and only if $\natural\, c = \natural\, d$. Since this is a common relation, it will be denoted by writing $\natural$ for E instead of the empty bag. Thus the relation $=^\natural$ is one which compares for equality while ignoring repetitions. A similar notation will be also be used for the relations which are defined from $=^E$ whenever E is empty.

Note that while $=^\natural$ is an equivalence relation, $=^E$ is rarely an equivalence relation because it is not transitive in general. This does not usually matter, because in most cases equality using any one bag E is only used once in a development—at the instant when an action system is split into two or more running in parallel, with the state spaces of the smaller systems being the elements of E. However, it is true that if s and t are state spaces which are disjoint from each other and from each element of E, then

$$=^{E\cup\{s\cup t\}} ; =^{\{s,t\}} \subseteq =^{E\cup\{s\}\cup\{t\}} .$$

If $c =^{E\cup\{s\cup t\}} x$ and $x =^{\{s,t\}} d$ then the common subsequence e demonstrating that $c =^{E\cup\{s\}\cup\{t\}} d$ is the intersection of those from $c =^{E\cup\{s\cup t\}} x$ and $x =^{\{s,t\}} d$.

Variations of set membership and set inclusion will be defined from the equality relation as follows:

$$a \in^E S \mathrel{\widehat{=}} \exists b : S \bullet a =^E b$$
$$S \subseteq^E T \mathrel{\widehat{=}} \forall a : S \bullet a \in^E T.$$

Refinement with parallelism can now be defined thus: for two action systems or specifications X and Y, $X \sqsubseteq^E Y$ if and only if $\hat{Y} \subseteq^E \hat{X}$.

3.4 Data Refinement

It is common practice to apply a data refinement to the local variables of an action system: new local variables are added with a coupling invariant, then the actions are amended so that they depend upon the new local variables. This makes some of the original local variables redundant and they are removed. During this process the observable effects of the action system remain the same because only the local variables were affected and they are hidden from the environment. When data refinements are applied which affect the communications of an action system, this is no longer the case and some justification is needed for altering the observable behaviour of the system.

We may assume that the communications (and/or global variables) of the action system are required by an environment or user. Consider the system and the environment as a whole in which the communications of the action system are as local variables to the whole system which are not visible from the outside. We may then apply a data refinement which changes the communications of the action system but which leaves the whole system unchanged, and while doing this we may view the communications by their effect on the variable $\langle \mathit{comms} \rangle$ so that the usual, variable-based data refinement technique can be used on the communications.

The usual way of carrying out a data refinement is to choose a coupling invariant for the whole system, add in the new variables, and then take away any auxiliary variables. This is sometimes done in one step, with the result that some variables are simply replaced by new ones, but this should be accompanied by a proof that the removed data was auxiliary. If the action system A has an environment which can be written as a process E, and A is data-refined to A', then this amounts to a proof that there exists E' such that $A \parallel E \sqsubseteq A' \parallel E'$ (where $\parallel$ is some parallel operator). A special case of this is when $E' = E \parallel T$ for some T, a translator which reverses the data refinement on A in such a way that E is still correct (this does not necessarily imply that $A \sqsubseteq A' \parallel T$, but it does mean that $A \parallel E \sqsubseteq A' \parallel T \parallel E$: the former is a further special case of the condition). If the data refinement on A is one-to-many, then a translator T may always be found such that $A = A' \parallel T$ and this condition will be satisfied. However in any case we have that A and $A' \parallel T$ are so similar that E cannot distinguish between them. This implies an equivalence relation on computations: $a \sim b$ if a and b cause E to behave identically when they are computations of a process which is run in parallel with E. If a translator T exists, then we have that for any computation b of $A' \parallel T$ there is a computation a of A such that $a \sim b$. If the coupling relation which connects A to A' is R, then T is likely to resemble R^{-1}, and $\sim$ is likely to be the transitive closure of $R;\ R^{-1}$.

For example: an environment takes a sequence of natural numbers between 10 and 20 for input, and keeps a running count of odd numbers in the sequence. If the data refinement R consists of changing each natural number to its remainder after division by 2, then a new environment could keep count by simply adding up the inputs. Because the (old) environment treats all even numbers the same and all odd numbers the same, it will not be able to tell the difference between, say, the sequences $\langle 11, 13, 16, 14, 19\rangle$ and $\langle 11, 11, 10, 10, 11\rangle$. The equivalence relation $\sim$ equates all sequences whose lengths are the same and whose nth elements differ by an even number for each n. Thus $\sim$ is exactly $R;\ R^{-1}$. Given the data-refinement of the above sequences, which is $\langle 1, 1, 0, 0, 1\rangle$, the translator T could simply add 10 to each number, giving the second of the two sequences. This would be a valid translator because it is a subset of R^{-1} and for any sequence s we have that $(R;\ T)(s)$ is a member of $(R;\ R^{-1})(s)$ and is therefore equivalent to the original s.

This forms the basis of a method for data refinement on action systems in which the observables of the system may change as well as the local variables. Whether E can be written as a process or not, we assume that there is an equivalence relation $\sim$ on computations of an action system or specification A such that it can be proven that if $a \sim b$ then a and b are equally acceptable as computations of A and will be indistinguishable to E. This relation gives rise to equivalence classes on the computations of A; for any computation a let $\mathcal{E}(a)$ be the equivalence class under $\sim$ of which a is a member. We will define data refinement in terms of a relation R which relates states of the original specification or action system, or 'abstract' states, to states of the new one, or 'concrete' states. It will be necessary to apply R^{-1} to computations, and this will be defined as the limit of the following applied to ever-increasing prefixes of a computation.

$$R^{-1}\langle\rangle = \{\langle\rangle\}$$
$$R^{-1}(c : cs) = \begin{cases} \{(a : as) \mid a\ R\ c \wedge as \in R^{-1}(cs)\} & \text{if } c \in \operatorname{ran}(R) \\ R^{-1}(cs) & \text{if } c \notin \operatorname{ran}(R) \end{cases}$$

As with the $\natural$ operator, R^{-1} may transform an infinite computation into a set of finite computations; the members of this set will usually fail to satisfy the abstract liveness condition. Note that $R^{-1}(c)$ is always non-empty; the least that it can contain is the empty computation.

Let R be lifted to computations by defining $a\ R\ c$ to be true for computations a and c if and only if $a \in R^{-1}(c)$.

We will say that if A is a specification and P an action system, then $A \sqsubseteq_R P$ if and only if for each computation c in $\hat{P}$ there exists a computation a in $\hat{A}$ such that $R^{-1}(c) \subseteq^\natural \mathcal{E}(a)$. In other words, it should be possible to 'translate' each computation of the action system into an acceptable abstract computation (using R^{-1}) in such a way that if c is a concrete computation corresponding to the abstract computation a then all possible translations of c should appear to be equivalent to a according to the environment's equivalence relation $\sim$. The presence of the $\natural$ operator in this definition allows the possibility that the concrete action system could introduce some new actions which produce only hidden effects in the concrete system. Such actions change the state, but when R^{-1} is applied to the state change it turns into a repetition.

In a similar manner, we will say that $A \sqsubseteq_R^E P$ if and only if for each computation c in $\hat{P}$ there exists a computation a in $\hat{A}$ such that $R^{-1}(c) \subseteq^E \mathcal{E}(a)$. In addition, data refinement between two specifications and between two action systems will be defined in a manner identical to that between a specification and an action system.

Proposition: $A \sqsubseteq_R^E B$ if and only if for each concrete action system P, $B \sqsubseteq^\natural P$ implies $A \sqsubseteq_R^E P$, provided that $\exists b : R^{-1}(c) \bullet b \in^E \mathcal{E}(a)$ implies $R^{-1}(c) \subseteq^E \mathcal{E}(a)$. This condition may be written in a number of ways; for example it is true if R is a one-to-many relation on states or if $R;\ R^{-1} \subseteq \sim$ and $\sim;\ =^E \subseteq =^E;\ \sim$.

Proof: To show forward implication, suppose $B \sqsubseteq^\natural P$ and $A \sqsubseteq_R^E B$. Let $c' \in \hat{P}$ and find $c \in \hat{B}$ such that $c' =^\natural c$, which is possible since $B \sqsubseteq^\natural P$. Let S be a subset of R such that $S^{-1}(s)$ is a singleton for each state s (unless $R^{-1}(s)$ is empty) and let $x = S^{-1}(c)$ and $y = S^{-1}(c')$, so that $x \in R^{-1}(c)$ and $y \in R^{-1}(c')$ and $x =^\natural y$. Then, since $A \sqsubseteq_R^E B$ and thus there exists a in $\hat{A}$ such that $R^{-1}(c) \subseteq^E \mathcal{E}(a)$, it must be the case that $y \in^E \mathcal{E}(a)$. The conditions of the proposition then give that $R^{-1}(c') \subseteq^{\mathcal{E}} (a)$ and therefore $A \sqsubseteq_R^E P$.

Implication the other way is simply proved by letting $\hat{P}$ equal $\hat{B}$.

Proposition: $P \sqsubseteq_R^E Q$ if and only if for each abstract specification A, $A \sqsubseteq^\natural P$ implies $A \sqsubseteq_R^E Q$, provided that $=^\natural$ is a subset of $\sim$.

Proof:

$$
\begin{aligned}
& \forall A \bullet A \sqsubseteq^\natural P \to A \sqsubseteq_R^E Q \\
= \ & \forall A \bullet \hat{P} \subseteq^\natural \hat{A} \to (\forall c : \hat{Q}\ \exists a : \hat{A} \bullet R^{-1}(c) \subseteq^E \mathcal{E}(a)) \\
= \ & \forall c : \hat{Q}\ \exists a' : \hat{P} \bullet R^{-1}(c) \subseteq^E \mathcal{E}(a') \ \{\mathcal{E}(a) = \mathcal{E}(a') \text{ whenever } a =^\natural a'\} \\
= \ & P \sqsubseteq_R^E Q.
\end{aligned}
$$

If p is a predicate on states then let $\mathrm{mk}\,p$ be the relation $\{(x,y) \mid p(y)\}$ and $\mathrm{id}\,p$ be the relation $\{(x,x) \mid p(x)\}$.

Proposition: If $A = (r, I, l)$ and $C = (r', I', l')$ are specifications and if the relation $\sim$ satisfies $a \sim \natural\, a$ and $a \sim b \to (a \in \hat{A} \leftrightarrow b \in \hat{A})$ then sufficient conditions for $A \sqsubseteq_R C$ are:

(a) $\forall c \bullet (I'(c) \vee \neg r'(c)) \to (\exists a : R^{-1}(c) \bullet R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a))$
(b) $\mathrm{mk}\,r;\ R \subseteq \mathrm{mk}\,r'$
(c) $\mathrm{mk}\,l' \subseteq \mathrm{mk}\,l;\ R$
(d) $R;\ \mathrm{id}\,I' \subseteq \mathrm{id}\,I'R$

(of which (a) is automatically true if $R;\ R^{-1} \subseteq \sim$).

Proof:

$$
\begin{array}{lll}
 & A \sqsubseteq_R C & \\
= & \forall c : \hat{C}\ \exists a : \hat{A} \bullet R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a) & \\
= & \forall c : \hat{C}\ \exists a : R^{-1}(c) \bullet R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a) & (1) \\
\wedge & \forall c : \hat{C}\ \exists a : R^{-1}(c) \bullet a \in \hat{A} & (2)
\end{array}
$$

{if an a' can be found satisfying (2) and an a satisfying (1), then since $a' \in R^{-1}(c)$ and $R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a)$ we have $a'(=^{\natural};\ \sim)a$ and therefore, by the conditions of the proposition, since a' is a member of $\hat{A}$, so is a.}

$$
\begin{array}{llll}
(1) & \Leftarrow & (a) & \text{\{by weakening the antecedent\}} \\
(2) & = & \forall c \bullet (r'(c) \to I'(c) \wedge l'(c)) \to (\exists a : R^{-1}(c) \bullet a \in \hat{A}) & \\
 & \Leftarrow & \forall c \bullet I'(c) \to (\forall a : R^{-1}(c) \bullet I(a)) & (3) \\
 & \wedge & \forall c \bullet l'(c) \to (\exists a : R^{-1}(c) \bullet l(a)) & (4) \\
 & \wedge & \forall c \bullet \neg r'(c) \to (\exists a : R^{-1}(c) \bullet \neg r(a)) & (5)
\end{array}
$$

in which (3) = (d), (4) = (c) and (5) $\Leftarrow$ (b).

If for three entities A, B and C we have $A \underset{\sim}{\sqsubseteq} B$ and $B \sqsubseteq_R C$, then we will say that $A \underset{\sim}{\sqsubseteq}_R C$. This definition of refinement allows arbitrary finite contiguous subsequences of $\hat{A}$ to be refined by other sequences of actions in $\hat{C}$, each sequence corresponding to a single action of B.

3.5 Combinations of Refinements

Since this chapter has introduced several different refinement symbols, it will sometimes occur that several are used in a development and therefore it will be useful to know how some of these behave in combination. In addition, although $\sqsubseteq^{\natural}$ is clearly transitive, it is less clear which of the other relations are transitive. Some of the combinations of refinement relations will be simplified in this section.

3.5.1 E-Refinement and R-Refinement

Proposition: If F is a bag such that $(\sim;\ =^E) \subseteq (=^F;\ \sim)$ then $\sqsubseteq^E;\ \sqsubseteq_R \subseteq \sqsubseteq^F_R$.

Proof: Let $X \sqsubseteq^E Y \sqsubseteq_R Z$.

$$
\begin{array}{rll}
& \hat{Y} \subseteq^E \hat{X} \ \ \wedge \ \ (\forall c : \hat{Z}\ \exists a : \hat{Y} \bullet R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a)) & \\
\Rightarrow & \forall c : \hat{Z}\ \exists a : \hat{Y}\ \exists a' : \hat{X} \bullet & \\
& a =^E a' \wedge R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a) & (1) \\
(1) \ = & \forall x : R^{-1}(c) \bullet a =^E a' \wedge x \in^{\natural} \mathcal{E}(a) & \\
= & \forall x : R^{-1}(c) \bullet x(=^{\natural};\ \sim;\ =^E)a' & \\
\Rightarrow & \forall x : R^{-1}(c) \bullet x(=^{\natural};\ =^F;\ \sim)a' & \\
= & \forall x : R^{-1}(c) \bullet x(=^F;\ \sim)a' & \\
= & R^{-1}(c) \subseteq^F \mathcal{E}(a') &
\end{array}
$$

and hence $X \sqsubseteq^F_R Z$.

Proposition: If F is a bag such that $(=^E;\ R^{-1}) \subseteq (R^{-1};\ =^F)$ then $\sqsubseteq_R;\ \sqsubseteq^E \subseteq \sqsubseteq^F_R$.

Proof: Let $X \sqsubseteq_R Y \sqsubseteq^E Z$. Take an element c' from $\hat{Z}$. Since $Y \sqsubseteq^E Z$, there exists an element c of $\hat{Y}$ such that $c =^E c'$. Since $X \sqsubseteq_R Y$, there exists an element a of $\hat{X}$ such that $R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a)$. If y is an element of $R^{-1}(c')$ then:

$$
\begin{array}{rll}
& c(=^E;\ R^{-1})y & \\
\Rightarrow & c(R^{-1};\ =^F)y & \\
= & \exists z : R^{-1}(c) \bullet y =^F z & \\
\Rightarrow & \exists z \bullet y =^F z \wedge z(=^{\natural};\ \sim)a & \{\text{since } R^{-1}(c) \subseteq^{\natural} \mathcal{E}(a)\} \\
= & y(=^F;\ =^{\natural};\ \sim)a & \\
= & y(=^F;\ \sim)a &
\end{array}
$$

so $R^{-1}(c') \subseteq^F \mathcal{E}(a)$ and hence $X \sqsubseteq^F_R Z$.

3.5.2 Two R-refinements

Proposition: If $=^{\natural} \subseteq \sim$ and $(\sim;\ R^{-1}) \subseteq (R^{-1};\ \sim)$ then $\sqsubseteq_R;\ \sqsubseteq_S \subseteq \sqsubseteq_{R;\,S}$.

Proof: Let $X \sqsubseteq_R Y \sqsubseteq_S Z$. Take an element c from $\hat{Z}$. Since $Y \sqsubseteq_S Z$, there exists an element b of $\hat{Y}$ such that $S^{-1}(c) \subseteq^{\natural} \mathcal{E}(b)$ and since

$X \sqsubseteq_R Y$ there exists an element a of $\hat{X}$ such that $R^{-1}(b) \subseteq^\natural \mathcal{E}(a)$. If x is an element of $R^{-1}(S^{-1}(c))$ then:

$$
\begin{array}{ll}
 & \exists y : S^{-1}(c) \bullet x \in R^{-1}(y) \\
\Rightarrow & \exists y \bullet x \in R^{-1}(y) \wedge b(\sim;\ =^\natural)y \qquad\qquad \{\text{since } S^{-1}(c) \subseteq^\natural \mathcal{E}(b)\} \\
= & b(\sim;\ =^\natural;\ R^{-1})x \\
\Rightarrow & b(R^{-1};\ \sim)x \\
= & \exists z \bullet z \in R^{-1}(b) \wedge z \in \mathcal{E}(x) \\
\Rightarrow & \exists z \bullet z \in \mathcal{E}(a) \wedge z \in \mathcal{E}(x) \qquad\qquad \{\text{since } R^{-1}(b) \subseteq^\natural \mathcal{E}(a)\} \\
= & x \in \mathcal{E}(a)
\end{array}
$$

so $R^{-1}(S^{-1}(c)) \subseteq \mathcal{E}(a)$ and hence $X \sqsubseteq_{R;\,S} Z$.

3.5.3 R-refinement and Subtrace-refinement

Let $\preceq$ denote the relation between computations such that $x \preceq y$ if and only if x is a subsequence of y which is infinite if y is infinite. Thus $X \underset{\sim}{\sqsubseteq} Y$ means that for each computation y of Y there exists a computation x of X such that $y \preceq x$.

The combination $\underset{\sim}{\sqsubseteq};\ \sqsubseteq_R$ is defined to equal $\underset{\sim}{\sqsubseteq}_R$. Combining the relations in the other order is slightly more difficult.

Proposition: If $\sim$ is pointwise (that is, $(x : xs) \sim (y : ys) \Leftrightarrow \langle x\rangle \sim \langle y\rangle \wedge xs \sim ys$), then $\sqsubseteq_R;\ \underset{\sim}{\sqsubseteq}\ \subseteq\ \underset{\sim}{\sqsubseteq}_R$, provided that $\underset{\sim}{\sqsubseteq}$ only modifies computations in a way which preserves the property 'contains infinitely many states which are in the image of R'.

Proof: Let $X \sqsubseteq_R Y \underset{\sim}{\sqsubseteq} Z$. Take an element d from $\hat{Z}$. Since $Y \underset{\sim}{\sqsubseteq} Z$ there exists an element c of $\hat{Y}$ such that $d \preceq c$. Since $X \sqsubseteq_R Y$ there is an element a of $\hat{X}$ such that $R^{-1}(c) \subseteq^\natural \mathcal{E}(a)$. If x is an element of $R^{-1}(d)$ then there exists an element y of $R^{-1}(c)$ such that $x \preceq y$. This is because the effect of inserting extra elements into d to form c can be matched by inserting extra elements into x to form y (as long as infinitely many elements do not need to be added—which is prevented by the conditions of the proposition).

Since $y \in R^{-1}(c)$, y is related to an element of $\mathcal{E}(a)$ by the relation $=^\natural$. It is possible to find an a' such that $a' \preceq a$ and $x \in^\natural \mathcal{E}(a')$. This is done by deleting elements from a to make a' with the deleted elements corresponding to those which are present in y but not in x. Since these in turn correspond to those which are present in c but not d, the computation a' which results is independent of x and y.

This shows that if x is an element of $R^{-1}(d)$ then there exist computations a and a' such that $x \in^\natural \mathcal{E}(a')$, $a' \preceq a$ and $a \in \hat{X}$. Hence, for some system Y', $X \underset{\sim}{\sqsubseteq} Y' \sqsubseteq_R Z$ so that $X \underset{\sim}{\sqsubseteq}_R Z$.

3.5.4 Two Subtrace-refinements

Since the relation $\preceq$ is transitive it is clear that the refinement relation $\underset{\sim}{\sqsubset}$ is transitive also. This, in addition with the previous subsections, shows that the following also hold providing the combined side conditions are satisfied.

$\underset{\sim}{\sqsubset};\ \underset{\sim}{\sqsubset}_R = \underset{\sim}{\sqsubset}_R$

$\underset{\sim}{\sqsubset}_R;\ \underset{\sim}{\sqsubset} \subseteq \underset{\sim}{\sqsubset}_R$

$\underset{\sim}{\sqsubset}_R;\ \underset{\sim}{\sqsubset}_S \subseteq \underset{\sim}{\sqsubset}_{R;\,S}$

3.5.5 Remarks

There are several combinations which do not feature in this section. In particular, there are several combinations involving the $\sqsubseteq^E$ relation which do not have concise solutions given the way that this relation is defined in this chapter. The $\sqsubseteq^E$ relation is not transitive except under certain strict conditions. Fortunately, most combinations rarely occur. (See also section 12.3).

4

The Devil's Dice

This short example demonstrates a data refinement on an action system which is written as a datatype. The data refinement is done not so that the data types are easier to implement, but in order to make a search algorithm faster. This solution is based on one which is presented in graph theory by Busacker and Saaty[4].

4.1 The Problem

The 'Devil's Dice' problem consists of four cubes. Each face of each cube is painted in one of four different colours. The problem is to stack up the four cubes into a tower in such a way that each of the four sides of the tower contains four distinctly coloured cube faces. The problem may have several solutions, or none at all. For convenience, the four sides of the tower will be referred to using the points of the compass.

If a solution exists, it may be found by searching through all combinations. One possible method is a brute-force search, while another is a backtrack search method. However, in this chapter we will examine a datatype which is used by these search methods without concentrating on the details of the search. For simplicity, the search may be assumed to be a brute-force search.

4.2 A Datatype

The local variable space of the datatype action system will contain the four coloured cubes and their orientations. The action system should be able to input a set of cubes, test for a solution, output a solution, and rotate any of the cubes so that a different combination can be tested. The action system will be written with labelled actions and an external selection mechanism will be used to execute the actions. The actions are as follows:

$input$: $true \rightarrow$ input 4 cubes
rot_a: $true \rightarrow in?n$; rotate cube n through 90° about a vertical axis
$rot2_a$: $true \rightarrow in?n$; rotate cube n through 180° about a vertical axis

rot_b: $true \rightarrow in?n$; rotate cube n through 90° about a North-South axis
$rot2_b$: $true \rightarrow in?n$; rotate cube n through 180° about a North-South axis
$test$: $true \rightarrow out!solution$
$output$: $solution \rightarrow$ output 4 cubes

where *solution* is a function which returns true if and only if for each face of the tower each cube has a different colour on that face.

The system which is to find a solution to the problem will first input the cubes, then repeatedly make one or more rotations followed by a test. Provided the test indicates that the current set of cubes is a solution, the system may ask for the solution to be output. The system may continue to search for more solutions if desired. It can be shown that the sequence of actions:

$rot2_a;\ rot2_b;\ rot2_a;\ rot_b;\ rot2_a;\ rot2_b;\ rot2_a;\ rot_a$

when repeated three times on any one cube will visit each possible orientation of the cube precisely once before returning the cube to its original position. The system which seeks a solution may visit each possible combination of cubes in turn using this sequence applied repeatedly to the cubes—though it may skip over some combinations since, for example, each possible tower appears eight times in the list of all combinations due to rotational symmetry.

4.3 A Data Refinement

A data refinement R between the above system and a new one will relate the two if and only if each of the new cubes can be re-oriented to its position in the old system by using only $rot2_a$ and $rot2_b$ actions. This makes two of the actions much simpler. Section 3.4 defines a data refinement to be valid if each concrete computation c is related by R^{-1} to an abstract computation a, and if also each member of $R^{-1}(c)$ is a member of $\mathcal{E}(a)$. The latter condition will be dealt with in the following section. Informally, we can see that the following system satisfies the former condition, because each action is an equivalent of an action of the abstract system modulo the relation R (this will be dealt with more formally in chapter 7).

$input'$: $true \rightarrow$ input 4 cubes
rot'_a: $true \rightarrow in?n$; rotate cube n through 90° about a vertical axis
$rot2'_a$: $true \rightarrow in?n$; **skip**
rot'_b: $true \rightarrow in?n$; rotate cube n through 90° about a North-South axis
$rot2'_b$: $true \rightarrow in?n$; **skip**
$test'$: $true \rightarrow out!solution'$
$output'$: $solution' \rightarrow$ output 4 cubes

In this system the $solution'$ function returns true if and only if a solution can be found by using any number of rotations from $rot2_a$ and $rot2_b$.

The advantage of the new system is that, since two of the actions no longer have any effect on the system, the sequence of actions which visits every position of a cube is reduced from the above sequence of 24 actions to just 6:

rot'_b; rot'_a; rot'_b; rot'_a; rot'_b; rot'_a.

This divides the time taken to find a solution by 4 for each cube on which the above sequence of actions is used.

The disadvantage of the system is that, seemingly, the *solution'* function has been made more complicated, and the system no longer outputs a direct solution.

4.4 Translating the Solution

It may be observed that applying rotations from $rot2_a$ and $rot2_b$ to a cube has the effect of exchanging one or both of the North-South and East-West pairs of faces, and that moreover any combination of those exchanges may be achieved with rotations from $rot2_a$ and $rot2_b$. Thus the *solution'* function, which originally tested whether each face of the tower contained one face in each of the four colours, now tests whether this can be achieved by exchanging pairs of opposite faces. As demonstrated below, this is equivalent to testing whether each pair of opposite faces of the tower contains two faces in each of the four colours. Thus the *solution* function has not been significantly increased in complexity by the data refinement.

The remaining task is to show that the second condition for data refinement described above is satisfied: given a computation a of the abstract system and another c of the concrete system, $R^{-1}(c)$ is required to be a subset of $\mathcal{E}(a)$. This is immediate if we define the equivalence relation $\sim$ to relate two states if and only if one can be obtained from the other using any number of rotations from $rot2_a$ and $rot2_b$. However, in order to satisfy the user of the system it must be shown that the definition of $\sim$ was not too weak. In other words, the non-solutions produced by the refined action system are acceptable because they can be transformed into solutions with relative ease. The process which does this is similar to the translator T described in section 3.4. Given a set of cubes for which the *solution'* function returns true, the translator will produce a true solution to the problem. It may happen that the solution is not unique, but the translator given here will produce only one solution.

The translator will use the technique of exchanging opposite faces. The algorithm will be applied twice: first to the North and South faces of the tower, using $rot2_a$ to exchange the North and South faces of the cubes, and then to the East and West faces of the tower, using $rot2_b$ to exchange the East and West faces of the cubes while keeping the North and South faces fixed. Before the algorithm is applied to each pair of opposite faces of the tower, the two faces together contain two cube faces in each colour, and afterwards each face of the tower contains one cube face in each colour. The algorithm applied to the North and South faces of the tower is as follows.

Let S be the set of four cubes. Let F be a subset of S which is initially empty. Repeat the following for as long as S is not a solution (that is, as long as the four North faces of S are not distinct):

> Let c be an element of F and d be an element of S such that the North faces of c and d are the same colour, if possible; otherwise let c and d be elements of S whose North faces are the same colour and let $F := F \cup \{c\}$. This is always possible because the four North faces of S are not distinct. Exchange the North and South faces of d to give d'. Let $S := (S \setminus \{d\}) \cup \{d'\}$ and $F := F \cup \{d'\}$. For as long as there exists e in $S \setminus F$ such that the South face of e is the same colour as the North face of d', let $F := F \cup \{e\}$ and $d' := e$.

This algorithm has as an invariant the fact that the North faces of all the cubes in F are distinct, and since the size of F increases on each pass the algorithm must terminate with the cubes in a configuration which is a solution to the problem.

5

The Parallel Operator

5.1 Semantic Parallel

One of the aims of action-system development is to partition an action system so that it can be implemented as two or more processes in parallel. An explicit parallel operator in the language serves to define such partitioning whenever communication between the processes is via channels rather than shared variables, and also allows the developer to make a separate refinement on each process after the partitioning.

A 'semantic parallel' operator will be defined which specifies the result of placing two action systems in parallel in terms of their computations. The operator will be written $\hat{\|}$. First it is necessary to define a 'stretch' relation which serves as the 'glue' between two action systems at different levels of atomicity.

Definition: If a and b are computations on a state space, and ch is a set of communication channels, then $a \underset{ch}{\sharp} b$ is true if and only if there exist infinite increasing prefixes $\{a_n\}$ and $\{b_n\}$ of a and b respectively such that $a_n \underset{ch}{\sharp}^{\perp} b_n$, where $\perp$ is an element which compares unequal to every state, and

$$
\begin{aligned}
&\langle\rangle \underset{ch}{\sharp}^{x} \langle\rangle \Leftrightarrow \mathit{true}\\
&\langle\rangle \underset{ch}{\sharp}^{x} (b : bs) \Leftrightarrow x = b \wedge \langle\rangle \underset{ch}{\sharp}^{x} bs\\
&(a : as) \underset{ch}{\sharp}^{x} (b : bs) \Leftrightarrow\\
&\qquad [a = b \wedge as \underset{ch}{\sharp}^{a} bs] \vee [x = b \wedge (a : as) \underset{ch}{\sharp}^{x} bs] \vee\\
&\qquad [(a \upharpoonright ch) \neq (b \upharpoonright ch) \wedge (b \setminus ch) = (x \setminus ch) \wedge (a : as) \underset{ch}{\sharp}^{b} bs]
\end{aligned}
$$

where $s \setminus T$ is the projection of s on to a state space not containing elements from T.

In essence, if $a \underset{ch}{\sharp} b$ then b must be the same as a but with repetitions and extra elements added in such a way that each extra element agrees with the preceding member of b except possibly in its communications on channels from ch (note that since b is a computation the extra communications are not arbitrary; they must appear in the following members of b also). The effect of adding these extra elements is to split an atomic communication of several values into several communications in sequence. If the atomic communication is accompanied by a change in the state (other than in the communication histories of channels from the set ch) then in the new computation this state change accompanies the last in the sequence of communications. The relation $\underset{ch}{\sharp}$ will be raised to an operator on sets as usual:

$$\underset{ch}{\sharp} S \mathrel{\hat{=}} \{b \mid (\exists a : S \bullet a \underset{ch}{\sharp} b))\}.$$

Definition: If P and Q are action systems with no shared variables then c is a computation of $P \mathbin{\hat{\|}} Q$ if and only if there exists a computation d on the states $(states\ P) \times (states\ Q)$ and channels $(\alpha P) \cup (\alpha Q)$ such that c equals d with channels from the set $ch = (\alpha P) \cap (\alpha Q)$ hidden, and $d \restriction P \in \underset{ch}{\sharp} \hat{P}$ and $d \restriction Q \in \underset{ch}{\sharp} \hat{Q}$, where αX is the set of channels on which action system X communicates and $x \restriction X$ is the computation x after projecting each of its elements on to the state space of action system X.

When P is executed in parallel with Q, the result is a computation which looks like a computation of P when projected on to the state space of P, and similarly for Q. The internal communications (that is, communications between P and Q) are hidden. The definition allows for action systems of different levels of atomicity to be run in parallel because the $\sharp$ operator reduces the common communications to a level of atomicity which is at least as fine as that of each system. The effect of this is not observed because the common communications are hidden. The definition allows for non-communicating actions of P and Q to be executed alone or in true concurrency. A specification may detect a difference between these situations, but a refinement relation $\sqsubseteq^E$ in which E contains as elements the state spaces of P and Q will hide this difference.

There are two restrictions in the definition of the parallel operator. Firstly, P and Q must not share variables. Variable sharing between two processes may be simulated using communication along channels, by placing an output instruction in one action system whenever it wishes to influence a shared variable owned by the other, and an input instruction at any place where a variable is allowed to change by the influence of the other system. This has the advantage that any external influence on a particular variable may be controlled by guarding the input action, or by placing extra constraints in the system's rely condition. Secondly, the internal communications of P and Q are hidden, which means that a third system cannnot synchronise on those communications, as is possible in CSP for instance. This can be simulated by having one of each pair of communicating instructions copy the communication to an external channel, where it can be observed. This has the advantage that the atomicity of this communication will be that dictated by the communicating action system; it is not reduced to a finer level by the $\sharp$ operator. Thus these two restrictions do not result in loss of generality.

Proposition: $\hat{\|}$ is an associative and commutative operator which is monotonic with respect to $\sqsubseteq^{\natural}$.

Proof: $\hat{\|}$ is commutative because the definition of $P \mathbin{\hat{\|}} Q$ is symmetric in P and Q. It is associative because $(P \mathbin{\hat{\|}} Q) \mathbin{\hat{\|}} R$ can be rewritten as a formula which is symmetric in P, Q and R:

$$
\begin{array}{rl}
& c \text{ is a computation of } (P \mathbin{\hat{\|}} Q) \mathbin{\hat{\|}} R \\
= & \exists d \bullet c = d \setminus (\alpha PQ \cap \alpha R) \\
& \qquad \wedge \;\; d \restriction PQ \in \underset{\alpha PQ \cap \alpha R}{\sharp} \widehat{PQ} \;\; \wedge \;\; d \restriction R \in \underset{\alpha PQ \cap \alpha R}{\sharp} \hat{R} \\
& (\text{where } PQ \text{ is } P \mathbin{\hat{\|}} Q) \\
= & \exists d \exists e \bullet c = d \setminus (\alpha PQ \cap \alpha R) \;\; \wedge \;\; d \restriction R \in \underset{\alpha PQ \cap \alpha R}{\sharp} \hat{R} \\
& \qquad \wedge \;\; d \restriction PQ \;\; \underset{\alpha PQ \cap \alpha R}{\sharp}^{-1} e \setminus (\alpha P \cap \alpha Q) \\
& \qquad \wedge \;\; e \restriction P \in \underset{\alpha P \cap \alpha Q}{\sharp} \hat{P} \;\; \wedge \;\; e \restriction Q \in \underset{\alpha P \cap \alpha Q}{\sharp} \hat{Q} \\
= & \{\text{channels from } \alpha P \cap \alpha Q \text{ do not appear in } d\text{, which is a compu-} \\
& \text{tation of } PQ \text{ and } R \text{ running in parallel}\} \\
& \exists d \exists e \bullet c = d \setminus ((\alpha P \cup \alpha Q) \cap \alpha R) \;\; \wedge \;\; d \restriction R \in \underset{(\alpha P \cup \alpha Q) \cap \alpha R}{\sharp} \hat{R} \\
& \qquad \wedge \;\; e \setminus (\alpha P \cap \alpha Q) \underset{(\alpha P \cup \alpha Q) \cap \alpha R}{\sharp} d \restriction (P \times Q) \\
& \qquad \wedge \;\; e \restriction P \in \underset{\alpha P \cap \alpha Q}{\sharp} \hat{P} \;\; \wedge \;\; e \restriction Q \in \underset{\alpha P \cap \alpha Q}{\sharp} \hat{Q} \\
= & \{e \text{ is transformed such that } e \underset{(\alpha P \cup \alpha Q) \cap \alpha R}{\sharp} e'\text{; note that } \underset{S}{\sharp} ; \underset{T}{\sharp} = \underset{S \cup T}{\sharp}\text{. Al-} \\
& \text{though } [(\alpha P \cup \alpha Q) \cap \alpha R] \cup (\alpha P \cap \alpha Q) = (\alpha P \cap \alpha Q) \cup (\alpha Q \cap \alpha R) \cup \\
& (\alpha P \cap \alpha R)\text{, channels from } \alpha Q \cap \alpha R \text{ do not occur in } \hat{P} \text{ so this} \\
& \text{term may be dropped from } \sharp P\text{. A similar observation applies} \\
& \text{to } \sharp Q.\} \\
& \exists d \exists e \bullet c = d \setminus ((\alpha P \cup \alpha Q) \cap \alpha R) \;\; \wedge \;\; d \restriction R \in \underset{(\alpha P \cup \alpha Q) \cap \alpha R}{\sharp} \hat{R} \\
& \qquad \wedge \;\; e \setminus (\alpha P \cap \alpha Q) = d \restriction (P \times Q) \\
& \qquad \wedge \;\; e \restriction P \in \underset{\alpha P \cap (\alpha Q \cup \alpha R)}{\sharp} \hat{P} \;\; \wedge \;\; e \restriction Q \in \underset{\alpha Q \cap (\alpha P \cup \alpha R)}{\sharp} \hat{Q} \\
= & \{\text{expand the state space of } e \text{ such that } e' \restriction (P \times Q) = e\text{. Note} \\
& \text{that } e' \restriction P = e \restriction P \text{ and } e' \restriction Q = e \restriction Q.\} \\
& \exists d \exists e \bullet c = d \setminus ((\alpha P \cup \alpha Q) \cap \alpha R) \;\; \wedge \;\; d \restriction R \in \underset{(\alpha P \cup \alpha Q) \cap \alpha R}{\sharp} \hat{R} \\
& \qquad \wedge \;\; e \setminus (\alpha P \cap \alpha Q) = d
\end{array}
$$

$$\wedge \quad e \restriction P \in \sharp_{\alpha P \cap (\alpha Q \cup \alpha R)} \hat{P} \;\wedge\; e \restriction Q \in \sharp_{\alpha Q \cap (\alpha P \cup \alpha R)} \hat{Q}$$

$$= \quad \{\text{replace } d \text{ with } e \setminus (\alpha P \cap \alpha Q). \text{ Note that } \hat{R} \text{ does not contain channels from } \alpha P \cap \alpha Q.\}$$

$$\exists e \bullet c = e \setminus ((\alpha P \cap \alpha Q) \cup (\alpha Q \cap \alpha R) \cup (\alpha P \cap \alpha R))$$
$$\wedge \quad e \restriction R \in \sharp_{\alpha R \cap (\alpha P \cup \alpha Q)} \hat{R} \;\wedge\; e \restriction P \in \sharp_{\alpha P \cap (\alpha Q \cup \alpha R)} \hat{P}$$
$$\wedge \quad e \restriction Q \in \sharp_{\alpha Q \cap (\alpha R \cup \alpha P)} \hat{Q}.$$

For monotonicity, we have that if $Q \sqsubseteq^{\natural} R$ and c is a computation of $P \mathbin{\hat{\|}} R$ then:

$$\exists d \bullet c = d \setminus (\alpha P \cap \alpha R) \;\wedge\; d \restriction P \in \sharp\hat{P} \;\wedge\; d \restriction R \in \sharp\hat{R}$$

$$\Rightarrow \quad \exists d \bullet c = d \setminus (\alpha P \cap \alpha R) \;\wedge\; d \restriction P \in \sharp\hat{P} \;\wedge\; d \restriction R \in^{\natural} \sharp\hat{R}$$

$$\Rightarrow \quad \{d' \text{ is } d \text{ with extra repetitions}\}$$
$$\exists d' \bullet c =^{\natural} d' \setminus (\alpha P \cap \alpha R) \;\wedge\; d' \restriction P \in \sharp\hat{P} \;\wedge\; d' \restriction R \in \sharp\hat{R}$$

$$\Rightarrow \quad c \in^{\natural} \widehat{PQ}.$$

5.2 Syntactic Parallel

A syntactic parallel operator defines the result of placing the two action systems in parallel in terms of their syntax. It is therefore simple to write for an action system P action systems Q and R which give P when run in parallel, and vice versa. The syntactic parallel operator will be written $\|$.

If two instructions contain matching communications on one or more channels, then they may be placed in parallel with the result that the communications become assignments. If P, Q, S and T are instructions, and if S and T are free of communications on the shared channels, then:

$$\begin{array}{lcl} \textbf{skip} \,\| S & = & S \\ (c?x;\ P) \,\|\, (c!y;\ Q) & = & x := y;\ (P \,\|\, Q) \\ P \,\|\, (T;\ Q) & = & T;\ (P \,\|\, Q). \end{array}$$

The result of placing two instructions in parallel is **abort** if the communications are mismatched. Note that this parallel operator is not defined on all instructions; for example instructions of the form **do** ... **od** are not covered (although chapter 6 describes how an instruction of the form **if** ... **fi** can be dealt with).

Two actions $\overline{c} \wedge g \rightarrow S$ and $\overline{d} \wedge h \rightarrow T$ (where $\overline{c}$ and $\overline{d}$ are combinations of probes on the shared channels and g and h are guards, possibly including probes on non-shared channels) may be placed in parallel to give $g \wedge h \rightarrow (S \,\|\, T)$ whenever that result is defined.

If P and Q are the action systems

$$P = \textbf{var } v \bullet S_0;\ \textbf{do } ([]_{i \in I} A_i)\ \textbf{od}$$
$$Q = \textbf{var } w \bullet T_0;\ \textbf{do } ([]_{j \in J} B_j)\ \textbf{od},$$

and if P and Q have no shared variables but share a set c of channels, then the action system $P \parallel Q$ is defined if and only if the following is defined, and in that case $P \parallel Q$ equals it:

$$
\begin{array}{l}
\textbf{var } v \cup w \bullet \\
S_0 \wedge T_0; \\
\textbf{do} \\
\quad\ \ ([]\ i : I \mid \text{no channel from } c \text{ occurs in } A_i \bullet A_i) \\
\quad [] \ ([]\ j : J \mid \text{no channel from } c \text{ occurs in } B_j \bullet B_j) \\
\quad [] \ ([]\ i : I, j : J \mid A_i \text{ and } B_j \text{ share at least one channel} \bullet A_i \parallel B_j) \\
\textbf{od}.
\end{array}
$$

In this definition a communication action from P is selected only if a matching action from Q is also enabled—ignoring probes on the shared channels—and these actions are executed in synchrony. This means that Q has influenced the selection mechanism which executes the actions of P. The probes of channels shared by P and Q are lost: they were informational only, as mentioned in section 2.5. A communication action on a shared channel whose probe is false will not be selected for execution in any case.

Proposition: For any action systems P and Q, $P \mathbin{\hat{\parallel}} Q \sqsubseteq P \parallel Q$ and $P \parallel Q \sqsubseteq^E P \mathbin{\hat{\parallel}} Q$, where E contains as elements the state spaces of P and Q, providing $P \parallel Q$ is defined and does not contain **abort**.

Proof: $P \mathbin{\hat{\parallel}} Q \sqsubseteq P \parallel Q$:

Take a computation c of $P \parallel Q$. Let R be an action system which is similar to $P \parallel Q$ but which exposes the hidden communications; that is, each pair $ch?x \parallel ch!y$ which became an assignment in $P \parallel Q$ is accompanied in R by an output $ch!y$ and a 'magic' action which assigns $\overline{ch} = \mathit{true}$ in the state immediately preceding the output in each computation. There is a computation d of R such that $d \setminus cs = c$, where cs is the set of common channels. The initial state of d satisfies the initial condition of each of P and Q. Each transition in d is the result of an A_i, a B_j or an $A_i \parallel B_j$. When the transition is restricted to P it becomes either A_i, **skip** or A_i respectively. Each of these is a possible transition of $\underset{cs}{\sharp}\hat{P}$, and hence $d \restriction P \in \underset{cs}{\sharp}\hat{P}$. Similarly, $d \restriction Q \in \underset{cs}{\sharp}\hat{Q}$, so therefore c is a computation of $P \mathbin{\hat{\parallel}} Q$.

$P \parallel Q \sqsubseteq^E P \hat{\parallel} Q$:

Let c be a computation of $P \hat{\parallel} Q$. By definition, there exists d' such that $d \setminus cs = c$, $d' \upharpoonright P \in \underset{cs}{\sharp} \hat{P}$ and $d' \upharpoonright Q \in \underset{cs}{\sharp} \hat{Q}$. The proof of the proposition will be to find a computation e such that $(e \setminus cs) =^E (d \setminus cs)$ which can be a result of executing R. The initial state of d must satisy the initial condition of each of P and Q, so that the first element of d can be the first element of e. The other elements of e will be defined by applying transitions which are actions of R. Each transition is formed by taking one or more of the remaining transitions of d as follows. Take transitions of d up to the point where a state change occurs which is not restricted to cs. If no shared communication occurs in this list of transitions, then the list contains precisely one transition (plus possibly some transitions which have no effect)—which must be a transition from either P or Q or one transition from each system executed concurrently. If a shared communication did occur during the list of transitions, then the list of transitions is an $A_i \parallel B_j$ action, possibly followed by another action or actions. Because the system $P \parallel Q$ is defined, and therefore the communicating actions contain matched communications, it is possible to determine the point at which the $A_i \parallel B_j$ action ends: if the action causes a state change which is not restricted to cs then the action ends at the state change in the list of transitions, and otherwise the action ends at the last communication in the list of transitions which is also a communication of A_i and B_j. Now, given a list of transitions from d which is from A_i, B_j, $A_i \parallel B_j$, or an A_i and a B_j executing concurrently, the list is added to the computation e as a single transition, except in the latter case when the actions are added in sequence to e.

It is clear that $(e \setminus cs) =^E (d \setminus cs)$, since in transforming d to e states were only omitted when they were repetitions or intermediate states in a single transition, and states were only added when they were the intermediate states between actions of P and actions of Q. It is also clear that e is a computation of R, since it starts with an initial state of R and continues with transitions of R.

This result shows that semantic parallel and syntactic parallel are equivalent with respect to $=^E$ on action systems where both are properly defined, and hence it is possible to combine the ease of use of the syntactic parallel operator with the associative and monotonic properties of the semantic parallel operator, saying either

$$A = B \parallel C \sqsubseteq^E B \hat{\parallel} C$$

or $$B \hat{\parallel} C \sqsubseteq B \parallel C = A$$

as required. It is also possible to prove the corresponding result involving several action systems in parallel (either using a generalisation of the above proof or by proving that $\hat{\|}$ is monotonic with respect to a modified version of the $\sqsubseteq^E$ relation).

5.3 Example

In section 1.4.2, the following action system was developed to implement an unordered buffer:

$$\begin{array}{l}
\mathbf{var}\ t, u : \mathbf{bag}\ Msg \bullet t = u = \emptyset;\\
\mathbf{do}\ \overline{in} \rightarrow [\![\mathbf{var}\ m : Msg \bullet in?m;\ t := t \cup \{m\}]\!]\\
\ \ [\!]\ t \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in t;\ t := t - \{m\};\ u := u \cup \{m\}]\!]\\
\ \ [\!]\ \overline{out} \wedge u \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in u;\ u := u - \{m\};\ out!m]\!]\\
\mathbf{od}.
\end{array}$$

The middle action in this system represents the communication of elements from the sender, which owns the bag t, to the receiver, which owns the bag u. It is now possible to decompose this action system into two running in parallel and communicating via a channel. The first step is to introduce an assignment instruction which represents the communication of a value between the two processes. This will be inserted into the middle action of the above system, giving the following action which is equivalent.

$$t \neq \emptyset \rightarrow [\![\mathbf{var}\ m, n : Msg \bullet m :\in t;\ t := t - \{m\};\ n := m;\ u := u \cup \{n\}]\!]$$

The definition of syntactic parallel states that this action is equal to the combination of the following two running in parallel. The instructions on either side of the assignment instruction—including the creation and deletion of local variables—may be distributed between the two actions in any convenient way.

$$\begin{array}{l}
\overline{mid} \wedge t \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in t;\ t := t - \{m\};\ mid!m]\!]\\
\overline{mid} \rightarrow [\![\mathbf{var}\ n : Msg \bullet mid?n;\ u := u \cup \{n\}]\!]
\end{array}$$

(The probes have been inserted for consistency with the rest of the system, and by the definition of syntactic parallel they are ignored). It is now possible to divide the remaining actions of the system between the two processes. Since they do not share channels they may be placed unchanged into the new systems. It can now be verified that no variable is shared between the two systems, which are as follows.

$$\begin{array}{l}
\mathbf{var}\ t : \mathbf{bag}\ Msg \bullet t = \emptyset;\\
\mathbf{do}\ \overline{in} \rightarrow [\![\mathbf{var}\ m : Msg \bullet in?m;\ t := t \cup \{m\}]\!]\\
\ \ [\!]\ \overline{mid} \wedge t \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in t;\ t := t - \{m\};\ mid!m]\!]\\
\mathbf{od}
\end{array}$$

$$
\begin{array}{l}
\mathbf{var}\ u : \mathbf{bag}\, Msg \bullet u = \emptyset; \\
\mathbf{do}\ \overline{mid} \rightarrow [\![\mathbf{var}\ n : Msg \bullet mid?n;\ u := u \cup \{n\}]\!] \\
\ \ [\,]\ \overline{out} \wedge u \neq \emptyset \rightarrow [\![\mathbf{var}\ m : Msg \bullet m :\in u;\ u := u - \{m\};\ out!m]\!] \\
\mathbf{od}
\end{array}
$$

As promised, the system now consists of two unordered buffers running in parallel, each of which is similar to the original one (although the names of the variables and channels have changed).

6

A Routing Switch

This example demonstrates the use of the syntactic parallel operator to show that a chain of several processes is a correct solution to a problem. Two different inductive proofs are presented.

6.1 An Initial Implementation

A simple routing switch has one input channel (l) and k output channels (v_i for each i from 0 to $k-1$). Each input contains a number h between 0 and $k-1$ and a message m. After each input the switch will output message m on output v_h. The specification of the switch is therefore:

$$r = \mathrm{Inf}(\overline{in}) \wedge (\forall i : 0 \,..\, k \bullet \mathrm{Inf}(\overline{v_i}))$$

$$I = \forall i : 0 \,..\, k \bullet \langle v_i \rangle = \langle m \mid (i, m) \in \langle l \rangle \rangle$$

$$l = \forall N : \mathbb{N}\ \exists n : \mathbb{N} \bullet \#\langle comms \rangle_n > N.$$

An initial implementation of this specification is the following action system, which inputs each (number,message) pair and sends the message to the required output using an alternation instruction.

$$
\begin{array}{l}
\textbf{var } h : \mathbb{N}, m : Msg \bullet \\
\textbf{do } true \rightarrow l?(h, m); \\
\qquad \textbf{if } h = 0 \rightarrow v_0!m \\
\qquad \;[\!]\; h = 1 \rightarrow v_1!m \\
\qquad \;\vdots \\
\qquad \;[\!]\; h = k-1 \rightarrow v_{k-1}!m \\
\qquad \;[\!]\; h \geq k \rightarrow \textbf{skip} \\
\qquad \textbf{fi} \\
\textbf{od}
\end{array}
$$

This action system can easily be shown to preserve the required invariant, and it satisfies the liveness condition because its action always eventually terminates provided the rely condition is met. This action system discards any inputs containing numbers greater than or equal to k. According to the above description this will never occur, but this feature has been added for convenience in the proof below.

6.2 A Chain of Processes

An implementation of the switch will be written as a chain of processes. A 'chaining' operator will be defined to run the processes in parallel with the appropriate channels renamed and a *SINK* process will be defined to absorb unused output:

Definition: $P \gg Q \mathrel{\widehat{=}} P[mid/r] \parallel Q[mid/l]$
$\mathit{SINK} \mathrel{\widehat{=}} \mathbf{var}\ x \bullet \mathbf{do}\ \mathit{true} \to l?x\ \mathbf{od}.$

With these definitions, the switch may be written:

$$A_0 \gg A_1 \gg \cdots \gg A_{k-1} \gg \mathit{SINK}$$

where A_j is the action system:

$$\begin{array}{l}
\mathbf{var}\ h : \mathbb{N}, m : \mathit{Msg} \bullet \\
\mathbf{do}\ \mathit{true} \to l?(h, m);\ \mathbf{if}\ h = j \to v_j!m \\
\qquad\qquad\qquad\qquad\quad [\!]\ h \neq j \to r!(h, m) \\
\qquad\qquad\qquad\qquad\quad \mathbf{fi} \\
\mathbf{od}.
\end{array}$$

This represents a pipeline of processes, one connected to each output. Each (number,message) pair is passed along the pipeline until it reaches the correct process, and then the message is sent to the corresponding output. We may prove that this implementation is the same as that given earlier by writing the system as a single process as defined in section 5.2 and using induction. Section 5.2 does not define the results of placing in parallel systems containing **if** instructions, so this will be dealt with by transforming the action containing an **if** instruction into several that do not, essentially by extending the guards of the actions to cover the guards of the **if** instruction using an angelic 'assumption' construction.

Definition: For any predicate p, the assumption $\{p\}^\top$ is the instruction $[\mathit{true}, p]$, which is **magic** if p is false and **skip** otherwise.

Using this notation, the action system A_j can be written as follows.

$$\begin{array}{ll}
\mathbf{var}\ h : \mathbb{N}, m : \mathit{Msg} \bullet & \\
\mathbf{do}\ \mathit{true} \to l?(h, m);\ \{h = j\}^\top;\ v_j!m & \text{(A1)} \\
[\!]\ \mathit{true} \to l?(h, m);\ \{h \neq j\}^\top;\ r!(h, m) & \text{(A2)} \\
\mathbf{od} &
\end{array}$$

Because the assumptions cover all cases and the parts of the actions which lead up to the assumptions are the same, the system can always choose an execution sequence which does not require **magic**.

The hypothesis for induction will be that for any n less than k, the action system B_n defined as

$$
\begin{array}{l}
\textbf{var}\ h : \mathbb{N}, m : \mathit{Msg} \bullet \\
\textbf{do}\ \mathit{true} \rightarrow l?(h, m); \\
\quad\quad \textbf{if}\ h = n \rightarrow v_n!m \\
\quad\quad \;[]\ h = n + 1 \rightarrow v_{n+1}!m \\
\quad\quad \;\vdots \\
\quad\quad \;[]\ h = k - 1 \rightarrow v_{k-1}!m \\
\quad\quad \;[]\ h < n \vee h \geq k \rightarrow \textbf{skip} \\
\quad\quad \textbf{fi} \\
\textbf{od}
\end{array}
$$

is the same as $A_n \gg A_{n+1} \gg \cdots \gg A_{k-1} \gg \mathit{SINK}$. Then when $n = 0$ we have the result that the two switch implementations are equal. The base case is that in which $n = k - 1$. Then we have simply $A_{k-1} \gg \mathit{SINK}$, which is defined to be the action system:

$$
\begin{array}{ll}
\textbf{var}\ h : \mathbb{N}, m : \mathit{Msg}, x \bullet & \\
\textbf{do}\ \mathit{true} \rightarrow l?(h, m);\ \{h = k - 1\}^{\top};\ v_{k-1}!m & \{\text{A1 (no communications)}\} \\
[]\ \mathit{true} \rightarrow l?(h, m);\ \{h \neq k - 1\}^{\top};\ x := (h, m);\ \textbf{skip} & \{\text{from A2} \parallel \mathit{SINK}\} \\
\textbf{od}. &
\end{array}
$$

Since x is never used, it can be discarded. When this action system is rewritten in terms of the **if** instruction, the result is B_{k-1}.

The induction step is to show that $A_{n-1} \gg B_n = B_{n-1}$. Since the **if** instruction in B_n does not communicate on a shared channel, it need not be removed and therefore the system $A_{n-1} \gg B_n$ may be written as follows. Since the two systems contain similarly named variables, those of B_n will be renamed to h' and m'.

$$
\begin{array}{ll}
\textbf{var}\ h : \mathbb{N}, m : \mathit{Msg}, h' : \mathbb{N}, m' : \mathit{Msg} \bullet & \\
\textbf{do}\ \mathit{true} \rightarrow l?(h, m);\ \{h = n - 1\}^{\top};\ v_{n-1}!m & \{\text{A1 (no communications)}\} \\
[]\ \mathit{true} \rightarrow l?(h, m);\ \{h \neq n - 1\}^{\top};\ h', m' := h, m; & \{\text{from A2} \parallel B_n\} \\
\quad\quad \textbf{if}\ h' = n \rightarrow v_n!m' & \\
\quad\quad \;[]\ h' = n + 1 \rightarrow v_{n+1}!m' & \\
\quad\quad \;\vdots & \\
\quad\quad \;[]\ h' < n \vee h' \geq k \rightarrow \textbf{skip} & \\
\quad\quad \textbf{fi} & \\
\textbf{od} &
\end{array}
$$

Since the values of the variables h' and m' equal those of h and m throughout the **if** instruction and since none of these variables is changed, references to h' and m' may be replaced by references to h and m. After this, h' and m' become surplus and may be discarded. When the assumption instructions in the system are rewritten as an **if** instruction, this is the result.

$$
\begin{array}{l}
\mathbf{var}\ h : \mathbb{N}, m : \mathit{Msg} \bullet \\
\mathbf{do}\ \mathit{true} \to l?(h, m); \\
\quad \mathbf{if}\ h = n - 1 \to v_{n-1}!m \\
\quad\ \ [\!]\ h \neq n - 1 \to \\
\qquad \mathbf{if}\ h = n \to v_n!m \\
\qquad\ \ [\!]\ h = n + 1 \to v_{n+1}!m \\
\qquad\ \ \vdots \\
\qquad\ \ [\!]\ h < n \vee h \geq k \to \mathbf{skip} \\
\qquad \mathbf{fi} \\
\quad \mathbf{fi} \\
\mathbf{od}
\end{array}
$$

A law of software engineering (for example, a combination of laws 18.3 and 18.6 from [17]) states that the instruction

$$\mathbf{if}\ \left([\!]_i\ A_i\right)\ [\!]\ g \to S;\ \mathbf{if}\ \left([\!]_j\ g_j \to S_j\right)\ \mathbf{fi}\ \mathbf{fi}$$

is the same as the instruction

$$\mathbf{if}\ \left([\!]_i\ A_i\right)\ [\!]\ \left([\!]_j\ g \wedge g_j \to S;\ S_j\right)\ \mathbf{fi}$$

provided S does not change the value of any g_i (this may be proved by rewriting the instruction as a set of assumption actions, combining assumptions, and rewriting as an **if** instruction). Applying this rule to the above action system gives B_{n-1} as required.

The action system A_j may be refined to W_j, which is as follows. It inputs the number separately from the message, and therefore has already decided where to send the message when it starts to input the message.

$$
\begin{array}{ll}
\mathbf{var}\ h : \mathbb{N}, m : \mathit{Msg} \bullet & \\
\mathbf{do}\ \mathit{true} \to l?h;\ \mathbf{if}\ h = j \to l?m;\ v_j!m & \text{(W1)} \\
\qquad\qquad\qquad\ \ [\!]\ h \neq j \to l?m;\ r!h;\ r!m & \text{(W2)} \\
\qquad\qquad\qquad\ \ \mathbf{fi} & \\
\mathbf{od} &
\end{array}
$$

This involves changing the inputs of the system so that the number and the message are separate entities. It is clear by inspection of this system that the message is always input immediately after the number. The outputs on channel r are changed similarly. There is a one-to-one relation between computations of W_j and computations of A_j and therefore the data refinement is valid. This simple data refinement allows the system to be implemented more efficiently because in practice the sequential communications $l?m;\ v_j!m$ and $l?m;\ (r!h;\ r!m)$ may be implemented in such a way that they execute concurrently and so that the entire message m need not be held in memory at once. This system can thus be implemented on k processors in such a way that the delay at each processor for any one message equals the time taken to receive and compare the destination number. This algorithm is commonly called the 'wormhole' algorithm [7].

6.3 A Faster Algorithm

M_j will be defined to be the action system:

$$
\begin{array}{ll}
\textbf{var } h : \mathbb{N}, m : Msg \bullet & \\
\textbf{do } true \rightarrow l?h;\ r!h;\ \textbf{if } h = j \rightarrow l?m;\ v_j!m & \text{(M1)} \\
\qquad\qquad\qquad\qquad\quad [\!]\ h > j \rightarrow l?m;\ r!m & \text{(M2)} \\
\qquad\qquad\qquad\qquad\quad [\!]\ h < j \rightarrow \textbf{skip} & \text{(M3)} \\
\qquad\qquad\qquad\qquad\quad \textbf{fi} & \\
\textbf{od}. &
\end{array}
$$

This system is similar to the W_j system except that it always outputs the number for each message to the next process in the chain. If the number indicates that the message has not yet arrived at its destination, then the message is sent after the number. Otherwise the number will be ignored by the other processors as it travels along the chain. A chain of processes M_j may be shown to implement a switch in an inductive manner as above; however, an alternative proof will be given here. The proof involves showing that a chain of processes M_j refines a chain of processes W_j. Since the communications of W_j and M_j are different, it is difficult to show that refinement holds between the two individual action systems. Instead, an intermediate system Z_j is introduced whose communications on the left resemble those of M_j and whose communications on the right resemble those of W_j. This system is:

$$
\begin{array}{ll}
\textbf{var } h : \mathbb{N}, m : Msg \bullet & \\
\textbf{do } true \rightarrow l?h;\ \textbf{if } h = j \rightarrow l?m;\ v_j!m & \text{(Z1)} \\
\qquad\qquad\qquad\quad [\!]\ h > j \rightarrow r!h;\ l?m;\ r!m & \text{(Z2)} \\
\qquad\qquad\qquad\quad [\!]\ h < j \rightarrow \textbf{skip} & \text{(Z3)} \\
\qquad\qquad\qquad\quad \textbf{fi} & \\
\textbf{od}. &
\end{array}
$$

The proof relies on the following four properties of the three action systems, which will be shown later.

(a) $Z_0 = W_0$
(b) $Z_j \gg W_{j+1} = Z_j \gg Z_{j+1}$
(c) $Z_{k-1} \gg \mathit{SINK} = M_{k-1} \gg \mathit{SINK}$
(d) $M_j \gg Z_{j+1} = Z_j \gg Z_{j+1}$

From these properties, we have:

$$\begin{aligned}
& W_0 \gg W_1 \gg \cdots \gg W_{k-1} \gg \mathit{SINK} && \\
= \; & Z_0 \gg W_1 \gg \cdots \gg W_{k-1} \gg \mathit{SINK} && \{\text{(a)}\} \\
= \; & Z_0 \gg Z_1 \gg \cdots \gg W_{k-1} \gg \mathit{SINK} && \{\text{(b)}\} \\
= \; & Z_0 \gg Z_1 \gg \cdots \gg Z_{k-1} \gg \mathit{SINK} && \{\text{(b) (repeatedly)}\} \\
= \; & M_0 \gg Z_1 \gg \cdots \gg Z_{k-1} \gg \mathit{SINK} && \{\text{(d)}\} \\
= \; & M_0 \gg \cdots \gg M_{k-2} \gg Z_{k-1} \gg \mathit{SINK} && \{\text{(d) (repeatedly)}\} \\
= \; & M_0 \gg \cdots \gg M_{k-2} \gg M_{k-1} \gg \mathit{SINK} && \{\text{(c)}\}.
\end{aligned}$$

The M_j action system is faster than the W_j system to implement, because each input is matched by an output of the same data and these inputs and outputs may all be implemented concurrently. This applies even to the destination number. If this number is transmitted as a sequence of bits (or other units of communication) then the delay at each processor for one message is reduced to the time taken to receive one bit. This algorithm has been named the ‘mad postman’ algorithm [12], because each processor starts transmitting the message even before the destination is fully known. This means that stray destination numbers travel through the switch because they have been incorrectly directed. These stray destination numbers are just ignored.

In practice, several switches of the kind described here will be piped together to give a 2- or 3-dimensional switch, and the ‘mad postman’ algorithm will be combined with the ‘wormhole’ algorithm to make an efficient routing system. When load is light, messages will not have to wait at any processor, and the ‘mad postman’ algorithm may be employed in order to deliver the messages as rapidly as possible. When load increases up to the point when a message has to wait at a processor for a time which is long enough for the whole of a destination number to be received, the processor may decide whether or not to relay this number depending on its value. In this way, the number of stray destination numbers within the network is cut down when there are many messages in the system.

6.3.1 Proof of property (a)

Provided the histories of the channels l and r are considered separately, $Z_0 = W_0$ by inspection since $h < 0$ is always false. Since chaining the action systems together hides l and r individually, considering the histories separately makes no difference to the working of the whole system.

6.3.2 Proof of property (c)

Since chaining a process to *SINK* has the sole effect of hiding all communications done on channel r, $Z_{k-1} \gg SINK = M_{k-1} \gg SINK$ by inspection.

6.3.3 Proof of property (b)

The system $Z_j \gg W_{j+1}$ may be written as follows, after rewriting each system in terms of assumptions.

$$
\begin{array}{ll}
\textbf{var } h : \mathbb{N}, m : Msg, h' : \mathbb{N}, m' : Msg \bullet & \\
\textbf{do } true \rightarrow l?h;\ \{h = j\}^{\top};\ l?m;\ v_j!m & \{\text{Z1}\} \\
[\!]\ true \rightarrow l?h;\ \{h < j\}^{\top};\ \textbf{skip} & \{\text{Z3}\} \\
[\!]\ true \rightarrow l?h;\ \{h > j\}^{\top};\ h' := h;\ \{h' = j+1\}^{\top}; & \{\text{Z2} \parallel \text{W1}\} \\
\qquad l?m;\ m' := m;\ v_{j+1}!m' & \\
[\!]\ true \rightarrow l?h;\ \{h > j\}^{\top};\ h' := h;\ \{h' \neq j+1\}^{\top}; & \{\text{Z2} \parallel \text{W2}\} \\
\qquad l?m;\ m' := m;\ r!h';\ r!m' & \\
\textbf{od} &
\end{array}
$$

In the last action of this system, the assumption $h' \neq j+1$ is the same as $h' > j+1$, since it has already been assumed that $h > j$ and we have that $h = h'$. A further action may be added to the system, namely:

$$true \rightarrow l?h;\ \{h > j\}^{\top};\ h' := h;\ \{h' < j+1\}^{\top};\ \textbf{abort}$$

because, since the two assumptions contradict each other, the action will never be selected for execution. The action system which results is the same as that which results from writing $Z_j \gg Z_{j+1}$.

6.3.4 Proof of property (d)

The system $M_j \gg Z_{j+1}$ may be written as follows, after rewriting each system in terms of assumptions.

$$
\begin{array}{ll}
\textbf{var } h : \mathbb{N}, m : Msg, h' : \mathbb{N}, m' : Msg \bullet & \\
\textbf{do } true \rightarrow l?h;\ h' := h;\ \{h = j\}^{\top};\ \{h' = j+1\}^{\top};\ \textbf{abort} & \{\text{M1} \parallel \text{Z1}\} \\
[\!]\ true \rightarrow l?h;\ h' := h;\ \{h = j\}^{\top};\ \{h' > j+1\}^{\top};\ \textbf{abort} & \{\text{M1} \parallel \text{Z2}\} \\
[\!]\ true \rightarrow l?h;\ h' := h;\ \{h = j\}^{\top};\ \{h' < j+1\}^{\top};\ l?m;\ v_j!m & \{\text{M1} \parallel \text{Z3}\} \\
[\!]\ true \rightarrow l?h;\ h' := h;\ \{h > j\}^{\top};\ \{h' = j+1\}^{\top}; & \{\text{M2} \parallel \text{Z1}\} \\
\qquad l?m;\ m' := m;\ v_{j+1}!m' & \\
[\!]\ true \rightarrow l?h;\ h' := h;\ \{h > j\}^{\top};\ \{h' > j+1\}^{\top}; & \{\text{M2} \parallel \text{Z2}\} \\
\qquad l?m;\ r!h;\ m' := m;\ r!m' & \\
[\!]\ true \rightarrow l?h;\ h' := h;\ \{h > j\}^{\top};\ \{h' < j+1\}^{\top};\ \textbf{abort} & \{\text{M2} \parallel \text{Z3}\}
\end{array}
$$

$$[]\ true \to l?h;\ h' := h;\ \{h < j\}^\top;\ \{h' = j+1\}^\top;\ \mathbf{abort} \qquad \{\mathrm{M3} \parallel \mathrm{Z1}\}$$
$$[]\ true \to l?h;\ h' := h;\ \{h < j\}^\top;\ \{h' > j+1\}^\top;\ \mathbf{abort} \qquad \{\mathrm{M3} \parallel \mathrm{Z2}\}$$
$$[]\ true \to l?h;\ h' := h;\ \{h < j\}^\top;\ \{h' < j+1\}^\top;\ \mathbf{skip} \qquad \{\mathrm{M3} \parallel \mathrm{Z3}\}$$
od

After deleting most of the actions which will never be selected and combining some of the assumptions, the system simplifies to:

$$\mathbf{var}\ h : \mathbb{N}, m : Msg, h' : \mathbb{N}, m' : Msg \bullet$$
$$\mathbf{do}\ true \to l?h;\ h' := h;\ \{h = j\}^\top;\ l?m;\ v_j!m$$
$$[]\ true \to l?h;\ h' := h;\ \{h > j\}^\top;\ \{h' = j+1\}^\top;\ l?m;\ m' := m;\ v_{j+1}!m'$$
$$[]\ true \to l?h;\ h' := h;\ \{h > j\}^\top;\ \{h' > j+1\}^\top;\ l?m;\ r!h;\ m' := m;\ r!m'$$
$$[]\ true \to l?h;\ h' := h;\ \{h > j\}^\top;\ \{h' < j+1\}^\top;\ \mathbf{abort}$$
$$[]\ true \to l?h;\ h' := h;\ \{h < j\}^\top;\ \mathbf{skip}$$
od

which is the same as the action system $Z_j \gg Z_{j+1}$ because the variable h' may be removed from some of the actions since none of the actions depends on the initial value of h'.

This concludes the proof that the 'mad postman' algorithm represents a correct implementation of a switch.

7 Simulation

Simulation is a technique for data refinement in which each action of the concrete action system models an action of the abstract action system. Since the correspondence is between actions rather than between computations, developing a simulation is easier than performing a more general data refinement. Two kinds of simulation are well known in relation-based systems [10] and are described here: downward simulation and upward simulation.

7.1 The Relational View of an Action

If a is the action $g \rightarrow S$, then a may be viewed as a relation which relates any two states x and y if and only if $g(x)$ and execution of S starting in state x could result in state y (a similar interpretation appears in [1]). The action is treated as being **magic** if it is executed in a state which does not satisfy the guard: a does not relate x to any state if $g(x)$ is not true. When an action system is viewed as a relation, it can be placed in sequence with other relations giving the usual result. A sequence of actions $a_0;\ a_1;\ \ldots;\ a_n$ is a possible trace of an action system starting from a state s if and only if the relation $a_0;\ a_1;\ \ldots;\ a_n$ relates s to some state. If this is not the case then either one of the actions has a false guard or one of the actions is **magic**.

A state called $\perp$ will be added to the state space to represent non-termination. If the action A is enabled in state x but does not terminate when executed in that state, then the relation A will relate x to $\perp$. This allows non-termination to be dealt with implicitly, for if $C \subseteq A$ and A always terminates when started in state x, then C must also always terminate when started in state x because x must not be related to $\perp$ by C. If an action A relates a state x to $\perp$ then it will usually also relate x to every other state. This allows C to exhibit any behaviour when it is executed in state x.

The element $\perp$ is present in all state spaces, and any data refinement must relate the $\perp$ of one state space to the $\perp$ of the other, and to no other element.

7.2 Downward Simulation

Let A and C be the following action systems:

$$A = \mathbf{var}\ v : T \bullet init(v);\ \mathbf{do}\ ([]_{i \in I} A_i)\ \mathbf{od}$$
$$C = \mathbf{var}\ v' : T' \bullet init'(v');\ \mathbf{do}\ ([]_{i \in I} C_i)\ \mathbf{od}.$$

Let gg_A be the set of states which satisfy at least one of the guards of A, and similarly for gg_C. Suppose that the following conditions hold for some relation R between states of A and states of C:

(a) $\forall y \bullet init'(y) \to (\exists x \bullet x\ R\ y \wedge init(x))$;
(b) $R;\ C_i \subseteq A_i;\ R$ for each $i \in I$;
(c) $R(gg_A) \subseteq gg_C$, and
(d) $R;\ R^{-1} \subseteq \sim$ (where $\sim$ is an equivalence relation on computations as described in section 3.4).

Then C is a *downward simulation* of A, and $A \sqsubseteq_R C$.

The proof that refinement holds is as follows. Suppose c is a computation of C. We have that c_0, the first element of c, satisfies $init'$ and therefore by (a) there exists a_0 such that $a_0\ R\ c_0 \wedge init(a_0)$. This a_0 is an initial state of A and will form the first state in a computation a of A which is a member of $R^{-1}(c)$. For $i > 0$, if a_i, the ith element of a, has been chosen such that $a_i\ R\ c_i$ and c_i is not the last element of c, then $a_i\ R;\ C_n\ c_{i+1}$ for some action C_n of C. Therefore, by (b), $a_i\ A_n;\ R\ c_{i+1}$ and so there exists a_{i+1} such that $a_i\ A_n\ a_{i+1}$ and $a_{i+1}\ R\ c_{i+1}$. This a_{i+1} is a suitable next element of the computation a which preserves the property $a \in R^{-1}(c)$. On the other hand, if c_i is the last element of c, then either there is an action C_n which does not terminate in state c, or c_i is not a member of gg_C. In the former case, A_n must relate a_i to $\perp$ by an argument similar to the above, and therefore a_i is a valid last element of the computation a. In the latter case, by (c), c_i is not a member of $R(gg_A)$, and since $a_i\ R\ c_i$, this means that a_i is not a member of gg_A. Hence a_i is a termination state of A and is a valid last element of the computation a.

Hence there exists a which is a member of both $\hat{A}$ and $R^{-1}(c)$. By (d), if x is a member of $R^{-1}(c)$ then $x \sim a$, and so $R^{-1}(c) \subseteq \mathcal{E}(a)$ as required.

Note that condition (b) can be rewritten as follows. If $A_i = g \to S$ and $C_i = h \to T$ then $R;\ C_i \subseteq A_i;\ R$ if and only if $R^{-1}(h) \cap \mathrm{dom}(R;\ T) \subseteq g$ and $R;\ \{h\}^{\top};\ T \subseteq S;\ R$ (writing g and h as sets of states satisfying the respective predicates and S and T as relations which relate states before the respective commands to states afterwards).

If a liveness condition L is given to the abstract action system, then a downward simulation may still be carried out. If L constrains the possible order of execution of the actions, then the same liveness condition should be applied to the concrete action system. Since the order of execution of actions is identical between the two systems, the simulation will be valid. If L is a predicate on computations, then a liveness condition L' should be applied to the concrete system, where $R^{-1}(L') \subseteq L$, so that if c is a computation of the concrete action system then a must satisfy L whenever c is a simulation of a, since a is a member of $R^{-1}(c)$.

7.3 Simulation of null actions

If the abstract or concrete actions systems contain **null** actions, then an extra rule is required to ensure liveness. In addition to the above conditions (in which the sets gg_A and gg_C do not include the *true* guards of **null** actions), the following condition is required:

(e) If $\{C_j \mid j \in J\}$ is the set of concrete actions which simulate **null** actions from the abstract system, then for any point in the execution of the concrete action system, there must be a point in the future at which each C_j for j in J is disabled.

A special case of this condition which often occurs is one in which each C_j disables itself after finitely many executions. In other words, there is some function V from concrete states to members of some well-founded set W such that in any state each enabled C_j strictly decreases the value of V when it is executed. This gives a version of downward simulation which is closely related to the 'hiding refinement rule' of Butler [5].

The extension to the proof of the previous section to show that condition (e) is sufficient consists of showing that for any computation c, the computation a which the proof produces is a valid computation of A because it does not contain a sequence of infinitely many **null** actions executed consecutively. This is implied by the fact that (e) prevents this from happening in the concrete system because all the actions corresponding to **null** actions will always eventually be disabled.

Any C_j action of the concrete system which is not equal to **null** may be treated as an ordinary action in further refinements, since its property of not being executed infinitely many times in succession has now been encoded in the action system and the action no longer requires special treatment.

7.4 Example

In section 1.4.2, it was stated that the action system

$$
\begin{array}{l}
\textbf{var } s : \textbf{bag } Msg \bullet s = \emptyset; \\
\textbf{do } \overline{in} \rightarrow [\![\textbf{var } m : Msg \bullet in?m;\ s := s \cup \{m\}]\!] \\
[\!]\ \overline{out} \wedge s \neq \emptyset \rightarrow [\![\textbf{var } m : Msg \bullet m :\in s;\ s := s - \{m\};\ out!m]\!] \\
\textbf{od}
\end{array}
$$

may be simulated by this action system:

$$
\begin{array}{l}
\textbf{var } t, u : \textbf{bag } Msg \bullet t = u = \emptyset;\\
\textbf{do } \overline{in} \to [\![\textbf{var } m : Msg \bullet in?m;\ t := t \cup \{m\}]\!]\\
[\!]\ t \neq \emptyset \to [\![\textbf{var } m : Msg \bullet m :\in t;\ t := t - \{m\};\ u := u \cup \{m\}]\!]\\
[\!]\ \overline{out} \wedge u \neq \emptyset \to [\![\textbf{var } m : Msg \bullet m :\in u;\ u := u - \{m\};\ out!m]\!]\\
\textbf{od}.
\end{array}
$$

The data refinement relation R relates an abstract state to a concrete state if $t \cup u = s$ (where s belongs to the abstract state and t and u belong to the concrete state) and if all the other components of the states are identical. The equivalence relation $\sim$ in this simulation will be the identity. It is now possible to show that these systems satisfy all the conditions for a simulation.

(a) If $t = u = \emptyset$ then $s = \emptyset$ is a related state which satisfies the abstract initial condition.

(b) For each pair of corresponding actions, $g \to S$ and $h \to T$, it is required that $R^{-1}(h) \cap \mathrm{dom}(R;\ T) \subseteq g$ and $R;\ \{h\}^{\top};\ T \subseteq S;\ R$.

  (1) The concrete guard $\overline{in}$ of the first action is mapped by R^{-1} to the abstract guard $\overline{in}$. The relation $R;\ T$ corresponds to partitioning s into t and u followed by adding an input element to t. This is a special case of adding an input element to s and then partitioning it into t and u, which is $S;\ R$.

  (2) The second action in the concrete system corresponds to a **null** action in the abstract system. Since the abstract guard is *true*, it always contains $R^{-1}(h)$. The relation $R;\ \{h\}^{\top};\ T$ corresponds to partitioning s into t and u with t nonempty followed by moving an element from t to u. This is a special case of partitioning s into t and u, which is **skip**; R.

  (3) The concrete guard $\overline{out} \wedge u \neq \emptyset$ of the remaining action is mapped by R^{-1} to the abstract guard $\overline{out} \wedge s \neq \emptyset$. The relation $R;\ \{h\}^{\top};\ T$ corresponds to partitioning s into t and u with u nonempty followed by outputting an element from u. This is a special case of outputting an element from s and then partitioning it into t and u, which is $S;\ R$, as long as s is nonempty. However, s must be nonempty in order to give u nonempty in the partition.

(c)
$$
\begin{array}{ll}
 & R(gg_A)\\
= & R(\overline{in} \vee (\overline{out} \wedge s \neq \emptyset))\\
= & \overline{in} \vee (\overline{out} \wedge t \cup u \neq \emptyset)\\
\Rightarrow & \overline{in} \vee t \neq \emptyset \vee (\overline{out} \wedge u \neq \emptyset)\\
= & gg_C
\end{array}
$$

(d) $R;\ R^{-1}$ corresponds to partitioning s into t and u and then joining them back into s, which leaves the state unchanged.

(e) The variant V for the middle action of the concrete system is the number of elements in t. It strictly decreases whenever the action is executed and is bounded below by zero.

7.5 Upward Simulation

If A and C are the action systems:

$$A = \textbf{var } v : T \bullet init(v);\ \textbf{do } ([]_{i \in I} A_i) \textbf{ od}$$
$$C = \textbf{var } v' : T' \bullet init'(v');\ \textbf{do } ([]_{i \in I} C_i) \textbf{ od},$$

then C is an *upward simulation* of A if the following conditions hold:

(a) $\forall x\ \forall y \bullet init'(y) \wedge x\ R\ y \rightarrow init(x)$;
(b) $C_i;\ R^{-1} \subseteq R^{-1};\ A_i$ for each $i \in I$;
(c) $\forall x \bullet R^{-1}(x) \subseteq gg_A \rightarrow x \in gg_C$;
(d) every reachable state of C is in the image of R;
(e) A has finite nondeterminism or R^{-1} is image-finite (or both), and
(f) $R;\ R^{-1} \subseteq \sim$.

A has *finite nondeterminism* if and only if the sets *init* and $A_i(x)$ for each state x and each action A_i are finite. R^{-1} is *image-finite* if and only if the set $R^{-1}(x)$ is finite for each state x.

If C is an upward simulation of A then $A \sqsubseteq_R C$. The proof of this is as follows. Suppose c is a finite computation or a finite prefix of a computation of C. If c is a finite computation which ended because of a non-terminating action, then let c_n, the last element of c, be $\perp$ and let a_n also be $\perp$. Otherwise, by (d), c_n has a non-empty image under R^{-1}; let a_n be an element from it. For $i < n$, we have that $c_i\ C_m\ c_{i+1}$ for some m and $a_{i+1}\ R\ c_{i+1}$ so by (b) there exists a_i such that $a_i\ R\ c_i$ and $a_i\ A_m\ a_{i+1}$. When a_0 has been found, we have that a_0 satisfies *init* by (a) because c_0 satisfies *init'*. Hence we have a sequence a which is a member of $R^{-1}(c)$ and is a computation or a prefix of a computation of A. If c is a finite computation which ended because of a non-terminating action, then so is a because a ends with $\perp$. If c is a finite computation which ends with a termination state, then c_n is not a member of gg_C. By (c), $R^{-1}(x)$ contains an element which is not a member of gg_A. It is possible to choose a_n to be this element, in which case the resulting computation of a is a complete computation of A.

If c is an infinite computation of C, then let $K_0 = \mathbb{N}$ and let $\{c^k \mid k \in K_0\}$ be a set of finite prefixes of c such that $\#c_m < \#c_n$ whenever $m < n$. Let a^k be an element of $R^{-1}(c^k)$ obtained as above for each k. The set of possible values for the nth element of a^k for any n and k is the intersection between $R^{-1}(c^k_n)$ and the set of states which are reachable by executing the first n of the actions A_m from above. Since the actions A_m are fixed by the computation c, and because of (e), this intersection must be finite. That is, for any n and any r for which K_r has been defined, the sequence of values taken by a_n^k as k takes on all values from K_r is finite. Providing K_r is infinite, this sequence must therefore have a limit point l_n, and K_{r+1}, the set of all k from K_r such that $a_n^k = l_n$, is infinite.

Thus, letting n and r increase from zero upwards, it is possible to find a sequence of states $\langle l_n \mid n \in \mathbb{N} \rangle$ which is a limit of a^k as k tends to infinity. This sequence is an infinite computation of A which is an element of $R^{-1}(c)$.

In all cases, a computation a has been found such that a is a computation of A and is a member of $R^{-1}(c)$. By (f), therefore, $R^{-1}(c) \subseteq \mathcal{E}(a)$ as required.

The same observations regarding extra liveness conditions and **null** actions as for downward simulation also apply to upward simulation.

8 The Sliding-Window Protocol

8.1 Outline

The sliding-window protocol is an algorithm which implements an N-place buffer and at the same time provides reliable one-way communication over a two-way faulty channel. A faulty channel is one which may lose or duplicate messages. It must not deliver messages out of order, and it may lose only a finite number of successive messages or deliver a message a finite number of times in succession. We do not consider channels which may deliver corrupt messages because, by adding a checksum to each message and discarding all messages received with incorrect checksums, such a channel may be transformed into a channel which either loses a message or delivers it correctly with high probability. An output of a value v on a faulty channel c may be simulated with the instruction:

$$[\![\mathbf{var}\ n : \mathbb{N} \bullet \mathbf{do}\ n > 0 \rightarrow c!v;\ n := n - 1\ \mathbf{od}]\!]$$

provided that n is initially zero for at most a finite number of times in succession when this instruction is repeated. The process which is receiving messages from channel c will see n copies of the message (where n is an arbitrary natural number), so that the message will appear to have been lost or duplicated unless $n = 1$.

In a version of the sliding-window protocol, the sender and receiver each have a window of size N containing messages which have been received but not transmitted or output. The sender may input messages from the environment and transmit messages along the faulty channel. Each message sent along the faulty channel is tagged by a natural number between 0 and $2N - 1$, the tags being allocated in sequence as the messages are received from the environment. The receiver may receive messages from the faulty channel and output messages to the environment. The receiver may also transmit back to the sender an acknowledgement consisting of the set of natural-number tags within the receiver's window for which a message has not yet been received. Thus the sender will repeatedly resend messages until an acknowledgement indicates that they have been received, and the receiver will ignore duplicate messages which have tags identical to those of recently received messages.

The sliding-window protocol will be developed from a specification using the laws of refinement, and in particular using downward simulations. The faulty channel will initially be simulated with assignments, and will later be introduced as if it were a perfect channel by using the syntactic parallel operator. Finally the algorithm will be shown to work even when a faulty channel is used.

8.2 A Circular Buffer

A specification for the N-place buffer is as follows.

$$\begin{aligned} r &= \mathrm{Inf}\,\overline{in} \wedge \mathrm{Inf}\,\overline{out} \\ I &= \langle out\rangle \leq_N \langle in\rangle \\ l &= \forall N : \mathbb{N}\ \exists n : \mathbb{N} \bullet \#\langle out\rangle_n > N \end{aligned}$$

where for any two sequences x and y, $x \leq_N y$ if and only if x is a prefix of y and $\#y - \#x \leq N$.

None of the data refinements in this chapter will change the input and output data, so the equivalence relation $\sim$ required for data refinement in each case is:

$$x \sim y \Leftrightarrow x \restriction \{\langle out\rangle, \langle in\rangle\} =^{\natural} y \restriction \{\langle out\rangle, \langle in\rangle\}.$$

This relation preserves each of the above invariant and liveness conditions. If a relation R does not change the input and output data, then $R;\ R^{-1} \subseteq \sim$, as required by section 7.2, condition (d), and downward simulations will be carried out without further reference to this condition.

An action system which refines the above specification is:

$$\begin{array}{l} \mathbf{var}\ \ b : \mathbf{seq}\, Msg \bullet b = \langle\rangle; \\ \mathbf{do}\ \overline{in} \wedge \#b < N \rightarrow [\![\mathbf{var}\ \ x : Msg \bullet in?x;\ \ b := b \mathbin{+\!\!+} \langle x\rangle]\!] \\ \quad [\!]\ \overline{out} \wedge b \neq \langle\rangle \rightarrow out!b_0;\ \ b := b' \\ \mathbf{od}. \end{array}$$

It is clear that this system maintains the invariant

$$\langle in\rangle = \langle out\rangle \mathbin{+\!\!+} b \ \ \wedge\ \ \#b \leq N$$

and hence satisfies the invariant of the specification. Also, given that $\overline{in}$ and $\overline{out}$ are each always eventually true, one of the actions will always eventually be executed, so that $\langle in\rangle$ or $\langle out\rangle$ grows indefinitely. Since $\#\langle out\rangle$ is always at most N away from $\#\langle in\rangle$, this implies that the liveness condition is satisfied.

A simple refinement of this system uses a circular buffer of size $2N$. In fact it is possible to use a buffer of size N, but an informal reason for the choice of $2N$ is that the combined sizes of the sender's and receiver's windows in the sliding-window protocol can be as large as $2N$ and therefore at least this many distinct natural-number tags are required. The tags will come from the positions of the messages within the $2N$-element buffer.

The refinement takes place using a one-to-one relation which relates two states when:

$$j \ominus i = \#b \ \wedge \ (\forall k : 0 \mathinner{..} \#b \bullet b[k] = m[i \oplus k])$$

where $\oplus$ is addition modulo $2N$ and $\ominus$ is subtraction (giving a non-negative result) modulo $2N$. The resulting system is:

$$\begin{array}{l}
\mathbf{var}\ m : \mathbf{seq}_{2\mathrm{N}}\, Msg;\ i, j : 0 \mathinner{..} 2N \bullet i = j = 0; \\
\mathbf{do}\ \overline{in} \wedge j - i \neq \pm N \rightarrow in?m[j];\ j := j \oplus 1 \\
\ [\!]\ \overline{out} \wedge j - i \neq 0 \rightarrow out!m[i];\ i := i \oplus 1 \\
\mathbf{od}.
\end{array}$$

8.3 Developing the Protocol

The first step in developing the sliding-window protocol from a circular buffer is to introduce an action which will eventually correspond to the faulty channel. The channel will be used to transmit m from the sender to the receiver. A variable n will be introduced which contains the messages from m which have been transmitted so far. The refinement relation will relate two states if the values of the common variables are equal and also:

$$\forall k : 0 \mathinner{..} 2N \bullet n[k] \neq \perp \rightarrow k \in i \circ\circ j \wedge n[k] = m[k]$$

where $\perp$ is an element which cannot be a message, indicating an empty slot in n, and $i \circ\circ j$ is:

$$\begin{array}{l}
\{k : \mathbb{N} \mid i \leq k < j\}, \text{ if } i \leq j; \\
\{k : \mathbb{N} \mid i \leq k < 2N \vee 0 \leq k < j\}, \text{ otherwise.}
\end{array}$$

The following action system is a downward simulation of the circular buffer after adding one **null** action (section 2.7).

$$\begin{array}{l}
\mathbf{var}\ m, n : \mathbf{seq}_{2\mathrm{N}}\, Msg;\ i, j : 0 \mathinner{..} 2N \bullet i = j = 0 \wedge (\forall k : 0 \mathinner{..} 2N \bullet n[k] = \perp); \\
\mathbf{do}\ \overline{in} \wedge j - i \neq \pm N \rightarrow in?m[j];\ j := j \oplus 1 \\
\ [\!]\ \mathrm{dom}\, n \neq i \circ\circ j \rightarrow [\![\mathbf{var}\ k : \mathbb{N} \bullet k :\in i \circ\circ j \setminus \mathrm{dom}\, n;\ n[k] := m[k]\,]\!] \\
\ [\!]\ \overline{out} \wedge j - i \neq 0 \rightarrow out!m[i];\ n[i] := \perp;\ i := i \oplus 1 \\
\mathbf{od} \\
\text{where } \mathrm{dom}\, n = \{k : 0 \mathinner{..} 2N \mid n[k] \neq \perp\}.
\end{array}$$

The second action refines the **null** action and clearly disables itself after a finite number of repeated executions (section 7.3).

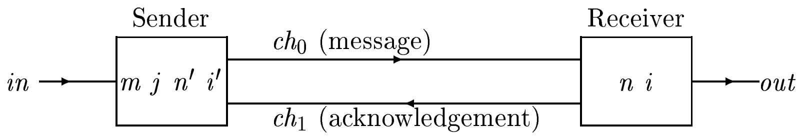
It is possible to remove m and j from the third action by strengthening the guard. If $n[i] \neq \perp$ then the condition on n used in the previous refinement implies that $j - i \neq 0$ and $n[i] = m[j]$, so the action may be replaced by:

$$\overline{out} \wedge n[i] \neq \perp \rightarrow out!n[i];\ n[i] := \perp;\ i := i \oplus 1.$$

The rule of downward simulation implies that strengthening a guard preserves correctness as long as the disjunction of the guards remains the same. In this case, we have that

$$(\overline{out} \wedge j - i \neq 0) \vee \operatorname{dom} n \neq i \circ\circ j \Leftrightarrow (\overline{out} \wedge n[i] \neq \perp) \vee \operatorname{dom} n \neq i \circ\circ j.$$

The aim of development is to partition the system into a sender and receiver which do not share variables except by means of the channels. The receiving action is already free of m and j, but the sending action and the sender's half of the channel action (the second action of the system) contain n and i, which will eventually belong to the receiver. It will be necessary to add variables n' and i' which approximate n and i from the sender's point of view. The distribution of variables is shown in the following diagram.



The new variables n' and i' will satisfy:

$$(i \circ\circ i \oplus N) \setminus \operatorname{dom} n \subseteq n' \wedge i \in i' \circ\circ j \wedge j \ominus i' \leq N.$$

The set $A = (i \circ\circ i \oplus N) \setminus \operatorname{dom} n$ represents the set of unfilled slots in the receiver's window. Given A and a value of i' satisfying the above, it is possible to calculate the value of i, provided $i \notin \operatorname{dom} n$, thus:

$$i = \begin{cases} \min A & \text{if } A \cap (i' \mathbin{..} 2N) = \emptyset \\ \min(A \cap (i' \mathbin{..} 2N)) & \text{if } A \cap (i' \mathbin{..} 2N) \neq \emptyset. \end{cases}$$

Let this be denoted $i = f(A, i')$. Then the following system is a downward simulation of the previous one after adding one **null** action.

$$
\begin{array}{l}
\mathbf{var}\ m, n : \mathbf{seq}_{2\mathrm{N}}\, Msg;\ n' : \mathbb{P}(0 \mathbin{..} 2N);\ i, j, i' : 0 \mathbin{..} 2N \bullet \\
i' = i = j = 0 \wedge n' = (0 \mathbin{..} N) \wedge (\forall k : 0 \mathbin{..} 2N \bullet n[k] = \perp); \\
\mathbf{do}\ \overline{in} \wedge j - i' \neq \pm N \rightarrow in?m[j];\ j := j \oplus 1 \\
\ [\!]\ \operatorname{dom} n \neq i \circ\circ j \rightarrow [\![\mathbf{var}\ k : \mathbb{N} \bullet k :\in i \circ\circ j \setminus \operatorname{dom} n;\ n[k] := m[k]\ ]\!] \\
\ [\!]\ \overline{out} \wedge n[i] \neq \perp \rightarrow out!n[i];\ n[i] := \perp;\ i := i \oplus 1 \\
\ [\!]\ i \notin \operatorname{dom} n \rightarrow n' := A;\ i' := f(n', i') \\
\mathbf{od}
\end{array}
$$

The last action in this system corresponds to sending an acknowledgement down the faulty channel from the receiver to the sender. It refines a **null** action, but only if it can be guaranteed that $i \notin \operatorname{dom} n$ is always eventually false. This requires a liveness condition stating that the second action is always eventually executed, since this is the only action which can add i to $\operatorname{dom} n$. Repeated executions of this action will indeed do this, and moreover if it cannot be executed because it is disabled then it must be the case that $i \in \operatorname{dom} n$ as required. Provided the environment is always eventually willing to accept an output, one of the last two actions will always eventually execute, so the disjunction of the guards is *true* and thus has not been strengthened. This completes the verification that the downward simulation is correct.

Aside: The condition that the environment is always eventually willing to accept an output is stronger than is necessary for an ordinary N-place buffer. It is however necessary for this development: if the environment at some time insists on inputting N messages before receiving any back, then this version of the sliding-window protocol is doomed to deadlock, because an acknowledgement might be lost, causing the system to think that it contains more messages than it actually does until another message is output and acknowledged. ■

In order to remove n and i from the sender's half of the channel action, another action is added:

$$(i' \circ\circ i) \cap n' \neq \emptyset \vee n' \cap \operatorname{dom} n \neq \emptyset \rightarrow \mathbf{skip}\,.$$

If this action is always eventually to terminate, we require that the last action of the most recent system above is always eventually executed, because this is the only action which changes i' and n'. Executing this action does indeed falsify the guard of the above new action. A sufficient liveness condition for the whole system is that one of the sender's actions (the first two in the most recent system) is always eventually executed, and one of the receiver's actions (the last two) is always eventually executed. This is sufficient because the first action in each pair always eventually disables itself, leaving the other to execute and satisfy liveness by the arguments given above. This is a reasonable liveness condition because we expect the sender and receiver to be executed concurrently as two processes so that neither of them halts indefinitely.

The reasoning for the addition of the above action is that its guard covers the cases in $(i' \circ\circ j) \cap n'$ which arise due to i' and n' not being exact representations of i and n. It can be incorporated into the second action of the most recent system, giving the following action.

$$\begin{array}{l} (i' \circ\circ j) \cap n' \neq \emptyset \rightarrow [\![\mathbf{var}\ k : 0\,..\,2N \bullet k :\in (i' \circ\circ j) \cap n'; \\ \qquad\qquad\qquad\qquad\quad \mathbf{if}\ k \in i \circ\circ i \oplus N \rightarrow n[k] := m[k] \\ \qquad\qquad\qquad\qquad\quad\ [\!]\ k \notin i \circ\circ i \oplus N \rightarrow \mathbf{skip} \\ \qquad\qquad\qquad\qquad\quad \mathbf{fi}]\!] \end{array}$$

The sender's half of this action (that is, the top line plus the right-hand side of the assignment in the second line) no longer contains n or i, and so the partitioning of the system is complete.

8.4 Introducing Channels

The definition of the parallel operator $\|$ allows the system to be written as a sender process and a receiver process in parallel. After assignments within the channel actions have been rewritten as communications in parallel (as in section 5.2), the whole system can be written as *Sender* $\|$ *Receiver*, where *Sender* and *Receiver* are as follows.

$$
\begin{array}{l}
\mathit{Sender} \mathrel{\widehat{=}} \\
\quad \mathbf{var}\ m : \mathbf{seq}_{2N}\ \mathit{Msg};\ n' : \mathbb{P}(0\,..\,2N);\ i', j : 0\,..\,2N \bullet \\
\quad i' = j = 0 \wedge n' = (0\,..\,N); \\
\quad \mathbf{do}\ \overline{in} \wedge j - i' \neq \pm N \rightarrow in?m[j];\ j := j \oplus 1 \\
\quad [\!]\ (i' \circ\circ j) \cap n' \neq \emptyset \rightarrow [\![\mathbf{var}\ k : 0\,..\,2N \bullet k :\in (i' \circ\circ j) \cap n';\ ch_0!(k, m[k])\,]\!] \\
\quad [\!]\ \mathit{true} \rightarrow ch_1?n';\ i' := f(n', i') \\
\quad \mathbf{od} \\
\mathit{Receiver} \mathrel{\widehat{=}} \\
\quad \mathbf{var}\ n : \mathbf{seq}_{2N}\ \mathit{Msg};\ i : 0\,..\,2N \bullet \\
\quad i = 0 \wedge (\forall k : 0\,..\,2N \bullet n[k] = \bot); \\
\quad \mathbf{do}\ \mathit{true} \rightarrow [\![\mathbf{var}\ k : 0\,..\,2N;\ M : \mathit{Msg} \bullet ch_0?(k, M); \\
\qquad\qquad\qquad \mathbf{if}\ k \in i \circ\circ i \oplus N \rightarrow n[k] := M \\
\qquad\qquad\qquad\ [\!]\ k \notin i \circ\circ i \oplus N \rightarrow \mathbf{skip} \\
\qquad\qquad\qquad \mathbf{fi}]\!] \\
\quad [\!]\ \overline{out} \wedge n[i] \neq \bot \rightarrow out!n[i];\ n[i] := \bot;\ i := i \oplus 1 \\
\quad [\!]\ n[i] = \bot \rightarrow ch_1!A \\
\quad \mathbf{od}
\end{array}
$$

It is important to notice that the actions which output to channels ch_0 and ch_1 do not record the communication, and that the actions which input from these channels are always enabled. This makes the system tolerant of faulty channels. For instance, the action

$$n[i] = \bot \rightarrow ch_1!A$$

is refined by

$$n[i] = \bot \rightarrow [\![\mathbf{var}\ v : \mathbb{N} \bullet \mathbf{do}\ v > 0 \rightarrow ch_1!A;\ v := v - 1\ \mathbf{od}]\!]$$

(which simulates output on a faulty channel) provided that the value $v = 0$ is chosen at most a finite number of times in succession. Refinement holds because the communication action does not change the state (except in the value of $\langle ch_1 \rangle$)

and an execution of the latter action is equivalent to a succession of zero or more executions of the former action (which is possible because of the nondeterminism of the action-selection mechanism).

A similar observation applies for the other output along the faulty channel; here we also rely on the fact that in normal execution of the system the same value of k can be chosen many times in succession while the output action is repeated by the action-selection mechanism. This is equivalent to the repeated output which is caused by the faulty channel.

8.5 Variations

8.5.1 Alternating-Bit Protocol

In the alternating-bit protocol, each message sent across the faulty channel is tagged with a binary digit, the tag alternating between the values of 0 and 1 for successive messages. Only one message is contained in the system at any instant. After the message has been input, it is repeatedly sent with its tag along the channel until an acknowledgement consisting of the tag value has been received. Another message may then be input for transmission. Repeated messages and repeated acknowledgements may be recognised and discarded because they have the same tag value as the most recent message.

The alternating-bit protocol is a special case of the sliding-window protocol in the case $N = 1$ and the system which has been developed above may easily be modified to implement it. Certain optimisations to the system are possible. For example, the set $A = (i \mathrel{\circ\circ} i+1) \setminus \operatorname{dom} n$ always contains precisely one element whenever it is transmitted as an acknowlegement—namely the value of i—and therefore the function f is no longer needed. The system may be altered to transmit the single element $1 - i$ as an acknowlegdgement instead of the set A, because the two kinds of acknowledgement convey precisely the same information. This gives the algorithm described informally above.

8.5.2 Single-Element Acknowledgements

A common alternative to the sliding-window protocol developed here involves acknowledging each message as it is output by sending its tag number. Thus the acknowledgement action would be

$$n[i] = \perp \rightarrow ch_1!i \ominus 1$$

instead of the last action of the receiver process above (since $i \ominus 1$ is the number of the last message which was output). Then the sender process would not need to calculate i' from n'—it would just take the value received. For this reason the guard $n[i] = \perp$ is not required, although if it is omitted an extra liveness condition would need to be imposed in order to prevent this action from executing ad infinitum. Omitting the guard would have the advantage that the environment no longer needs to guarantee to receive any message until N messages have been sent in succession.

This variation causes the sender process to forfeit n', so it would have to approximate n' by $i' \circ\circ j$. It would then repeatedly send to the receiver process any message in the range $i' \circ\circ j$, and the receiver process would be adapted to discard duplicate messages in much the same way as in the above development. Another liveness condition would be required to ensure that the sender process would not continue to send messages which have already been received ad infinitum. The sender would ensure that message i' is chosen infinitely often, probably choosing k from a fair sequence of random numbers (one which contains each number infinitely often).

9

A Stack Process

9.1 A Simple Implementation

A stack has input nw and output sw and control lines $push$ and pop. Whenever the $push$ control is activated, the stack inputs an element of type T and whenever the pop control is activated, the stack outputs the most recent element which has been input but not output. If no such element exists then the stack outputs the special element $\perp$, which for convenience is assumed to be a member of T that will never be input.

The stack has a specification as follows.

$$r = \mathrm{Inf}(\overline{push} \vee \overline{pop}) \wedge (\overline{push} \rightarrow \overline{nw}) \wedge (\overline{pop} \rightarrow \overline{sw})$$

$$I = stack(\langle nw \rangle, \langle sw \rangle)$$

$$l = \forall N : \mathbb{N}\ \exists n : \mathbb{N} \bullet \#\langle comms \rangle_n > N$$

where $stack(x, y)$ is the following predicate, assuming that each element x_i of the sequence x is distinct:

$$\exists k : 0 \mathinner{..} m \bullet y_n = x_k \quad \wedge \quad k = \max\{i \mid i \in 0 \mathinner{..} m \wedge (\forall j < n \bullet x_i \neq y_j)\}$$

and where $m = \#x$, $n = \#y$ and $\max \emptyset = \perp = x_\perp$. The predicate $stack(x, y)$ need only specify the last element of y because it is an invariant and therefore must also have been true at the times when the other elements of y were created.

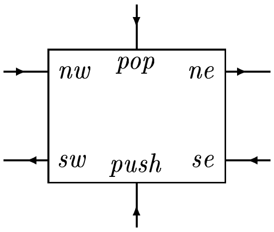
The stack has a simple implementation $Stack$ as follows, using the sequence S to contain all the elements which have been input but not output.

$$\begin{array}{l}
\mathbf{var}\ S : \mathbf{seq}\ T \bullet S = \{\perp\}^*; \\
\mathbf{do}\ true \rightarrow [\![\mathbf{var}\ x : T;\ t \bullet push?t;\ nw?x;\ S := (x : S)]\!] \\
\ [\!]\ true \rightarrow [\![\mathbf{var}\ t \bullet pop?t;\ sw!S_0;\ S := S']\!] \\
\mathbf{od}
\end{array}$$

This action system relies on an external selection mechanism to select the action for which the input on channel $push$ or pop will occur first in order to simplify the development. The values sent along these channels are thrown away, and only their presence or absence affects the system. It can easily be seen that the rely condition r above is sufficient for this system to establish the liveness condition l and, by showing that S always contains the elements which have been input but not output with the most recent element first and with an infinite supply of the element $\perp$ appended, it can be shown that the invariant I is also satisfied by this system.

9.2 A Cellular Implementation

The aim of this development is to implement the stack as a chain of identical hardware cells, so a system C will be introduced to model an individual cell. The following diagram shows such a cell with its inputs and outputs. Neighbours may be connected to the left and right of the cell.



In addition to the channels described in the previous section, each cell will have some channels for internal communication between the cells: input channel se and output channels ne, $push'$ and pop'. The latter two channels will be used for passing on the $push$ and pop signals to the other cells in the chain. Cells will be chained together by connecting the channels ne, se, $push'$ and pop' of one cell to chanels nw, sw, $push$ and pop of its neighbour, respectively. Therefore a chaining operator $\ggg$ will be defined so that $P \ggg Q$ is:

$$P[m_1, m_2, m_3, m_4/ne, se, push', pop'] \parallel Q[m_1, m_2, m_3, m_4/nw, sw, push, pop].$$

With that, we may write a system C which implements the stack when placed in an infinite chain of identical processes:

$$\begin{array}{l}
\mathbf{var}\ x : T \bullet x = \perp; \\
\mathbf{do}\ true \rightarrow [\![\mathbf{var}\ y : T;\ t \bullet push?t;\ push'!t;\ nw?y;\ ne!x;\ x := y]\!] \\
\ [\!]\ true \rightarrow [\![\mathbf{var}\ y : T;\ t \bullet pop?t;\ pop'!t;\ se?y;\ sw!x;\ x := y]\!] \\
\mathbf{od}.
\end{array}$$

We may show that it implements the stack by demonstrating that $Stack$ is a fixed point of the mapping $X \mapsto C \ggg X$, as follows.

$$
\begin{array}{ll}
 & C \ggg Stack \\
= & \textbf{var } S : \textbf{seq } T;\ x : T \bullet x = \perp \wedge S = \{\perp\}^*; \\
 & \textbf{do } true \rightarrow [\![\textbf{var } x', y : T;\ t, t' \bullet \\
 & \qquad\qquad push?t;\ t' := t;\ nw?y;\ x' := x;\ S := (x' : S);\ x := y]\!] \\
 & [\!]\ true \rightarrow [\![\textbf{var } y : T;\ t, t' \bullet \\
 & \qquad\qquad pop?t;\ t' := t;\ y := S_0;\ S := S';\ sw!x;\ x := y]\!] \\
 & \textbf{od} \\
= & \{t' \text{ and } x' \text{ are auxiliary variables}\} \\
 & \textbf{var } S : \textbf{seq } T;\ x : T \bullet x = \perp \wedge S = \{\perp\}^*; \\
 & \textbf{do } true \rightarrow [\![\textbf{var } y : T;\ t \bullet push?t;\ nw?y;\ S := (x : S);\ x := y]\!] \\
 & [\!]\ true \rightarrow [\![\textbf{var } y : T;\ t \bullet pop?t;\ y := S_0;\ S := S';\ sw!x;\ x := y]\!] \\
 & \textbf{od}
\end{array}
$$

which is the same as *Stack*, with $x : S$ of this system corresponding to the S of *Stack*.

9.3 Adding a Refresh Action

Looking ahead to a hardware implementation of the cell which uses dynamic memory, we add a ‘refresh’ action. The refresh action will do nothing except communicate on two new control channels and thus will refine a **null** action in a data refinement which relates two states if and only if they are the same when the new channels are ignored. A suitable equivalence relation $\sim$ for the refinement to occur (section 3.4) is one which considers computations equal if they are equal when the new channels are ignored. The new action will be:

$$true \rightarrow [\![\textbf{var } y : T;\ t \bullet refresh?t;\ refresh'!t;\ y := x;\ x := y]\!]$$

which is inspired by the fact that a dynamic memory refresh of a cell just requires the value that it contains to be stored again. In order for the resulting system to be live, it is simply required that the refresh action is never taken more than a finite number of times in succession.

By making a local change to the names of the variables within the *pop* action, it is possible to rewrite the action as follows.

$$true \rightarrow [\![\textbf{var } y : T;\ t \bullet pop?t;\ pop'!t;\ y := x;\ se?x;\ sw!y]\!]$$

For completeness, the *push* action will be repeated here:

$$true \rightarrow [\![\textbf{var } y : T;\ t \bullet push?t;\ push'!t;\ nw?y;\ ne!x;\ x := y]\!].$$

9.4 Developing a Hardware Implementation

Each request is now split into two signals so that the actions performed for each request can be sequenced more easily. In particular, each signal will correspond to either an assignment or two communications. The *push* request will be split into *push*1 and *push*2, and the *pop* request into *pop*1 and *pop*2. The *refresh* request will become *pop*1 followed by *push*2, as it can be seen that the *refresh* action contains the assignments from the start of the *pop* action and the end of the *push* action. In each case, the first of the two signals initiates the operation and the second is guaranteed to follow before the next operation. A data refinement relation R which models the splitting of the signals is given by the following description of when $a\ R\ c$ for communication histories a and c. For brevity, the output channels (for instance $push'$) have been omitted from the notation, but they are affected in precisely the same way as the input channels.

$$
\begin{aligned}
&a[push1, pop1, pop1/push, pop, refresh] = c \setminus \{push2, pop2\}\\
&a^{-}[(push1;\ push2), (pop1;\ pop2), (pop1;\ push2)/push, pop, refresh] = c^{-}\\
&\text{where } t^{-} = t \restriction \{push, pop, refresh, push1, pop1, push2, pop2\}
\end{aligned}
$$

The first of these indicates that the three operations are initiated by their first components; the second (which replaces each event by a sequence of two events) indicates that the original sequence of *push*, *pop* and *refresh* events is precisely modelled by pairs of operations in the new system. This R is a one-to-many relation, and it is therefore a valid data refinement under an equivalence relation $\sim$ which is the identity relation.

Aside from the above relation on communication histories, the relation R relates two states if and only if they are equal. This relation affects the probes by placing the following conditions on $\overline{push1}$, $\overline{push2}$, $\overline{pop1}$ and $\overline{pop2}$:

$$
\begin{aligned}
&\overline{push} = \overline{push1} \wedge \overline{push2}\\
&\overline{pop} = \overline{pop1} \wedge \overline{pop2}\\
&\overline{refresh} \vee (\overline{pop} \wedge \overline{push}) = \overline{pop1} \wedge \overline{push2}\\
&\overline{push1} \rightarrow \overline{push2}\\
&\overline{pop1} \rightarrow \overline{pop2} \vee \overline{push2}.
\end{aligned}
$$

At this point it is necessary to refer back to the point at which the *refresh* event was introduced and impose an extra rely condition so that it is possible to distinguish between the *refresh* request and the other requests: $(\overline{pop} \vee \overline{push}) \rightarrow \neg\overline{refresh}$. This is not an unreasonable condition since we expect to refresh the stack only during quiet periods.

The requirements which the new stack process will place on the environment may be calculated by applying the above relation to the original rely condition, giving the following result.

$$(\overline{push1} \rightarrow \overline{push2}) \wedge (\overline{pop1} \rightarrow \overline{pop2} \vee \overline{push2}) \wedge$$
$$(\overline{push1} \rightarrow \overline{nw} \wedge \overline{ne}) \wedge (\overline{pop1} \wedge \overline{pop2} \rightarrow \overline{sw} \wedge \overline{se}) \wedge$$
$$\mathrm{Inf}[(\overline{push1} \vee \overline{pop1}) \wedge \neg(\overline{pop1} \wedge \overline{push2})]$$

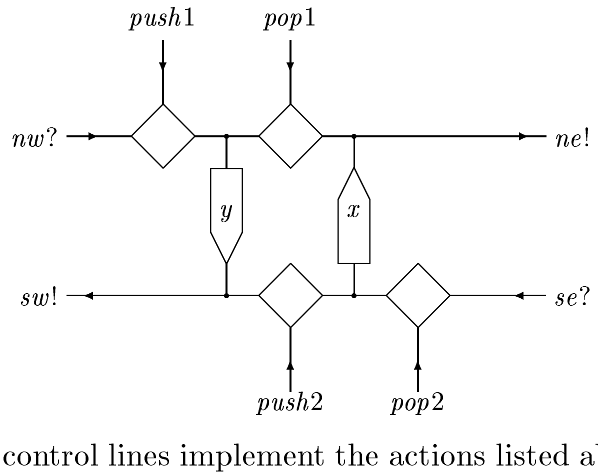
The new system may be obtained by transforming each signal into the first of the new pair of signals, and then inserting the second of the pair into an appropriate place in each action.

$$\begin{array}{l}
\mathbf{var}\ x : T \bullet x = \perp; \\
\mathbf{do}\ \mathit{true} \rightarrow [\![\mathbf{var}\ y : T;\ t \bullet push1?t;\ push1'!t;\ nw?y;\ ne!x; \\
\qquad\qquad\qquad\qquad push2?t;\ push2'!t;\ x := y]\!] \\
\ [\!]\ \mathit{true} \rightarrow [\![\mathbf{var}\ y : T;\ t \bullet pop1?t;\ pop1'!t;\ y := x; \\
\qquad\qquad\qquad\qquad pop2?t;\ pop2'!t;\ se?x;\ sw!y]\!] \\
\ [\!]\ \mathit{true} \rightarrow [\![\mathbf{var}\ y : T;\ t \bullet pop1?t;\ pop1'!t;\ y := x; \\
\qquad\qquad\qquad\qquad push2?t;\ push2'!t;\ x := y]\!] \\
\mathbf{od}
\end{array}$$

It may now be noted that the four signals correspond to simple actions, namely:

$push1 : nw?y;\ ne!x$
$push2 : x := y$
$pop1 : y := x$
$pop2 : se?x;\ sw!y.$

This leads to a simple implementation in hardware, as shown below.



In this design the control lines implement the actions listed above as follows. When *push*1 is activated, the value present at *nw* is passed into the *y* memory element. The value of *x* is simultanously output to the next cell in the chain; in fact the cell always outputs *x* but no current flows until the next cell is ready for input. The signal *pop*2 works in an identical manner to perform the input of *x* from *se* and the output of *y* to *sw*. The *pop*1 signal connects *x* to *y* and therefore performs the assignment $y := x$, and in a similar manner the *push*2 signal performs the assignment $x := y$. The output signal $push1'$ which appears in the action system is implemented simply by connecting a wire from *push*1 to the similarly named input in the neighbouring cell, and similarly for the other control wires.

Provided that the control signals are activated in the correct order as dictated by the action system (for example, with *pop*1 being followed by *pop*2 for a *pop* operation), a chain of hardware cells identical to the above will correctly implement the stack.

10

Multiplying Matrices on a Systolic Array

A sequence of instructions for multiplying matrices will be derived in this chapter using a combination of operation refinements, each of which replaces the sequence of instructions by another whose reachable states are the same, and data refinements which change the way in which data is input and output. This example is provided because the resulting instructions may be implemented on a variety of different architectures using only slightly different state transformations to map the local variables into processors. The last architecture shown is the familiar hexagonal array.

10.1 Specification

We are to develop a reactive action system which is suitable for implementation on a systolic array and whose function is to multiply pairs of matrices. Initially we will consider the input and output of an entire $n \times n$ matrix, so that the types of the input and output are:

$\langle in_x \rangle, \langle in_y \rangle, \langle out \rangle : \mathbf{seq}(Mat_{n\times n}\,\mathbb{R})$
(where $Mat_{n\times n}\,\mathbb{R}$ may be interpreted as $\mathbf{seq}_{n^2}\,\mathbb{R}$).

For a safety condition we follow the usual practice of insisting that the system will compute arbitrarily many matrix products given suitable co-operation from the environment. This, together with the invariant that each output is the matrix product of the two corresponding inputs, gives the following specification.

$$r = \mathrm{Inf}(\overline{in_x}) \wedge \mathrm{Inf}(\overline{in_y}) \wedge \mathrm{Inf}(\overline{out})$$

$$I = \#\langle in_x \rangle = \#\langle in_y \rangle = \#\langle out \rangle \ \wedge \ (\forall i : 0 \mathinner{..} \#\langle out \rangle \bullet \langle out \rangle_i = \langle in_x \rangle_i \times \langle in_y \rangle_i)$$

$$l = \forall M : \mathbb{N}\ \exists m : \mathbb{N} \bullet \#\langle comms \rangle_m > M$$

An initial action system satisfying this specification is:

$$\begin{array}{l}\textbf{var } A, B : Mat_{n\times n}\,\mathbb{R} \bullet \\ \textbf{do } true \rightarrow in_x?A;\ in_y?B;\ out!A \times B \\ \textbf{od}.\end{array}$$

Throughout the development we will apply data refinements to the system and operation refinements to the body of the action. Such an operation refinement is a special case of a downward simulation when the data refinement relation is the identity, and so preserves correctness.

10.2 Towards Implementation on n Processors

The development proceeds as follows. The notation m^T denotes the transpose of m.

$$\begin{array}{ll} & in?A;\ in?B;\ out!A \times B \\ \sqsubseteq & \textbf{var } a_0, \ldots, a_{n-1};\ b_0, \ldots, b_{n-1} : \mathbf{seq}_{\mathrm{n}}\,\mathbb{R} \\ & \{\text{where } A = \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1}\end{pmatrix} \text{ and } B = (b_0^T\ \cdots\ b_{n-1}^T)\}; \\ & in_x?a_0, \ldots, a_{n-1};\ in_y?b_0, \ldots, b_{n-1};\ out!\{a_i \cdot b_j \mid i, j \in 0\,..\,n\} \\ & \{\text{where } x \cdot y \text{ is the scalar product of vectors } x \text{ and } y\} \\ \sqsubseteq & in_x?a_0, \ldots, a_{n-1}; \\ & \underset{k\in 0..n}{\fatsemi}\ in_y?b_k;\ out!(a_0 \cdot b_k\ \cdots\ a_{n-1}\cdot b_k) \end{array}$$

The first step above just expresses a formula for multiplying two matrices in terms of rows of the first matrix and columns of the second. The input events written in the form $c?x_0, \ldots, x_{n-1}$ simply input all the values together in a similar manner to an input event of the form $c?A$ where A is a matrix. The second step, in which the notation $\fatsemi_{k\in 0..n} S_k$ expresses the sequence of instructions $S_0;\ \ldots;\ S_{n-1}$, splits the input matrix B into a sequence of vector inputs. The output matrix is similarly split up. For this data refinement the translation between computations of the original system and those of the new system is simple, although the order of the communications has been changed slightly, so the refinement is valid. That is, since it is possible to write a one-to-one relation R which relates original communication sequences to new communication sequences, it is easy to show that the conditions of section 7.2 are satisfied.

Noting that $a_j \cdot b_k = \sum_{i=0}^{n-1} a_{ji} b_{ik}$ we can implement the calculation of the output vector $(a \cdot b_k\ \cdots\ a_{n-1} \cdot b_k)$ as follows.

$$\begin{array}{ll} & out!(a \cdot b_k\ \cdots\ a_{n-1}\cdot b_k) \\ \sqsubseteq & \textbf{var } t : \mathbf{seq}_{\mathrm{n}}\,\mathbb{R} \bullet \\ & t := (a \cdot b_k\ \cdots\ a_{n-1}\cdot b_k);\ out!t \end{array}$$

$$\begin{aligned}\sqsubseteq\quad & t := \mathbf{0};\\ & \mathop{;}_{i\in 0..n}\left(\mathop{\|}_{j\in 0..n} t_j := t_j + a_{ji}\,b_{ik}\right)\end{aligned}$$

Note that section 5.2 defines the result of placing assignments in parallel as being the sequence of those assignments in any order, given that they do not have variables in common (assuming that t_j and a_{ji} are different variables for different values of j, and treating each b_{ik} as a constant for the purposes of the parallel assignment).

We now have a sequence of instructions which can be written as follows, noting that input actions commute with assignments as long as the input data is not used in the assignment in question. In other words, the possible states of the system after executing $c?y;\ x := E$ are the same as those after executing $x := E;\ c?y$ providing E does not contain y. Each input of a vector b_k has now been split into n inputs of individual elements of the vector. Again, this is valid because there is a simple correspondence between the computations involving single-element inputs and those involving vector inputs.

$$\begin{aligned}& in_x?a_0,\ldots,a_{n-1};\\ & \mathop{;}_{k\in 0..n}\ t := \mathbf{0};\\ & \qquad \left[\mathop{;}_{i\in 0..n} in_y?b_{ik};\ \left(\mathop{\|}_{j\in 0..n} t_j := t_j + a_{ji}\,b_{ik}\right)\right];\\ & \qquad out!t\end{aligned}$$

If each element of the sequence t is local to a separate process then each processor requires a different set of elements of A. However, each processor requires element b_{ix} of B at the same time. Thus we transform the i in the assignment into $i-j$ so that each processor accesses a different b_{ix} at any one step.

$$\begin{aligned}& \mathop{;}_{i\in 0..n} in_y?b_{ik};\ \left(\mathop{\|}_{j\in 0..n} t_j := t_j + a_{ji}\,b_{ik}\right)\\ \sqsubseteq\quad & \left[\mathop{;}_{i\in 0..n} in_y?b_{ik};\ \left(\mathop{\|}_{j\le i} t_j := t_j + a_{j\ i-j}\,b_{i-j\ k}\right)\right];\\ & \left[\mathop{;}_{i\in n..2n-1} \left(\mathop{\|}_{i-n<j<n} t_j := t_j + a_{j\ i-j}\,b_{i-j\ k}\right)\right]\end{aligned}$$

In order to prove this refinement correct, we observe firstly that each assignment or input occurring in one system occurs precisely once in the other (since in the latter system the value of $i-j$ for any particular j starts at zero when $i=j$ and increases with i until $i-j=n-1$), and secondly that all the assignments commute (providing that $b_{i-j\ k}$ has already been input when it is used, which is clearly true as an invariant is that $b_{i'k}$ has already been input for each $i' \le i$).

This sequence of instructions can be optimised slightly by observing that an assignment $t_j := 0$ commutes with the parallel assignment in the above system whenever $j > i$ since t_j is not accessed by the parallel assignment in that case, and that $out!t_j$ likewise commutes with the parallel assignment when $j < i - n$. Output of the vector t will be split into the outputs of its individual elements. The body of the action system which results is:

$$
\begin{array}{l}
in_x?a_0, \ldots, a_{n-1};\\
\underset{k \in 0..n}{\fatsemi} \left[\underset{i \in 0..n}{\fatsemi} in_y?b_{ik};\ t_i := 0;\ \left(\underset{j \le i}{\|}\ t_j := t_j + a_{j\ i-j} b_{i-j\ k} \right) \right];\\
\qquad \left[\underset{i \in n..2n}{\fatsemi} out!t_{i-n};\ \left(\underset{i-n<j<n}{\|}\ t_j := t_j + a_{j\ i-j} b_{i-j\ k} \right) \right].
\end{array}
$$

In order to implement this system on n processors we may observe that each element b_{ik} travels up the array (towards increasing j) as shown in figure 10.1 and requires only one memory element (t_j) per processor apart from any memory needed to store the matrix elements.

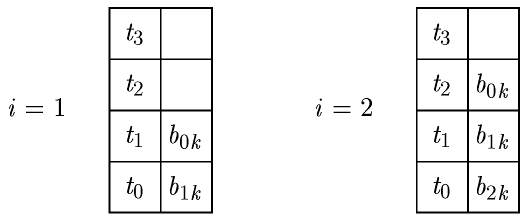


Figure 10.1: The movement of matrix B through n processors.

An efficient implementation of this algorithm would also merge together adjacent iterations of the loop in k, resulting in a system which outputs the results for one vector of the matrix product while inputting the next vector of B. This system has the disadvantage, however, that the entire matrix A must be input before computation begins. The system could perhaps be used for efficiently multiplying by a constant matrix A.

10.3 Towards Implementation on n^2 Processors

In the above system j may be viewed as a process number and i as a step number in the computation. The counter k is neither of these—it merely repeats the whole process for each column of the result. We can imagine each column being calculated by an independent set of n processors so that the columns are computed simultaneously. Each would compute one column and need not share any variable with the others except for the matrix A. Hence we give each local variable t_i an extra subscript k and, instead of having an outer loop over k, write n copies of each instruction within the loop. This changes the orders of some of the assignments, but since for each value of k a different set of variables is accessed (apart from the constant matrix A), the outcome of the sequence of assignments is not changed. It also changes the orders of the inputs and outputs, but as before there is a well-defined correspondence between the original order and the new order. The result of applying this refinement is the following action system body.

$$
\begin{array}{l}
in?a_0, \ldots, a_{n-1};\\
\left[\mathop{\fatsemi}_{i \in 0..n} \left(\mathop{\fatsemi}_{k \in 0..n} in_y?b_{ik}\right); \left(\mathop{\|}_{k \in 0..n} t_{ik} := 0\right); \left(\mathop{\|}_{\substack{k \in 0..n\\ j \le i}} t_{jk} := t_{jk} + a_{j\ i-j} b_{i-j\ k}\right)\right];\\
\left[\mathop{\fatsemi}_{i \in n..2n} \left(\mathop{\fatsemi}_{k \in 0..n} out!b_{i-n\ k}\right); \left(\mathop{\|}_{\substack{k \in 0..n\\ i-n<j<n}} t_{jk} := t_{jk} + a_{j\ i-j} b_{i-j\ k}\right)\right]
\end{array}
$$

The sequence of n inputs in this system could be implemented in hardware in parallel as n inputs along different wires, and similarly for the sequence of n outputs. They are written as sequences for convenience in the notation.

We have a system which runs on n^2 processors, but which shares data (namely the elements of the matrix A) between the processors. This data sharing may be eliminated in a manner very similar to the refinement which eliminated the sharing of b_{ik} in the previous system.

At step i, the set of parallel assignments for each value of k in the above system requires the set of elements $\{j : 0..n \mid j \le i \bullet a_{j\ i-j}\}$. This is an (anti-)diagonal of A. We wish the assignment to use a different diagonal for each k rather than requiring the same one for each different value of k. We therefore transform i into $i-k$. Each of the two loops on i is written as two loops, as before:

$$
\begin{array}{l}
in_x?a_0, \ldots, a_{n-1};\\
\left[\mathop{\fatsemi}_{i \in 0..n} \left(\mathop{\fatsemi}_{k \le i} in_y?b_{i-k\ k}\right); \left(\mathop{\|}_{k \le i} t_{i-k\ k} := 0\right); \left(\mathop{\|}_{\substack{k \le i\\ j \le i-k}} add(j, k, i-k-j)\right)\right];\\
\left[\mathop{\fatsemi}_{i \in n..2n} \left(\mathop{\fatsemi}_{i-n<k<n} in_y?b_{i-k\ k}\right); \left(\mathop{\|}_{i-n<k<n} t_{i-k\ k} := 0\right); \left(\mathop{\|}_{\substack{i-n<k<n\\ j \le i-k}} add(j, k, i-k-j)\right)\right];\\
\left[\mathop{\fatsemi}_{i \in n..2n} \left(\mathop{\fatsemi}_{k \le i-n} out!t_{i-n-k\ k}\right); \left(\mathop{\|}_{\substack{k \le i-n\\ i-k-n<j<n}} add(j, k, i-j-k)\right)\right];\\
\left[\mathop{\fatsemi}_{i \in 2n..3n} \left(\mathop{\fatsemi}_{k \le i-2n} out!t_{i-n-k\ k}\right); \left(\mathop{\|}_{\substack{i-2n<k<n\\ i-k-n<j<n}} add(j, k, i-j-k)\right)\right]
\end{array}
$$

where $add(x, y, z) \mathrel{\widehat{=}} t_{xy} := t_{xy} + a_{xz} b_{zy}$.

In this sequence of instructions, the middle two lines contain elements which commute. If

$$k' \le i' - n \ \wedge\ i - n < k < n \ \wedge\ j \le i - k \ \wedge\ i' \le i$$

(this condition having been taken from several quantifiers in the two lines being discussed, together with $i' < i$ to indicate that instructions with low values of i'

will be moved leftwards past instructions with high values of i) then the instruction $out!t_{i'-n-k'\;k'}$ commutes with $add(j,k,i-k-j)$ and with $t_{i-k\;k} := 0$ since $k' \leq i'-n \leq i-n < k$ implies that $k' \neq k$. Also, if

$$k' \leq i'-n \ \wedge\ i'-k'-n < j' < n \ \wedge\ i-n < k < n \ \wedge\ i' \leq i$$

then $add(j',k',i'-j'-k')$ commutes with $add(j,k,i-k-j)$ (because addition is commutative) and with $t_{i-k\;k} := 0$ (because $k \neq k'$). Therefore the middle two lines may be merged, giving the following line. This again changes the order of the inputs and outputs in a well-defined way.

$$\begin{aligned}\Bigg[\mathop{\fatsemi}_{i\in n..2n} \Bigg(\mathop{\fatsemi}_{i-n<k<n} in_y?b_{i-k\;k}\Bigg) ; \Bigg(\mathop{\fatsemi}_{k\leq i-n} out!t_{i-n-k\;k}\Bigg) ; \Bigg(\mathop{\|}_{i-n<k<n} t_{i-k\;k} := 0\Bigg) ;\\ \Bigg(\mathop{\|}_{\substack{i-n<k<n\\ j\leq i-k}} add(j,k,i-k-j)\Bigg) \,\|\, \Bigg(\mathop{\|}_{\substack{k\leq i-n\\ i-k-n<j<n}} add(j,k,i-j-k)\Bigg)\Bigg]\end{aligned}$$

Now

$$\begin{aligned} & 0 \leq i-n < k < n \ \wedge\ 0 \leq j \leq i-k \\ \Leftrightarrow\ & i-n < k \ \wedge\ i-n < j+k \leq i \ \wedge\ i-n, j, k \in 0\,..\,n \\ & \text{and} \\ & 0 \leq k \leq i-n < n \ \wedge\ i-k-n < j < n \\ \Leftrightarrow\ & 0 \leq k \leq i-n \ \wedge\ i-n < j+k \leq i \ \wedge\ i-n, j, k \in 0\,..\,n \end{aligned}$$

and therefore the two parallel additions above can be written as a single parallel addition, with quantifiers $i-n < j+k \leq i$ and $j,k \in 0\,..\,n$—this being the union of the two disjoint regions specified by the quantifiers above.

Now that data sharing has been eliminated, we can show that it is no longer necessary to input the whole matrix A at once. At any step i in $0\,..\,n$ the system accesses $a_{j\;i-k-j}$ for each $k \leq i$, $j \leq i-k$; in other words, $\{a_{xy} \mid x+y \leq i\}$. This applies also for each i in $n\,..\,2n$. When $i \geq 2n$ all elements of A have been accessed. We can commute the inputs of A forward until they are needed, and write the whole action system body as follows (the variables j and k are each constrained to be in the range $0\,..\,n$; this condition will be omitted from further versions of the system).

$$\begin{aligned}\mathop{\fatsemi}_{i\in 0..3n} \Bigg(\mathop{\fatsemi}_{i-n<k\leq i} in_y?b_{i-k\;k}\Bigg) ; \Bigg(\mathop{\fatsemi}_{i-n<k\leq i} in_x?a_{i-k\;k}\Bigg) ; \Bigg(\mathop{\fatsemi}_{i-2n<k\leq i-n} out!t_{i-k-n\;k}\Bigg) ;\\ \Bigg(\mathop{\|}_{i-n<k\leq i} t_{i-k\;k} := 0\Bigg) ; \Bigg(\mathop{\|}_{i-n<j+k\leq i} add(j,k,i-k-j)\Bigg)\end{aligned}$$

(Notice that for $i \in 0\,..\,n$, $i \in n\,..\,2n$ or $i \in 2n\,..\,3n$ this loop body reduces to one of the three lines in the earlier system except that the inputs of A have been moved into the loop. Also for $i = 3n-1$ the loop body reduces to **skip**).

Intuitively we can see that the data flow of matrix A is to the right (towards increasing k) and that of B is upwards (towards increasing j) by the following argument: at step i, the processor (j,k) which holds t_{jk} accesses $a_{j\ i-k-j}$ and $b_{i-k-j\ k}$. The inputs at this step are $\{a_{i-k\ k}\}$ and $\{b_{i-k\ k}\}$ which are used by processors $\{(i-k,0)\}$ and $\{(0,k)\}$ respectively—these are on the edges of the array. At step $i+1$, any processor $(j,k+1)$ accesses $a_{j\ i-k-j}$ and processor $(j+1,k)$ accesses $b_{i-k-j\ k}$, both of which were previously accessed by processor (j,k). However, the outputs from this system are not always at an edge of the array.

10.4 Implementation on a Systolic Array

If the array on which the system is implemented is to be systolic, then each processor should be relatively simple, with a constant-sized memory and a constant number of connections to its nearest neighbours. Inputs from and outputs to the environment should only be done at an edge of the array. Moreover, the processors should be identical. An individual processor (j,k) should not depend upon the values j, k or n.

The most important failing of the above system is that its outputs do not come from the edges of the array. There are, however, several different ways in which to implement the system on a systolic array.

10.4.1 Matrix C remains fixed until output

The method which outputs the product in the least time appears to require the most complex processors. The system is implemented in the most natural manner on the array of processors, except that the output values, when complete, migrate towards the edge of the array. Since the data on each diagonal $\{t_{i-k-n\ k}\}$ of the product matrix C are completed simultaneously it seems logical to allow the data to travel out along this diagonal. This entails replacing a simple output $out!t_{jk}$ with a sequence of assignments and an output:

$$t_{j-1\ k+1} := t_{jk};\ t_{j-2\ k+2} := t_{j-1\ k+1};\ \ldots;\ out!t_{0\ j+k}\ (\text{or } out!t_{j+k-n\ n}).$$

We can easily show that none of the variables in this sequence is accessed by the rest of the system in the meantime, and that this sequence refines $out!t_{jk}$ under a relation which compares only the communications of the system and not the final values of $\{t_{jk}\}$. We then have a loop body of:

$$\left(\mathop{⨟}_{i-n<k\le i} in_y?b_{i-k\ k}\right);\ \left(\mathop{⨟}_{i-n<k\le i} in_x?a_{i-k\ k}\right);$$

$$\left[\ \mathop{\fatsemi}_{\substack{d:d\le i-n\le 2d\\ \wedge 0\le d<2n-1}} (\textbf{if } d<n \to out!t_{0d} \mathbin{[\,]} d\ge n \to out!t_{d+1-n\ n-1}\ \textbf{fi});\right.$$
$$\left(\mathop{\fatsemi}_{\substack{j,k:j+k=d\\ \wedge j>0\wedge k<n-1}} t_{j-1\ k+1} := t_{jk}\right);$$
$$\left.(\textbf{if } d<n\to t_{d0}:=0 \mathbin{[\,]} d\ge n \to t_{n-1\ d+1-n} := 0\ \textbf{fi})\right];$$
$$\left(\mathop{\|}_{i-n<k\le i} t_{i-k\ k} := 0\right);\ \left(\mathop{\|}_{i-n<j+k\le i} add(j,k,i-k-j)\right)$$

(where the sequence of assignments is performed in an order such that no variable is written before being read). The expanded output action has the side effect that if $i > \min\{2d+n, 3n-2\}$ then $\forall j,k \mid j+k=d \bullet t_{jk}=0$, since it contains some extra assignments. Zeros are introduced along the bottom and right-hand edges of the array, which are propagated along the diagonals as if they were values waiting to be output (see figure 10.2 on page 74). This property can easily be shown with an invariant ($n+d+j<i \Rightarrow t_{j\ d-k}=0$ if $d<n$ and a similar one if $d\ge n$). This implies that the entire array of processors holds $t_{jk}=0$ when the loop finishes. Thus the original action system (of which we are developing an action body) has an invariant of $\forall j,k \bullet t_{jk}=0$ and it is easy to see that the initialisation ($\| \ t_{i-k\ k} := 0$) is not required.

It remains to model the data flow of A and B. If we add local variables x_{jk} and y_{jk} (to hold elements of A and B respectively) to each processor then we can show that the values of x_{jk} and y_{jk} in the following system are those of $a_{j\ i-k-j}$ and $b_{i-k-j\ k}$ in the previous system, since it follows from the last paragraph of section 10.3.

$$\mathop{\fatsemi}_{i\in 0..3n} \left(\mathop{\fatsemi}_{j,k:k<n-1} x_{j\ k+1} := x_{jk}\right);\ \left(\mathop{\fatsemi}_{j,k:j<n-1} y_{j+1\ k} := y_{jk}\right);$$
$$\left(\mathop{\fatsemi}_{i-n<k\le i} in_y?y_{0k}\right);\ \left(\mathop{\fatsemi}_{i-n<k\le i} in_x?x_{i-k\ 0}\right); \qquad \{b_{i-k\ k} \text{ is stored in } y_{0k} \text{ and } a_{i-k\ k} \text{ in } x_{i-k\ 0}\}$$
$$\left[\ \mathop{\fatsemi}_{\substack{d:0\le d<2n-1\\ \wedge d\le i-n}} (\textbf{if } d<n \to out!t_{od} \mathbin{[\,]} d\ge n \to out!t_{d+1-n\ n-1}\ \textbf{fi});\right.$$
$$\left(\mathop{\fatsemi}_{\substack{j,k:j+k=d\\ \wedge j>0\wedge k<n-1}} t_{j-1\ k+1} := t_{jk}\right);$$
$$\left.(\textbf{if } d<n\to t_{d0}:=0 \mathbin{[\,]} d\ge n \to t_{n-1\ d+1-n} := 0\ \textbf{fi})\right];$$
$$\left(\mathop{\|}_{i-n<j+k\le i} t_{jk} := t_{jk} + x_{jk}y_{jk}\right)$$

The range of d in the middle section has been increased for easier implementation. We can easily see that the extra assignments have no effect because all the variables involved are already zero. The extra outputs may be ignored by the environment; this data refinement is valid because as usual there is a simple correspondence between the communication sequences of the previous system and those of this system.

This system can now be implemented on a systolic array. The two **if** statements are used in the action system to model the edges of the array and do not add any processing. The restriction $d \leq i - n$ in the middle sequence of instructions is the hardest to implement; nevertheless it can be eliminated by observing that it becomes true just after the last element from matrix A or B has arrived at the processor. If $j + k = d$ then processor (j, k) has elements $a_{j\,i-d}$ and $b_{i-d\,k}$ of A and B respectively. If we make $a_{jk} = End$ when $k \geq n$ and $b_{jk} = End$ when $j \geq n$ then

$$d = j + k \wedge x_{jk} = End \Leftrightarrow i - d \geq n \Leftrightarrow d \leq i - n.$$

Similarly, if we assume that $x_{jk} = y_{jk} = End$ initially, so that this becomes an invariant (like $t_{jk} = 0$), thus ensuring that $a_{jk} = End$ when $k < 0$ and $b_{jk} = End$ when $j < 0$, then we also have that $i - n < j + k \leq i \Leftrightarrow x_{jk} \neq End$. We have arrived at an implementation in which i, j, k, d and n have been eliminated from each individual processor; each processor is described by the following system.

$$
\begin{array}{l}
\textbf{var } x, y, t : \mathbb{R} \bullet x = End \wedge y = End \wedge t = 0; \\
\textbf{do } \mathit{true} \rightarrow out_y!y;\ in_y?y;\ out_x!x;\ in_x?x; \\
\qquad\qquad \textbf{if } x = End \rightarrow out!t;\ in?t \\
\qquad\qquad [\!]\ x \neq End \rightarrow t := t + xy \\
\qquad\qquad \textbf{fi} \\
\textbf{od}
\end{array}
$$

Given that, except on the edges of the array, channel in_x of processor (j, k) is connected to channel out_x of processor $(j, k-1)$, channel in_y of processor (j, k) is connected to channel out_y of processor $(j-1, k)$, and channel in of processor (j, k) is connected to channel out of processor $(j-1, k+1)$, it is possible to write n^2 copies of the above system in syntactic parallel. After ignoring the communications on channels out_x and out_y at the edges of the array and adding a process which feeds a constant stream of zeros into each channel in at the edges of the array, the result is the action system which has been developed. For example, the n^2 inputs and outputs $out_y!y;\ in_y?y$ result in the sequences of $y_{j+1\,k} := y_{jk}$ instructions and also, at the lower edge of the array, the $in_y?y_{0k}$ instructions (there will be n of these input instructions, but since the matrix B has been extended with End elements, there are now also n input instructions in the previous system, the extra ones inputting the value End).

The array of processors may be arranged with the k axis running rightwards and the j axis running upwards, as shown in figure 10.2 on page 74.

It is clear by inspection of the system that this array will take no more than $3n$ steps to multiply two matrices using n^2 processors. In fact, the last step does

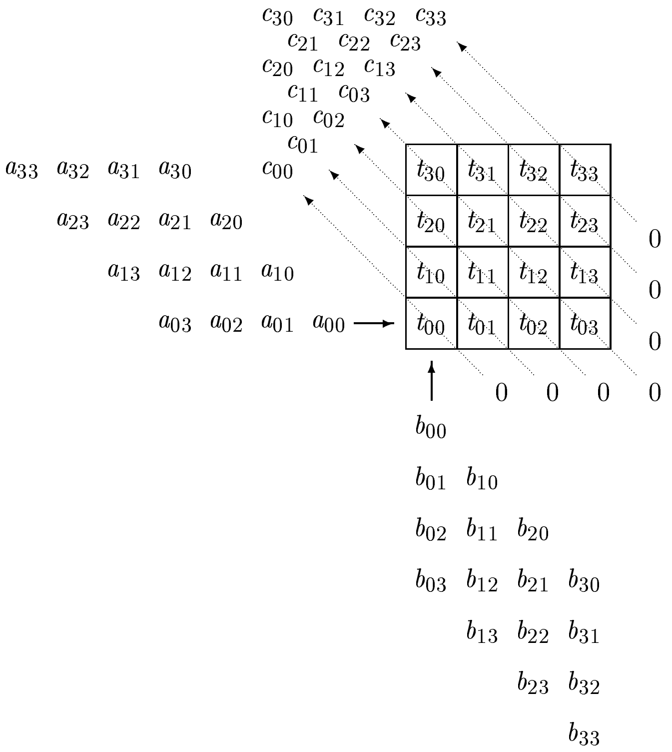


Figure 10.2: Multiplying matrices on an $n \times n$ array of processors.

nothing useful since in an earlier system all the quantifiers were empty in the case $i = 3n - 1$—thus the time required is $3n - 1$ steps.

10.4.2 Outputs migrate towards the origin

Another way of implementing the system on a systolic array is to make all the data slowly migrate towards the origin. The dth diagonal of the result is computed at step $n + d$; an element lying on diagonal d which moves towards the origin (from (j, k) to $(j - 1, k - 1)$) will as a result lie on diagonal $d - 2$. Hence the data could move towards the origin on every alternate step of the computation. This means we should store t_{jk} at processor $(j - \lfloor \frac{i}{2} \rfloor, k - \lfloor \frac{i}{2} \rfloor)$.

The co-ordinates of all the processors used in this computation are given by the set

$$\{i : 0 \mathrel{..} 3n, j : 0 \mathrel{..} n, k : 0 \mathrel{..} n \mid i - n < j + k \leq i \bullet (j - \lfloor \tfrac{i}{2} \rfloor, k - \lfloor \tfrac{i}{2} \rfloor)\}$$

which turns out to be a set of $n^2 + \lfloor \frac{n}{2} \rfloor$ processors arranged in a $n \times \lfloor \frac{n}{2} + 1 \rfloor$ rectangle whose sides are at 45° to the axes. Examples are shown in figure 10.3.

The processors which need to output results are given by the set

$$\{i : n \mathrel{..} 3n, k : 0 \mathrel{..} n \mid i - 2n < k \leq i - n \bullet i - n - k - \lfloor \tfrac{i}{2} \rfloor, k - \lfloor \tfrac{i}{2} \rfloor)\}.$$

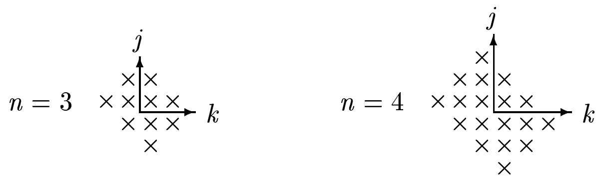


Figure 10.3: The positions of processors in section 10.4.2.

Each of these processors is on diagonal $-n+i-2\lfloor \frac{i}{2} \rfloor$, which is either $-n$ or $-n+1$. The processors on diagonal $-n+1$ are on the edge of the array, whilst the processors on diagonal $-n$ do not exist; the outputs which should have come from these processors will in fact have just emerged from the edge of the array when the data was shifted one place towards the origin. Thus the outputs always arrive at the edge of the array.

As before we may develop an action system to imitate the array, in which the main step is to make a data refinement which replaces $t_j k$ by a variable $p_{j-\lfloor \frac{i}{2} \rfloor\ k-\lfloor \frac{i}{2} \rfloor}$ which reflects the position of each element within the new array. In this system initialisation is carried out by inserting zeros along the lines $j+k \geq 0$ at each alternate step when the data moves toward the origin. Each processor is described by the following system and the array will look like that shown in figure 10.4 on page 76.

$$
\begin{array}{l}
\textbf{var } x, y, p : \mathbb{R};\ b : \mathbb{B} \bullet b := 0; \\
\textbf{do } b \rightarrow north!y;\ south?y;\ east!x;\ west?x; \\
\qquad\qquad p := p + xy; \\
\qquad\qquad b := \neg b \\
\ [\!]\ \neg b \rightarrow west!y;\ east?y;\ south!x;\ north?x;\ sw!p;\ ne?p; \\
\qquad\qquad p := p + xy; \\
\qquad\qquad b := \neg b \\
\textbf{od}
\end{array}
$$

Although all the outputs are made available as soon as they have been completed in this array, the time for each matrix multiplication is longer because the first computation takes place at processor $(0, 0)$ which is no longer on the edge of the array. The elements of each matrix will reach their initial positions on the nth step if they are fed in to the edges of the array in same manner in which they travel once the algorithm is started (no change to any processor is required for this). Hence the nth step of the computation in this array corresponds to the first step in the previous array, and this array will multiply two $n \times n$ matrices a using $n^2 + \lfloor \frac{n}{2} \rfloor$ processors in $4n - 2$ steps.

10.4.3 Implementation on a diamond-shaped array

A variation on the above array is to store t_{jk} at processor $(j - \frac{i}{2}, k - \frac{i}{2})$— thus allowing

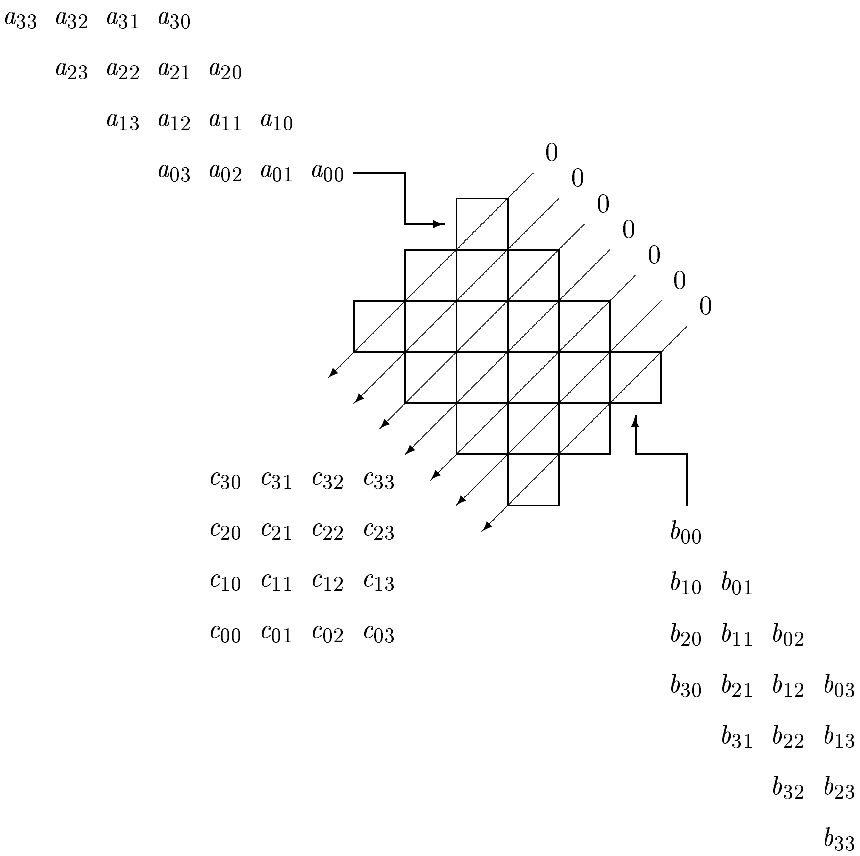


Figure 10.4: Multiplying matrices with outputs flowing towards the origin.

processors at non-integral co-ordinates. This has the advantage that each step of the computation is identical, and therefore simple 'transfer-and-add' processors may be used.

The array is similar to the previous one with processors added at the non-integral co-ordinates. The fact that $\frac{i}{2}$ is no longer truncated to an integer means that in this array we have $j + k \leq 0$ and the outputs are always at processors (j, k) such that $j + k = -n$. The array is a rectangle of size $n \times (2n - 1)$ whose sides are at 45° to the axes. The development is very similar to that of the previous aray, and each processor is described as follows, with the array shown in figure 10.5.

$$
\begin{array}{l}
\textbf{var } x, y, p : \mathbb{R} \bullet \\
\textbf{do } \mathit{true} \rightarrow nw_y!y;\ se_y?y;\ se_x!x;\ nw_x?x;\ sw!p;\ ne?p; \\
\qquad\qquad p := p + xy \\
\textbf{od}.
\end{array}
$$

This array multiplies two $n \times n$ matrices using $2n^2 - n$ processors in $4n - 2$ steps.

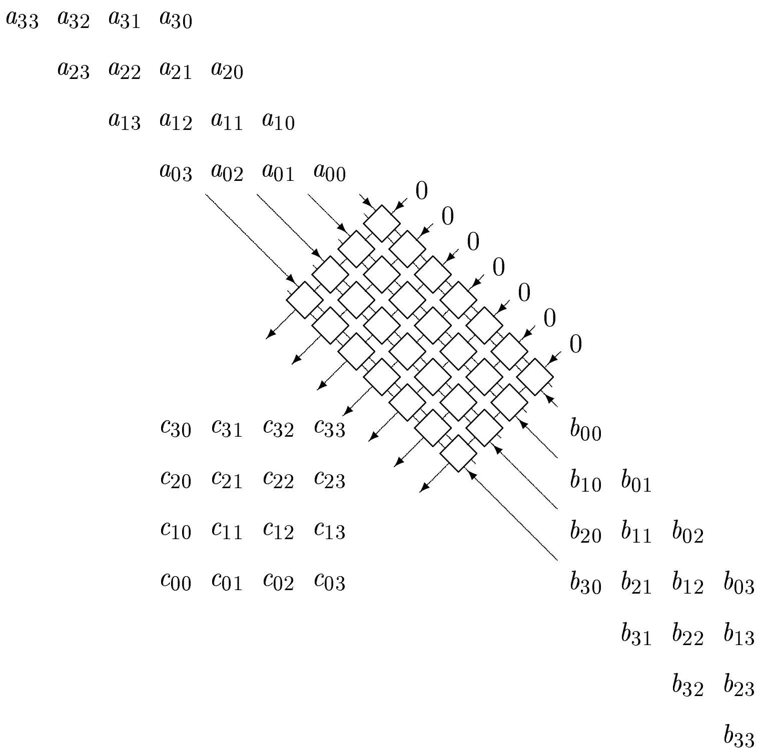


Figure 10.5: Multiplying matrices on a diamond-shaped array.

10.4.4 Implementation on a hexagonal array

The hexagonal array of [16] et al may be derived by allowing t_{jk} to be stored at processor $(j - \frac{i}{3}, k - \frac{i}{3})$. The advantage of this method is in multiplying band matrices. If $x_{j-\frac{i}{3}\ k-\frac{i}{3}} = a_{j\ i-k-j}$ then $x_{jk} = a_{j+\frac{i}{3}\ -j-k+\frac{i}{3}}$ (assuming that $j + \frac{i}{3}$ and $-j - k + \frac{i}{3}$ are integral), meaning that the difference between the two subscripts of a is independent of i. That is, x_{jk} holds elements from the $2j + k$th leading diagonal of A. If A and B are band matrices with only a small number of leading diagonals, then the number of processors is reduced; in fact the array requires at most $w_1 w_2$ processors, where w_1 and w_2 are the band widths (or the numbers of diagonals) of A and B respectively. In contrast, the arrays of preceding subsections may not be significantly reduced if the matrices are band matrices.

Each processor in the hexagonal array is a simple element identical to those in the previous array, but the data travel in different directions. The full set of processors required to multiply two matrices is given by

$$\{i : 0 \,..\, 3n, j : 0 \,..\, n, j : 0 \,..\, n \mid i - n < j + k \le i \bullet (j - \tfrac{i}{3}, k - \tfrac{i}{3})\}$$

which is a hexagon having n processors on each side and having sides parallel to the lines $j + 2k = 0$, $2j + k = 0$ and $j - k = 0$ (the hexagonal array can also clearly be transformed into a regular hexagon). The array layout is as shown in figure 10.6 on page 78.

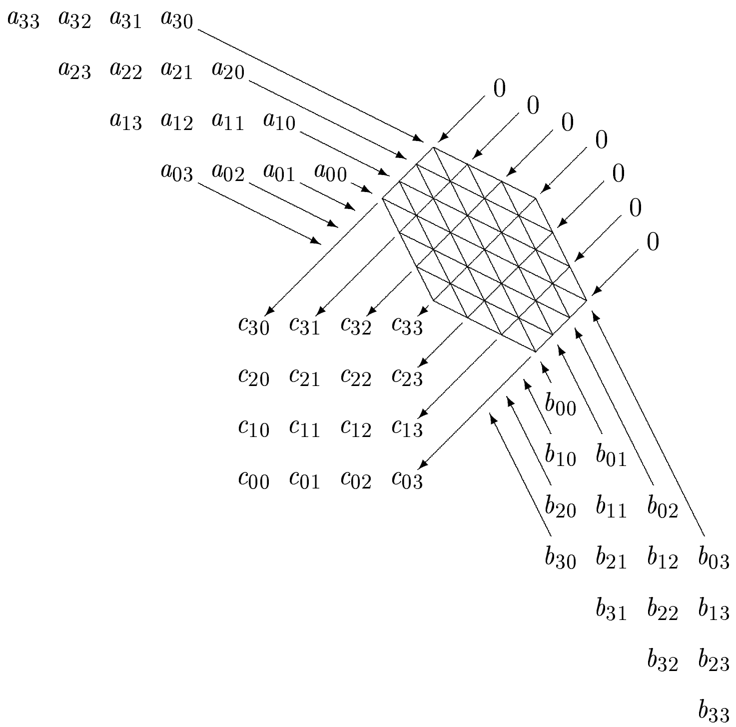


Figure 10.6: Multiplying matrices on a hexagonal array.

In this array the matrices A and B each take n steps to travel through the aray to their initial positions (with a_{00} and b_{00} in the centre of the array). Moreover, after the last computation is complete it must travel from the centre of the array and will be output from the edge of the array on the nth step of its travel. The array therefore multiplies the two $n \times n$ matrices using $3n^2 - 3n + 1$ processors in $5n - 3$ steps.

11

A Method for Refinement of Atomicity

Back[1] has formulated a method for refining the atomicity of a parallel program and shown this to be sound with respect to a total-correctness model of terminating action systems (see section 1.3). This chapter shows that a similar theorem is true for reactive action systems.

11.1 Definitions

The action A is said to *commute with* action B if $A;\ B$ is refined by $B;\ A$—that is, $B;\ A \subseteq A;\ B$ in the relational view of actions (see section 7.1). Let A and B be two actions and $B = h \rightarrow T$. We say that A *cannot disable* B if no state satisfying h is related by A to a state not satisfying h. We say that A *cannot enable* B if no state satisfying $\neg h$ is related by A to a state satisfying h. Theorem 1 of [1] gives sufficient conditions for two actions to commute: if $A = g \rightarrow S$ and $B = h \rightarrow T$ then A commutes with B if (i) A cannot disable B, (ii) B cannot enable A and (iii) $\{g \wedge h\}^{\top};\ T;\ S \subseteq S;\ T$.

We say that A *cannot be followed by* B if A does not relate any state to a state satisfying the guard of B. We say that A and B *exclude* each other if their guards are disjoint.

11.2 A Theorem

Let A be the action system

$$\textbf{var}\ x \bullet S_0;\ \textbf{do}\ \underset{i \in 1..m+1}{[\!]}\ A_i\ \textbf{od}$$

which always terminates normally, and B be the action system

$$\textbf{var}\ y \bullet T_0;\ \textbf{do}\ g \rightarrow A\ [\!]\ \Big(\underset{j \in 1..k+1}{[\!]}\ B_j\Big)\ \textbf{od}$$

which never stops abnormally whenever its rely condition is met (that is, it does not execute an action which waits forever or causes the system to be aborted). Let $A_0 = g \to x : [S_0]$ (where $x : [S_0]$ is an instruction which modifies x in such a way as to establish S_0, as in [17]) and define B' to be the action system

$$\mathbf{var}\ y, x \bullet T_0;\ \mathbf{do}\ \Big(\mathop{[]}_{i \in 0..m+1} A_i\Big)\ []\ \Big(\mathop{[]}_{j \in 1..k+1} B_j\Big)\ \mathbf{od}.$$

We refer to the actions A, $B_1, \ldots, B_k$ as the *old actions*, while the actions $A_0, \ldots, A_m$ are the *new actions*. Let gg stand for the disjunction of the guards of actions $A_1, \ldots, A_m$. Then, when the conditions below are satisfied, the refinement $B \sqsubseteq_R B'$ holds, where the relation R is the subset of the identity relation $\{(x, x) \mid \neg gg(x)\}$ (and where the equivalence relation $\sim$ required for the refinement is the identity).

The conditions for refinement of atomicity are as follows.

(1) An old action cannot disable any new action except A_0, nor is any new action except A_0 enabled in the initial states satisfying T_0.

(2) Action A_0 is excluded by action A_i for each i from $1 \mathinner{..} m+1$.

(3) The old actions, except A, that are not excluded by each A_i with $i \in 1 \mathinner{..} m+1$ are partitioned into two disjoint sets, the *left movers* $\{L_u \mid 1 \le u \le p\}$ and the *right movers* $\{R_v \mid 1 \le v \le q\}$ such that the following holds.

  (3.1) For each left mover L and each new action A_i, either A_i cannot be followed by L or L commutes with A_i.

  (3.2) For each right mover R and each new action (except A_0) or left mover C, either R cannot be followed by C or C commutes with R.

  (3.3) The action system

$$\mathbf{do}\ \Big(\mathop{[]}_{u \in 1..p+1} L_u\Big)\ []\ \Big(\mathop{[]}_{v \in 1..q+1} R_v\Big)\ \mathbf{od}$$

  is guaranteed to terminate normally for any initial state satisfying gg.

These conditions are the same as in [1] except that (3.3) has been strengthened slightly by adding the left-movers (this is to cover an extra case which is not required in the version of this theorem applied to terminating systems; this appears at the end of the following proof).

11.3 Proof of the Theorem

Since the conditions above are stronger than those of Back, Lemma 2 from [1] holds (the reader is referred to that document for the proof of the lemma, which will not be repeated here). This lemma states the following, in which the notation $\sigma[s]\tau$ denotes that the sequence of actions s may end in the state τ when executed from state σ.

Lemma: Let σ be some initial state such that B does not stop abnormally starting from σ. Let c be a finite prefix of a trace of B' executed starting from σ. For any τ such that $\sigma[c]\tau$, there exist sequences of actions c_1, c_2 and c_3 and states σ_1 and σ_2 such that $\sigma[c_1]\sigma_1[c_2]\sigma_2[c_3]\tau$ and the sequences are as follows.

(i) The sequence c_1 contains only old actions.

(ii) The sequence c_2 contains only new actions. If c_2 is not empty then its first action is A_0 and some action A_i, $i \neq 0$, is enabled in σ_2.

(iii) The sequence c_3 contains only right movers, and is empty if c_2 is empty.

Let C' be a computation of B' starting from a σ chosen as described above. The object of the proof is to find a computation C of B and an intermediate computation d of a mythical action system D (where $B \underset{\sim}{\sqsubseteq} D \sqsubseteq_R B'$) to show refinement. The conditions on d are as follows. Since $B \underset{\sim}{\sqsubseteq} D$, d must be a subsequence of C which is infinite if C is infinite. Since $D \sqsubseteq_R B'$, $R^{-1}(C')$ must equal d, since the equivalence relation $\sim$ is the identity and since $R^{-1}(x)$ is guaranteed to be a singleton set for any computation x. In other words, when the elements of C' that satisfy gg are removed, the result must be d.

Let t be a trace of B' inferred from the computation C'. Consider the above lemma applied to each prefix c of the trace t in turn. In each case let τ be the state in the computation C' which immediately follows the execution of the trace c. The lemma gives a sequence c_1; c_2; c_3 of actions corresponding to each c. Whenever τ does not satisfy gg (in other words, whenever τ is an element of d), the sequences c_2 and c_3 will be empty. This is because c_3 contains only old actions which cannot change the value of gg, and therefore σ_2 does not satisfy gg. By condition (ii), c_2 must therefore be empty and thus so is c_3. Hence, c and c_1 are both sequences of actions which end in state τ.

By construction of the proof of the lemma, the sequences c_1 obtained from increasing prefixes c of t are themselves increasing, and they reach a limit u which is a trace of B. Moreover, u can be executed to give a computation C in such a way that C contains every element of d, as demonstrated above since the sequences c_1 are prefixes of u. Therefore d is a subsequence of C. However, it must be verified that if C is finite then it is a complete computation of B and not just a prefix of one.

It is clear by construction that C is finite if and only if C' is finite; thus if C is finite then the computation C' is either a terminating computation or one which stopped abnormally. In the former case, the terminating state τ of C' does not satisfy gg and is therefore also the terminating state of C, showing that B has terminated normally. However, Back's proof of the refinement of atomicity theorem shows that the latter case is not possible: it shows that if C' was a computation that stopped abnormally then C must be also, and that contradicts an assumption made earlier about the starting state σ.

It remains to show that if C is infinite then so is d. This can be obtained from the fact that C' is infinite because we assume that the action system A always terminates in a finite time; in other words, it is always eventually true that gg is false, so that a state τ is always eventually added to the computation d. It is not possible for old actions to delay indefinitely the termination of A, because condition (3.3) prevents an infinite sequence of such actions.

11.4 Remarks

Simulations and applications of sequential refinement calculus provide useful techniques for developing action systems, but neither of these addresses the refinement of atomicity in a convenient manner. Refinement of atomicity is useful in distributed computing because it permits more parallelism; it splits a large action into several pieces which may require less synchronisation or fewer shared variables than the original. The use of this kind of refinement has been shown by examples [1],[3].

Condition (3) of the theorem is not as restrictive as it might at first seem. If it is only possible to class some of the old actions as left movers or right movers then the remaining actions may be made to exclude the new actions by introducing extra variables, called *sequencing variables*. For instance,

$$
\begin{array}{l}
\textbf{var } y, x \bullet T_0; \\
\textbf{do } g_0 \rightarrow S_0; \\
\qquad \textbf{do } \left([\!]_{i \in 1..m+1}\, g_i \rightarrow S_i \right) \textbf{ od} \\
\; [\!] \left([\!]_{j \in 1..k+1}\, h_j \rightarrow T_j \right) \\
\textbf{od}
\end{array}
$$

may be changed into

$$
\begin{array}{l}
\textbf{var } y, x;\ b : \mathbb{B} \bullet \neg b \wedge T_0; \\
\textbf{do } \neg b \wedge g_0 \rightarrow b := \neg b;\ S_0; \\
\qquad \textbf{do } \left([\!]_{i \in 1..m+1}\, b \wedge g_i \rightarrow S_i \right) \\
\qquad\quad [\!]\ \neg gg \wedge b \rightarrow b := \neg b \\
\qquad \textbf{od} \\
\; [\!] \left([\!]_{j \in 1..k+1}\, \neg b \wedge h_j \rightarrow T_j \right) \\
\textbf{od}
\end{array}
$$

which always passes all the conditions of the theorem. In this extreme case, however, applying the theorem does not result in any extra parallelism.

12

Discussion

In this chapter we draw some conclusions, make some comparison with related work, and suggest future developments.

12.1 Conclusions

In chapter 2 the semantics of an action system were defined in terms of the states that the system reaches during execution. Communication actions and probes were added to the notation, and the semantics were defined to include a history of communication events. The notation was expanded to allow extra constraints on the execution of an action system to be specified. This gives the added flexibility to write an action system obeying a constraint (such as fairness) which would otherwise be difficult or impossible to code.

Chapter 3 introduced a notation for specifying the desired behaviour of an action system while allowing the system to place requirements on the behaviour of its environment. This method of specification allows both terminating and reactive systems to be specified: a terminating system will be required to have all its computations of finite length with a last element satisfying certain properties, whereas a reactive system will be required to have all its computations infinite and to provide acceptable communications with the environment.

Also in chapter 3, several different refinement relations were introduced. The purpose of a refinement relation is to allow the developer to transform a system A into another system C, knowing that if $A \sqsubseteq C$ then C must satisfy the original specification from which A was developed. Each of the refinement relations was shown to possess this property. A rationale for allowing data refinements to change the inputs and outputs of a system was put forward, and this led to the relation $\sqsubseteq_R$. This relation was used in several examples in order to change the atomicity of the inputs and outputs of a system.

Chapter 5 introduced operators for composing two action systems in parallel. The first operator, the semantic parallel operator, was defined by specifying the computations of the process $P \mathbin{\hat{\|}} Q$ in such a way that the semantics of P and of Q are changed as little as possible when viewed individually. This parallel operator is able to compose any two action systems provided that no variables are shared between them. It has the property that when a system has been divided into two systems running in parallel, the individual systems may continue to be developed using operation refinements ($\sqsubseteq^{\natural}$) and correctness will be preserved. The second operator, the syntactic parallel operator, was defined by stating the syntax of $P \parallel Q$ in terms of the syntax of P and of Q. This operator was shown to be a special case of the semantic parallel operator, which is equivalent to it under the $\sqsubseteq^{E}$ refinement relation. Although syntactic parallel is not defined on all pairs of action systems, it has the advantage that for given P and Q, $P \parallel Q$ can be written explicitly as an action system, whereas $P \mathbin{\hat{\parallel}} Q$ cannot. The two operators together may be used to show that the parallel composition of two or more action systems is a valid implementation of some single action system.

In chapter 7, the technique of simulation was shown to be sound within the formalism described in this thesis. Downward and upward simulations were described, although only downward simulation was used in the various examples presented in later chapters. The use of special **null** actions was introduced in order to allow the developer to introduce internal actions. These actions produce a finite amount of repetition in the computations of the system being developed, in order to allow the new system to adjust its internal state.

Most of the remaining chapters demonstrated the use of action system techniques through a range of examples representing common problems in distributed computing, particularly in the areas of communication protocols and hardware design. Examples were described using specification triples and solved with simple, abstract, action systems. Each system was then developed into an efficient solution using a combination of software engineering refinements, downward simulations, and parallel composition.

In several places, actions and data refinements were described in relational terms. This turned out to be a convenient model in which to specify a simulation. It has been shown [9] that if predicate transformers are used to describe data refinements of terminating programs then there exists a single rule of simulation which describes such refinements in a manner which is both sound and complete. However, this result does not hold for systems with possibly infinite execution sequences: the upward simulation of section 7.5 needs to place constraints upon the action system and upon the refinement relation which are not required for downward simulation or for upward simulation involving terminating systems. By describing simulation in relational terms and separating the upward and downward simulations, we have avoided the problem of determining precisely which simulations require these constraints.

The parallel operators described in chapter 5 automatically hide the internal communications, which is something that does not happen in CSP [11], for example. This was found to be necessary in order to avoid exposing the effects of the stretching operator $\sharp$ upon the internal channels, which would otherwise change the atomicity of those actions which contain internal communications. One effect of this would have been to harm the relationship between the semantic and syntactic parallel operators. However, as mentioned in section 5.1, this does not result in loss of generality because the internal communications can be copied by one of the action systems into an external channel whenever the environment requires them to be visible.

12.2 Related Work

The structure of action systems described here is based on that introduced by Back and Kurki-Suonio[2]; however we have added communication events and probes similar to those of Martin[15]. Although Back's formalism allows parallel action systems to interact via shared variables, we have preferred to eliminate shared variables, allowing communication only via shared channels. Whilst this allows less flexibility, it has the advantage of allowing the developer to retain control over the changes to the variables of a system and also to express this control through the rely condition in the specification of the system.

The UNITY notation of Chandy and Misra [6] has a similar structure to that of action systems, but lacks explicit guards. A fairness constraint upon execution is required to ensure that a UNITY system progresses. A UNITY program never terminates in the usual sense, but it may reach a fixed point in which none of the actions can change the state. UNITY does not, however, have a parallel operator. Instead it has two program composition operators. One, the union operator, simply forms a system containing all the actions of the two component systems. The other, the superposition operator, modifies the actions of one system by placing extra instructions in parallel with them. This operator also allows the addition of actions, provided that they change only the variables which have been added to the system. All communication in UNITY is via shared variables. The technique for refinement adopted by Chandy and Misra is rather different from ours: rather than developing a system in steps they concentrate on the specification, adding more program variables and making it more detailed until it can easily be implemented.

Our addition of an extra liveness condition to an action system is somewhat similar to the 'supplementary property' of a specification in Abadi and Lamport[13]. This is a collection of computations which restricts the possible computations of a state machine which resembles an action system in its behaviour. Abadi and Lamport define a term 'machine closed' which means that the supplementary property does not specify any safety condition which is not already specified by the state machine. This is equivalent to the constraints which were placed on the added liveness condition in section 2.2.

Our simulations are similar to ones which are common in the literature; however it is worth noting that our downward simulation is very similar to the result which can be obtained by translating Butler's 'hiding refinement rule' [5] into a relational model. Also, the conditions of image finiteness and of finite nondeterminism which appear in the conditions for our upward simulation are very similar to conditions which appear in a paper on simulation by Lynch and Vaandrager[14]. This paper proves the properties of upward and downward simulation with respect to trace-refinement of automata which resemble action systems in their behaviour, and shows that upward simulations are only sound either if only the finite traces of the system are considered or if the simulation relation (which corresponds to our total relation R^{-1}) is image-finite.

The version of the sliding-window protocol presented in chapter 8 was described by van de Snepscheut in an unpublished manuscript [19]. He develops this protocol in a stepwise manner similar to ours, but in a less formal context. His development ends with the transformation of each of several actions of the form $ga \wedge gb \to x := e$ into communicating pairs of the form $ga \to c!e$ and $gb \wedge \overline{c} \to c?x$; this is done informally since a parallel operator is not available. Van de Snepscheut also ends by specialising his system to gain an implementation of the alternating-bit protocol. Our 'single-element acknowledgement' variation was based upon a description of the sliding-window protocol by Jeff Sanders[18].

The stack process of chapter 9 was suggested by Jeff Sanders, who gave a CSP description of the stack leading to a circuit diagram similar to ours in his lecture notes on hardware. The example originally derives from Mead and Conway[16]. It has previously been difficult to derive formally because of the event refinement involved.

12.3 Future Work

Chapter 3 contains six different kinds of refinement: $\sqsubseteq$, $\sqsubseteq^E$, $\underset{\sim}{\sqsubset}$, $\sqsubseteq_R$, $\sqsubseteq^E_R$ and $\underset{\sim}{\sqsubset}_R$. Several of these were shown to be necessary in the model; in particular, the refinements allowing parallelism (namely $\sqsubseteq^E$ and $\sqsubseteq^E_R$) are necessary for the assertion that $P \parallel Q$ is refined by $P \mathbin{\hat{\parallel}} Q$. This in turn is necessary if a system A is to be decomposed into two systems P and Q running in parallel, each of which are to be developed further. This is because the $\parallel$ operator is not monotonic with respect to refinement, whereas the $\hat{\parallel}$ operator is. Unfortunately, the existence of these various kinds of refinement sometimes tends to over-complicate matters—particularly when several of them are used in a single development. Section 3.5 describes the results when certain combinations of refinement are used in sequence, but one goal of future research must be to modify the model in such a way that fewer refinement symbols are needed and that their combinations are easy to calculate.

We have shown that the semantic parallel operator is monotonic with respect to the $\sqsubseteq^\natural$ refinement relation. It would also be advantageous that this operator should be monotonic with respect to data refinement. Unfortunately, the verification and proof of this property was not forthcoming and remains for future research.

It is known that for terminating programs the combination of downward and upward simulations is complete: every correct data refinement can be written as a pair of simulations applied in sequence. It has further been shown [9] that if predicate transformers are used to represent the data refinement then a single simulation is sufficient to model all data refinements. For our simulations, which cover reactive as well as terminating action systems, it is not known whether completeness can be attained—although Lynch and Vaandrager[14] and Butler[5] have given completeness results for models based on trace-refinement.

We have not investigated fully the rule for refinement of atomicity presented in chapter 11, although the examples presented in the literature suggest that this technique may be of considerable use. It could, for example, be used to split a sequential composition $P;\ Q$ by treating P as the initialisation S_0 and T as a loop **do** B_1 **od** whose body is executed precisely once.

Index

This index lists notation and defined terms used in the thesis. The section and page numbers given refer to the definition or first use of the notation.

Bibliography

[1] R. J. R. Back. A Method for Refining Atomicity in Parallel Programs. *Lecture Notes in Computing Science 366*, 1989.

[2] R. J. R. Back, R. Kurki-Suonio. Decentralization of process nets with centralized control. *2nd Annual Symposium on Principles of distributed Computing*, Montreal 1983.

[3] R. J. R. Back, K. Sere. Stepwise Refinement of Action Systems. *Science of Computer Programming 13*, 1989–90.

[4] R. G. Busacker, T. L. Saaty. *Finite Graphs and Networks: an introduction with applications*. McGraw-Hill, 1965.

[5] M. J. Butler. *A CSP Approach to Action Systems*. D. Phil. thesis, Oxford University, 1992.

[6] K. M. Chandy, J. Misra. *Parallel Program Design—a foundation*. Addison-Wesley, 1988.

[7] W. J. Dally, C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers, C-36(5)*, 1987.

[8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[9] P. H. B. Gardiner. On the Completeness of Data Refinement. Oxford University Computing Laboratory, January 2, 1990.

[10] J. F. He, C. A. R. Hoare, J. W. Sanders. Data Refinement Refined. *Lecture Notes in Computing Science 213*, 1986.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[12] C. R. Jesshope, P. R. Miller, J. T. Yantchev. High Performance Communications in Processor Networks. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, 1989.

[13] L. Lamport, M. Abadi. The Existence of Refinement Mappings. Digital Systems Research Center Report 29, 1988.

[14] N. Lynch, F. Vaandrager. Forward and Backward Simulations for Timing-Based Systems. *Lecture Notes in Computing Science 600*, 1991.

[15] A. J. Martin. The Probe: an Addition to Communication Primitives. *Information Processing Letters 20,3*, 1985.

[16] C. Mead, L. Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.

[17] C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.

[18] K. Paliwoda, J. W. Sanders. Incremental specification of the sliding-window protocol. *Distributed Computing (1991) 5*.

[19] J. L. A. van de Snepscheut. *The sliding-window protocol revisited*. Unpublished manuscript JAN 173. California Institute of Technology, 1992.