Stepwise Refinement of Communicating Systems

Michael J. Butler

Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, United Kingdom,
M.J.Butler@ecs.soton.ac.uk

The action system formalism [6] is a state-based approach to distributed computing. In this paper, it is shown how the action system formalism may be used to describe systems that communicate with their environment through synchronised value-passing. Definitions and rules are presented for refining and decomposing such action systems into distributed implementations in which internal communication is also based on synchronised value-passing. An important feature of the composition rule is that parallel components of a distributed system may be refined independently of the rest of the system. Specification and refinement is similar to the refinement calculus approach [4,27,29]. The theoretical basis for communication and distribution is Hoare's CSP [17]. Use of the refinement and decomposition rules is illustrated by the design of an unordered buffer, and then of a distributed message-passing system.

1 Introduction

The action system formalism, introduced by Back and Kurki-Suonio [6], is a state-based approach to distributed computing. A set of actions share some state variables on which they may act. An action system may be decomposed into a set of parallel sub-systems for implementation in a distributed fashion. An action system is decomposed by partitioning the actions, and interaction between parallel action systems is by shared variables. The refinement calculus approach for sequential programs of Back [4], Morgan [27], and Morris [29] may be used in the stepwise refinement of action systems [5]. Refinement is based on preservation of total correctness in the case of terminating action systems, and on state-trace refinement in the case of non-terminating, or reactive, action systems.

In this paper, a somewhat different approach to action systems is taken. Whereas Back takes a state-based approach and views the observable behaviour of an action system in terms of evolution of its state, we take an

Preprint submitted to Elsevier Preprint *10 April 1996*

event-based approach with the view that only the occurrence of actions is observable. We present rules for action system refinement that is consistent with this view, and describe parallel composition in which interaction is synchronous and based on shared actions rather than shared variables. The basis for our approach is Hoare's CSP [17].

A case study on the design of a distributed message-passing system is included to illustrate that our approach easily scales up to larger examples. The case study also shows that the mathematical reasoning used when applying the approach in practice is similar to that used in the refinement calculus approach for sequential programs, and that the introduction of parallelism based on synchronous value-passing places very little extra complexity on the proof obligations as compared with reasoning about sequential programs.

The paper is organised as follows: firstly, actions and action systems are described. In Section 4, the CSP view of action systems is outlined. Section 5 describes how actions may be parameterised to represent value communication. Refinement of actions is described in Section 6, and applied to action systems in Section 7. Section 8 introduces the notion of internal actions which is used to model hidden communication between sub-systems. In Section 9, parallel composition based on synchronisation over shared actions is introduced. This is extended in Section 10 to allow value-passing on synchronisation. In Section 11, the approach is used to design a distributed message-passing system.

2 Actions

An action is any statement in an extended version of Dijkstra's guarded command language [13]. Statements in the guarded command language are defined using *weakest precondition* formulae: for statement S and postcondition R, the formula $wp(S, R)$ (weakest precondition of S w.r.t. R) characterises those initial states from which action S is guaranteed to terminate in a state satisfying R. Dijkstra has given wp definitions of assignment ($x := E$), sequential composition ($S;\ T$), conditional statements (**if** ... **fi**), and iteration (**do** ... **od**). For example:

$$wp(S;\ T, R) \ \hat{=}\ wp(S, wp(T, R)).$$

Along with these, we use *specification statements*, *naked guarded commands*, and the *demonic choice* operator.

The specification statement $x : [post]$ assigns some value to x satisfying $post$. Also, $post$ may refer to the initial value of x as x_0. The wp definition of $x : [post]$

is as follows [25] (let $Q[x \backslash E]$ represent Q with all free occurrences of x replaced by E):

Definition 1 $wp(x : [post], R) \ \hat{=} \ (\forall x \bullet post \Rightarrow R)[x_0 \backslash x]$.

A special case of the specification statement is *generalised assignment*, written $x : \odot E$, where $\odot$ is some binary relation such as $=, <, \in$, and E is some expression possibly containing x. The assignment $x : \odot E$ is short for $x : [x \odot E[x \backslash x_0]]$.

For predicate G and statement S, the naked guarded command $G \rightarrow S$ is defined by:

Definition 2 $wp(G \rightarrow S, R) \ \hat{=} \ G \Rightarrow wp(S, R)$.

For statement S, $wp(S, false)$ represents those initial states in which S is guaranteed to establish any postcondition, i.e., behave miraculously. For example, the statement $false \rightarrow S$ is miraculous in any initial state:

$$
\begin{aligned}
& wp(false \rightarrow S, false) \\
= \ & false \Rightarrow wp(S, false) && \textit{Definition 2} \\
= \ & true.
\end{aligned}
$$

We take the view that a statement is enabled only in those initial states in which it behaves non-miraculously. The condition in which a statement S is enabled is called its *guard*, written $gd(S)$, and is defined as follows:

Definition 3 $gd(S) \ \hat{=} \ \neg \ wp(S, false)$.

For example, it can be shown that $gd(G \rightarrow x := E) = G$, provided E is well-defined.

The demonic choice of a set of actions $S_i, i \in I$, is written $([] \ i \in I \bullet S_i)$ and is defined by:

Definition 4 $wp(([] \ i \in I \bullet S_i), R) \ \hat{=} \ (\forall i \in I \bullet wp(S_i, R))$.

3 Action Systems

A basic action system $P = (A, v, P_\iota, P_A)$ consists of an alphabet of labels A, a state variable v (which may be a vector of variables), a set of labelled actions (statements) $P_A = \{P_\alpha \mid \alpha \in A\}$, and an initialisation action P_ι (P_ι is always enabled).

system $M1$

  var $g : \mathbb{B}$

  initially $g := false$

  action $a \bullet \quad g = false \rightarrow g := true$

  action $b \bullet \quad g = true \rightarrow g := false$

end

Fig. 1. Basic action system.

An action system proceeds by firstly executing the initialisation. Then, repeatedly, an enabled action is selected from the set P_A and executed. An action system deadlocks if no action is enabled, and diverges (behaves chaotically) whenever some action aborts.

When specifying an example action system, we write '**initially** S' for its initialisation, and '**action** $\alpha \bullet \quad S$' for the action labelled α.

Figure 1 gives an example of an action system. $M1$ has two labelled actions, a and b, and a single state variable g. Initially g is set to *false* so that only the a action is enabled. When the a action is executed, g is set to $true$, and only the b action is enabled. Execution of the b-action then results in a being enabled again and so on. Thus $M1$ describes a system that alternatively engages in an a-action then a b-action forever.

4 CSP and Action Systems

In CSP [17], a process interacts with its environment by synchronously engaging in atomic events. The set of events in which a process $\mathcal{P}$ engages is called its alphabet, written $alphabet(\mathcal{P})$. The behaviour of a process is defined in terms of the temporal ordering of events, and may be described using the CSP algebraic notation. An example of a basic process is the process $STOP$ that refuses to engage in any event. Sequencing of events is described by the prefix operator ($\rightarrow$): the expression $a \rightarrow \mathcal{P}$ describes the process that engages in the event a and then behaves as process $\mathcal{P}$. External choice of behaviour is described by the choice operator ($[]$): $\mathcal{P} \mathbin{[]} \mathcal{Q}$ represents the process that offers the choice to the environment between behaving as $\mathcal{P}$ or as $\mathcal{Q}$. CSP also has a separate nondeterministic-choice operator ($\sqcap$): $\mathcal{P} \sqcap \mathcal{Q}$ represents the process that internally chooses between behaving as $\mathcal{P}$ or as $\mathcal{Q}$, leaving the environment with no control over that choice. A recursive definition is written $(\mu X \bullet F(X))$, where $F(X)$ is some expression containing X.

The semantics of a CSP process $\mathcal{P}$, with alphabet A, is modelled by a set of failures, $\mathcal{F}[\![\mathcal{P}]\!]$, and a set of divergences, $\mathcal{D}[\![\mathcal{P}]\!]$. A failure is a pair of the form (s, X), where $s \in A^*$ (the set of finite sequences of elements of A) is an event-trace and $X \subseteq A$ is a refusal set. If (s, X) is in $\mathcal{F}[\![\mathcal{P}]\!]$, then after engaging in the sequence of events s, a process may refuse all events in X. A divergence is simply a finite event-trace and $s \in \mathcal{D}[\![\mathcal{P}]\!]$ means that, after engaging in s, process $\mathcal{P}$ may diverge (behave chaotically).

We take the view that an action system engages in actions jointly with its environment. The environment of an action system can only observe the actions and cannot observe the state of the system. In this way, the behaviour of an action system is similar to that of a CSP process. For example, the action system *M1* of Figure 1 behaves in the same way as the CSP process $\mathcal{M}1$ described algebraically as follows:

$$\mathcal{M}1 \;\hat{=}\; (\mu\, X \bullet a \to b \to X).$$

Morgan [26] has described a correspondence between action systems and CSP by defining the failures and divergences of an action system using weakest-precondition formulae. For action S and postcondition ϕ, the formula $\overline{wp}(S, \phi)$ (conjugate weakest precondition of S w.r.t. ϕ) characterises those initial states from which action S could *possibly* establish ϕ or may fail to terminate. It is defined as follows:

Definition 5 $\overline{wp}(S, \phi) \;\;\hat{=}\;\; \neg\, wp(S, \neg\, \phi)$.

For any set of labels $X \subseteq A$, we write $gd(P_X)$ for the *disjunction* of the guards of actions labelled from X. For any trace $t \in A^*$, we write P_t for the sequential composition of the actions of P labelled from t, with $P_{\langle\rangle} = \mathbf{skip}$. We require that the initialisation $P_\imath$ is unguarded ($gd(P_\imath) = true$). The failures-divergences semantics of action system P are given by the following definition [26]:

Definition 6 *For action system* $P \;=\; (A,\; v,\; P_\imath,\; P_A)$*, the failures of* P*, written* $\mathcal{F}\{\![P]\!\}$*, are those* $(s, X) \in (A^* \times \mathbb{P}\, A)$*, satisfying:*

$$\overline{wp}(P_{\langle\imath\rangle^\frown s}, \neg\, gd(P_X)),$$

and the divergences of P*, written* $\mathcal{D}\{\![P]\!\}$*, are those* $s \in A^*$ *satisfying:*

$$\overline{wp}(P_{\langle\imath\rangle^\frown s}, false).$$

Thus (s, X) is a failure of P if, after initialisation, P could possibly engage in the action trace s and then refuse all actions X. Trace s is a divergence of P if $P_{\langle\imath\rangle^\frown s}$ is not guaranteed to terminate.

It can be shown, using Definition 6, that action system *M1* of Figure 1 has the same failures-divergences semantics as the CSP process $\mathcal{M}1$ above.

The CSP failures-divergences model does not deal properly with unbounded nondeterminism [33]. In [8,10], Definition 6 is extended to deal with unbounded nondeterminism by including the infinite event-traces of an action system in its semantics. Details are omitted here.

5 Value-Passing Action Systems

In this section, we extend action systems to deal with value-passing. First we look at value-passing in CSP. Hoare [17] introduced the notion of a *channel* for use when describing CSP processes that communicate values with their environment. A channel named c is represented by a set of events of the form $c.i$. Occurrence of an event $c.i$ represents communication of the value i over the channel c. The set of all values that a process $\mathcal{P}$ can communicate over a channel c is defined as follows:

$$values_{\mathcal{P}}(c) \;\hat{=}\; \{\, i \mid c.i \in alphabet(\mathcal{P}) \,\}.$$

Hoare distinguishes syntactically between two forms of channel communication. A process $\mathcal{P}$ is said to be prepared to accept an *input* on a channel c when the process offers the environment the choice between each event of the form $c.i$ in $alphabet(\mathcal{P})$. The notation $c?x$ is used to describe this form of communication, and is defined by

$$(c?x \to \mathcal{P}_x) \;\hat{=}\; ([\!]\, i \in values_{\mathcal{P}}(c) \bullet c.i \to \mathcal{P}_i).$$

So $(c?x \to \mathcal{P}_x)$ describes a process that accepts any value x on channel c, and then behaves as process $\mathcal{P}_x$. Note that x is bound by the parentheses.

A process is said to be prepared to *output* a value on a channel c when it internally chooses which particular event $c.i$ to offer the environment. The notation $c!e$ is used to describe this form of communication, and is defined by

$$(c!e \to \mathcal{P}) \;\hat{=}\; (c.e \to \mathcal{P}).$$

So $(c!e \to \mathcal{P})$ describes a process that outputs the value e (where e is an expression) on channel c, and then behaves as process $\mathcal{P}$.

As an example, we have the following CSP process $\mathcal{COPY}$, that continually accepts values on channel *left* and then outputs them on channel *right*:

$$\mathcal{COPY} \mathrel{\hat{=}} (\mu\, Y \bullet \mathit{left}?x \rightarrow \mathit{right}!x \rightarrow Y).$$

We now extend action systems so that they may pass values either to or from the environment when they engage in actions in the manner of CSP channels. A channel will be represented by a parameterised action, where the parameter represents the value communicated on that channel. A parameterised action will represent either an input channel or an output channel. Each parameterised action will be given a type, representing the range of values that may be communicated over that channel.

A *value-passing* action system has the form

$$P = (A,\ v,\ P_\imath,\ P_A,\ \mathit{dir},\ \mathit{type}).$$

Here A is a set of channel names, v is the state variable, $P_\imath$ is the initialisation, and $P_A = \{P_\alpha \mid \alpha \in A\}$ is a set of actions labelled by channel names. The component dir is a total function from A to the set $\{\mathit{in}, \mathit{out}\}$. If $\mathit{dir}(\alpha) = \mathit{in}$, then P_α is a $(v, x?)$-action [1] that doesn't change variable $x?$, and we say that P_α is an input action and $x?$ is its input parameter with input type $\mathit{type}(\alpha)$. If $\mathit{dir}(\alpha) = \mathit{out}$, then P_α is a $(v, y!)$-action that doesn't read the initial value of $y!$, and we say that P_α is an output action and $y!$ is its output parameter with output type $\mathit{type}(\alpha)$. Both $x?$ and $y!$ must be distinct from the state-variable v.

A value-passing action system can accept an input on a channel $\alpha \in A$, where $\mathit{dir}(\alpha) = \mathit{in}$, whenever action P_α is enabled. The initial value of variable $x?$ of P_α represents the input value on channel α. A value-passing action system is prepared to output a value on a channel $\alpha \in A$, where $\mathit{dir}(\alpha) = \mathit{out}$, whenever action P_α is enabled. The value assigned to variable $y!$ by P_α represents the value to be output on channel α.

We specify an input action with notation of the form:

$$\textbf{action}\ \alpha\ \textbf{in}\ x? : T \bullet \qquad S,$$

where α is the channel label, $x?$ is the input variable, T is the channel type, and S is a statement in the extended guarded command language. Similarly, output actions are specified with notation of the form:

$$\textbf{action}\ \alpha\ \textbf{out}\ y! : T \bullet \qquad S.$$

[1] We say that S is a v-action when v is the list of variables that S reads from and assigns to.

system $Buffer^2$
  var $s : \text{seq } T$
  initially $s := \langle\rangle$
  action *left* **in** $x? : T$ $\bullet$ $\quad s := s^\frown\langle x?\rangle$
  action *right* **out** $y! : T$ $\bullet$ $\quad s \neq \langle\rangle \rightarrow y!, s := head(s), tail(s)$
end

Fig. 2. Ordered Buffer.

Figure 2 specifies an action system representing an ordered buffer. It is always ready to accept values of type T on the *left* channel, and to output on the *right* channel a value that has been input but not yet output. Values are output in the order in which they are input.

Definition 6 is extended to deal with value-passing in [8]. Details are omitted here. An important feature of the failures-divergences semantics of value-passing action systems as defined in [8] is that if the choice of output value on an output channel is nondeterministic, then that choice is made internally by the action system rather than by the external environment. In contrast, with an input channel, the choice between a range of input values is offered to the external environment. This is why we distinguish input and output channels.

6 Refinement of Actions

A statement T is said to refine S if T is at least as deterministic as S, and we write $S \preceq T$ for statement S is (*algorithmically*) refined by statement T. For predicates ϕ and ψ, we write $\phi \Rrightarrow \psi$ if any state satisfying ϕ also satisfies ψ. We then define $S \preceq T$ as follows:

Definition 7 *$S \preceq T$ if for each predicate ϕ,* $\quad wp(S, \phi) \;\Rrightarrow\; wp(T, \phi).$

For example, it can be shown that $x : [\ x < x_0\ ] \quad \preceq \quad x := x - 1.$

In the refinement calculus approach [4,27,30], program development involves stepwise refinement of an abstract program into an executable program by application of a series of correctness preserving transformations. For example, the following rule is derived from Definitions 1 and 7:

Rule 8 *If $post \Lleftarrow post'$, then* $\quad v : [\ post\ ] \;\preceq\; v : [\ post'\ ].$

Data refinement involves replacing abstract variables with concrete variables that are more easily implemented [4,16,28,30]. An abstraction invariant AI relating the abstract variables a and concrete variables c is used to replace abstract statements with concrete statements. If S is an (a, z)-statement, T is a (c, z)-statement, and AI is an abstraction invariant then we write $S \preceq_{AI} T$ for "S is data-refined by T under abstraction invariant AI" [4,28,30]:

Definition 9 *$S \preceq_{AI} T$ if for each ϕ independent of concrete variable c,*

$$(\exists\, a \bullet AI \wedge wp(S, \phi)) \;\Rrightarrow\; wp(T, (\exists\, a \bullet AI \wedge \phi)).$$

Often the abstraction invariant takes the form

$$AI \;\equiv\; I \wedge (a = F),$$

where both I and F are independent of abstract variable a. Such an abstraction invariant is said to be *functional*, since for any concrete value, there is at most one corresponding abstract value. Given a functional abstraction invariant, we may calculate the data-refinement of a specification statement $a, z : [\ post\ ]$ using the following rule, which is derived from Definitions 1 and 9:

Rule 10 *For $AI \;\equiv\; (a = F) \wedge I$, let $I_0 \;\hat{=}\; I[c, z \backslash c_0, z_0]$ and $F_0 \;\hat{=}\; F[c, z \backslash c_0, z_0]$. Then, for post independent of c, c_0,*

$$a, z : [\ post\ ] \;\;\preceq_{AI}\;\; c, z : [\ I_0 \wedge I \wedge post[a, a_0 \backslash F, F_0]\ ].$$

More general data-refinement calculators may be found in [28,30].

7 Refinement of Action Systems

In CSP, the process $\mathcal{Q}$ is a refinement of process $\mathcal{P}$, denoted $\mathcal{P} \sqsubseteq \mathcal{Q}$, if any behaviour exhibited by $\mathcal{Q}$ may be exhibited by $\mathcal{P}$: for CSP processes $\mathcal{P}$, $\mathcal{Q}$, both with alphabet A, $\mathcal{P} \sqsubseteq \mathcal{Q}$ if

$$\mathcal{F}[\![\mathcal{P}]\!] \supseteq \mathcal{F}[\![\mathcal{Q}]\!] \text{ and } \mathcal{D}[\![\mathcal{P}]\!] \supseteq \mathcal{D}[\![\mathcal{Q}]\!].$$

Similarly, we say that an action system P is refined by an action system Q, denoted $P \sqsubseteq Q$, if

$$\mathcal{F}\{\![P]\!\} \supseteq \mathcal{F}\{\![Q]\!\} \text{ and } \mathcal{D}\{\![P]\!\} \supseteq \mathcal{D}\{\![Q]\!\}.$$

It is easy to see that $\sqsubseteq$ is a transitive ordering.

Simulation is a proof technique for checking action system refinement. A simulation is a relation between action systems P and Q with the same alphabet but possibly different state-spaces:

$$P = (A,\ v,\ P_\iota,\ P_A,\ dir,\ type) \text{ and } Q = (A,\ w,\ Q_\iota,\ Q_A,\ dir,\ type).$$

Note that channel directions and types are the same in both P and Q. Assume that $v = (a, z)$ and $w = (c, z)$, so that a is the abstract variable, c is the concrete variable and z is common to both P and Q. Simulation relies on data refinement between corresponding actions of P and Q. To deal with initialisation statements we, introduce a special case of data-refinement ($\preceq'_{AI}$):

Definition 11 $S \preceq'_{AI} T$ *if for each ϕ independent of concrete variable c,*

$$wp(S, \phi) \ \Rrightarrow\ wp(T, (\exists\, a \bullet AI \land \phi)).$$

Simulation is defined as follows:

Definition 12 *A simulation, with abstraction invariant AI, is a relation on action systems, denoted $\sqsubseteq_{AI}$, such that for action systems*

$$P = (A,\ v,\ P_\iota,\ P_A,\ dir,\ type) \text{ and } Q = (A,\ w,\ Q_\iota,\ Q_A,\ dir,\ type)$$

$P \sqsubseteq_{AI} Q$ if each of the following conditions hold:

(i) $P_\iota \preceq'_{AI} Q_\iota$
(ii) $P_\alpha \preceq_{AI} Q_\alpha$, *each* $\alpha \in A$
(iii) $(\exists\, a \bullet AI \land gd(P_\alpha)) \ \Rrightarrow\ gd(Q_\alpha)$, *each* $\alpha \in A$.

Conditions 1 and 2 ensure that each action of Q is a refinement of its counterpart in P, and are referred to as data-refinement conditions. Condition 3 ensures that Q may only refuse an action when P may refuse it, and is referred to as a progress condition.

In [8], it is shown for value-passing action systems P, Q, that if $P \sqsubseteq_{AI} Q$ for some AI, then $P \sqsubseteq Q$, i.e., simulation is sound. Soundness of simulation for action systems without value-passing has already been shown in [37].

Internal and External Nondeterminism

An important feature of Definition 12 is that it distinguishes internal and external nondeterminism. Consider the action systems *K1* and *K2* of Figure 3.

system *K1*
  initially **skip**
  action *tea* • **skip**
  action *coffee* • **skip**
end

system *K2*
  var $g : \mathbb{B}$
  initially $g :\in \{true, false\}$
  action *tea* • $g \rightarrow g :\in \{true, false\}$
  action *coffee* • $\neg\, g \rightarrow g :\in \{true, false\}$
end

Fig. 3. External and internal choice.

K1 offers the environment the choice between a *tea*-event or a *coffee*-event, whereas *K2* chooses internally whether to offer a *tea*-event or a *coffee*-event. In CSP refinement, external nondeterminism must be preserved, though internal nondeterminism may be reduced. So, it is not the case that $K1 \sqsubseteq K2$, and Definition 12 fails (in particular, the progress condition fails). However, using the simulation rule of Back [5] which groups all the actions of an action system into a single action, it can be shown that *K1* is refined by *K2* (take $AI \equiv g \in \{true, false\}$). This is because Back's rule ensures state-trace refinement and the state-traces model of action systems does not distinguish choice of action (external choice) from nondeterministic effect of an action (internal choice).

With Definition 12, reducing the nondeterminism of individual actions may result in reducing internal nondeterminism. This includes the possibility of reducing the choice between a range of output values on an output channel. The progress condition of Definition 12 (Condition 3) ensures that external choice is preserved, including the choice between a range of input values on an input channel.

Example: Unordered Buffer

We specify and refine a buffer that does not guarantee to output values in the order in which they are input. An unordered buffer will be described by an action system that has a *bag* of values as its state variable. A bag is a collection of elements that may have multiple occurrences of any element. We write bag T for the set of finite bags of type T. Bags will be enumerated between bag brackets $\prec$ and $\succ$. *Addition* of bags b, c, is written $b + c$, while *subtraction* is written $b - c$.

The action system $UBuffer^1$ of Figure 4 describes an unordered buffer that communicates values of type T. The initialisation statement of $UBuffer^1$ sets

system *UBuffer*^{1}
    var a : bag T
    initially $a : [\ a = \prec\succ\ ]$
    action *left* **in** $x? : T$ • $\quad a : [\ a = a_0 + \prec x?\succ\ ]$
    action *right* **out** $y! : T$ • $\quad a : [\ y! \in a_0 \wedge a = a_0 - \prec y!\succ\ ]$
end

system *Buffer*^{1}
    var s : seq T
    initially $s : [\ s = \langle\rangle\ ]$
    action *left* **in** $x? : T$ • $\quad s : [\ s = s_0 {}^\frown \langle x?\rangle\ ]$
    action *right* **out** $y! : T$ • $\quad s : [\ y! = head(s_0) \wedge s = tail(s_0)\ ]$
end

Fig. 4. Unordered buffer and its ordered implementation.

the bag to be empty. The input action *left* accepts input values of type T, adding them to the bag a. Provided a is non-empty, the output action *right* nondeterministically chooses some element from a and outputs it.

We assume that variable typing is an explicit part of each statement. So, for example, the initialisation of *UBuffer* is short for $a : [\ a \in \text{bag } T \wedge a = \prec\succ\ ]$, while the definition of $UBuffer^{1}_{left}$ is short for

$$\textbf{action } \mathit{left} \textbf{ in } x? : T \bullet \qquad a : \begin{bmatrix} a, a_0 \in \text{bag } T \wedge \\ x? \in T \wedge a = a_0 + \prec x?\succ \end{bmatrix}.$$

The action guards of *UBuffer*^{1} are implicit, but may be calculated explicitly using the following rule (derived from Definitions 1 and 3):

Rule 13 $gd(v : [\ post\ ]) \;\equiv\; (\exists\, v, y! \bullet post)[v_0 \backslash v]$.

Here $y!$ is only relevant if the statement describes the effect of an output action. Using this rule, we get:

$$\begin{aligned} gd(UBuffer^{1}_{left}) &\equiv a \in \text{bag } T \wedge x? \in T \\ gd(UBuffer^{1}_{right}) &\equiv a \in \text{bag } T \wedge a \neq \prec\succ. \end{aligned}$$

Recall that an initialisation statement should always be unguarded. This may be checked using the following rule (derived from Rule 13):

Rule 14 *Initialisation* $v : [\ post\ ]$ *is unguarded if* $(\exists\, v \bullet post) \equiv true$.

Figure 4 also describes an ordered buffer, *Buffer*^{1}, similar to that of Figure 2.

We use Definition 12 to show that

$$UBuffer^1 \sqsubseteq Buffer^1.$$

As an abstraction invariant, we use

$$AI \;\hat{=}\; s \in \text{seq}\, T \wedge a = bag(s),$$

where $bag(s)$ represents the bag of elements in sequence s. Rules 8 and 10 may be used to show that the data-refinement conditions of Definition 12 are satisfied. For example, we have

$$\begin{array}{lll}
UBuffer^1_{right} = & a : [\; y! \in a_0 \wedge a = a_0 - \prec y! \succ \;] & \\
\preceq_{AI} & s : \left[\begin{array}{l} s_0, s \in \text{seq}\, T \wedge \\ y! \in bag(s_0) \;\wedge\; bag(s) = bag(s_0) - \prec y! \succ \end{array} \right] & \textit{Rule 10} \\
\preceq & s : \left[\begin{array}{l} s_0, s \in \text{seq}\, T \wedge y! \in T \wedge \\ y! = head(s_0) \wedge s = tail(s_0) \end{array} \right] & \textit{Rule 8} \\
= & Buffer^1_{right}. &
\end{array}$$

It can also be shown that the progress condition of Definition 12 is satisfied. For example:

$$\begin{array}{ll}
& (\exists\, a \bullet AI \wedge gd(UBuffer^1_{right})) \\
\Rrightarrow & (\exists\, a \bullet AI \wedge a \in \text{bag}\; T \wedge a \neq \prec\succ) \\
\Rrightarrow & s \in \text{seq}\, T \wedge bag(s) \neq \prec\succ \qquad \textit{1-point Rule} \\
\Rrightarrow & s \in \text{seq}\, T \wedge s \neq \langle\rangle \\
\Rrightarrow & gd(Buffer^1_{right}).
\end{array}$$

It can also be shown that $Buffer^1$ is refined by $Buffer^2$ of Figure 2. Hence, by transitivity, we have $UBuffer^1 \sqsubseteq Buffer^2$.

8 Internal Actions

In this section, action systems are extended to include internal actions. An action system may also have a set of internal actions P_H:

$$P = (A,\ v,\ P_\imath,\ P_A,\ P_H,\ dir,\ type).$$

The action $([\!]\, h \in H \bullet P_h)$ represents the demonic choice of all the internal actions of P. In such an action system, any number of executions of $([\!]\, h \in H \bullet P_h)$ may occur in between each execution of a visible action P_α. If the action system reaches a state where $([\!]\, h \in H \bullet P_h)$ can be executed infinitely, then the action system diverges. Internal actions do not have input and output parameters.

Overview of Semantics

The failures-divergences semantics of action systems is extended in [8,10] to deal with internal actions. A nondeterministic iterate operator is introduced to the guarded-command language to represent the arbitrary execution of internal actions. The nondeterministic iteration of action S is written **it** S **ti**. This is similar to the DO-loop **do** S **od**, except that **it** S **ti** may nondeterministically terminate before S becomes disabled. Like **do** S **od**, **it** S **ti** aborts when S may execute forever.

The failures-divergences semantics of an action system P, with internal actions P_H, is given by simply converting P into an action system without internal actions: instead of having separate internal actions, the action **it** $([\!]\, h \in H \bullet P_h)$ **ti** is sequentially composed before and after each visible action P_α and after the initialisation $P_\imath$; this models the possibility of arbitrary iteration of the internal actions between the visible actions, as well as the occurrence of divergence when the internal actions may be executed forever.

Hiding Operator

Process events in CSP are internalised by the hiding operator: hiding of a set of events $C \subseteq alphabet(\mathcal{P})$ is written $\mathcal{P} \setminus C$. We introduce a corresponding operator for action systems: hiding of the set of actions labelled C in action system P results in the action system $P \setminus C$. Hiding in action systems is achieved simply by internalising the actions labelled C, as follows:

Definition 15 *For action system $P = (A,\ v,\ P_\imath,\ P_A,\ P_H,\ dir,\ type)$, action set $C \subseteq A$,*

$$\begin{aligned} P \setminus C \ \hat{=}\ & (A - C,\ v,\ P_\imath,\ P_{A-C}, \\ & P_H \cup \{\ (\mathbf{var}\ x?, y! \bullet P_c) \mid c \in C\ \},\ C \lhd dir,\ C \lhd type) \end{aligned}$$

where $x?$ is the input parameter used by input actions of P, and $y!$ is the output parameter used by output actions of P.

Here, $C \lhd f$ restricts the domain of function f to those elements not in set C, and **var** makes $x?$ and $y!$ local to each action to be hidden. The definition of **var** is as follows:

Definition 16 *For ϕ independent of x,*

$$wp((\mathbf{var}\ x \bullet S), \phi) \ \hat{=}\ (\forall\, x \bullet wp(S, \phi)).$$

Definition 15 corresponds to CSP hiding in the following sense. For CSP process $\mathcal{P}$, let $[\![\mathcal{P}]\!]$ be short for the failures-divergences semantics of $\mathcal{P}$. In [17], the semantics of $\mathcal{P} \setminus C$ is defined as a function, HIDE, of $[\![\mathcal{P}]\!]$ and C:

$$[\![\mathcal{P} \setminus C]\!] \ \hat{=}\ \mathsf{HIDE}([\![\mathcal{P}]\!], C).$$

For value-passing action system P, let $\{\![P]\!\}$ be short for the failures-divergences semantics of P. We then have the following theorem:

Theorem 17 $\{\![P \setminus C]\!\} \ = \ \mathsf{HIDE}(\{\![P]\!\}, C)$.

Proof of Theorem 17 is given in [8,10] and is based on using properties of **it** S **ti** to show that $\{\![P \setminus C]\!\}$ can be characterised in terms of $\{\![P]\!\}$ and C. Because of Theorem 17, the hiding operator for action systems inherits properties of the CSP hiding operator. Important such properties include commutativity and monotonicity w.r.t. refinement:

Rule 18 $(P \setminus C) \setminus D \ = \ (P \setminus D) \setminus C \ = \ P \setminus (C \cup D)$.

Rule 19 *If* $P \sqsubseteq Q$ *then* $P \setminus C \ \sqsubseteq\ Q \setminus C$.

Example

An example of an action system with internal actions is given in Figure 5. *UBuffer*^{2} represents an unordered buffer with an input channel *left* and an output channel *right*. However, instead of having a single bag as its state variable, *UBuffer*^{2} has two bags, b and c. The *left* action places input values in bag b, while the *right* action takes output values from bag c. Values are moved from b to c by the internal action *mid*, which is enabled as long as b is non-empty. Since b is finite, *mid* will eventually be disabled, so it cannot cause divergence.

system $UBuffer^2$

  var b, c : bag T

  initially $b, c : [\ b = c = \prec\succ\ ]$

  action *left* **in** $x? : T \bullet$ $b : [\ b = b_0 + \prec x?\succ\ ]$

  action *right* **out** $y! : T \bullet$ $c : [\ y! \in c_0 \wedge c = c_0 - \prec y!\succ\ ]$

  internal *mid* $\bullet$ $b, c : \left[\begin{array}{l}(\exists\, z \in b_0 \bullet b = b_0 - \prec z\succ \\ \qquad\qquad \wedge\ c = c_0 + \prec z\succ)\end{array}\right]$

end

Fig. 5. Unordered buffer with internal action.

Refinement

Intuitively it can be seen that $UBuffer^2$ behaves the same as $UBuffer^1$ of Figure 4. We shall introduce a proof rule that allows us to verify that $UBuffer^1 \sqsubseteq UBuffer^2$. This rule is a special form of simulation in which the concrete system has some internal actions, and the abstract system has no internal actions.

To ensure that the internal actions do not introduce divergence, a well-foundedness argument is used. A set WF, with irreflexive partial order $<$, is *well-founded* if each non-empty subset of WF contains a minimal element under $<$. For example, the natural numbers with the usual ordering, or the cartesian product of two or more well-founded sets with lexicographic ordering, all form well-founded sets. The well-foundedness argument requires the use of a well-founded set WF and a *variant*, which is an expression in the state-variables. The variant should always be an element of WF, and it should be decreased by each internal action of the concrete system.

The simulation rule is as follows (Q_G are the internal actions of Q):

Definition 20 *For value-passing action systems*

$$P = (A,\ v,\ P_\imath,\ P_A,\ \{\},\ dir,\ type)$$
$$Q = (A,\ w,\ Q_\imath,\ Q_A,\ Q_G,\ dir,\ type)$$

$P \sqsubseteq_{(AI, WF, E)} Q$ *if the following conditions hold, for abstraction invariant AI, well-founded set WF, and variant E,*

(i) $P_\imath \ \preceq'_{AI}\ Q_\imath$

(ii) $P_\alpha \ \preceq_{AI}\ Q_\alpha$, *each* $\alpha \in A$

(iii) **skip** $\preceq_{AI}\ Q_g$, *each* $g \in G$

(iv) $(\exists\, a \bullet AI) \ \Rrightarrow\ E \in WF$

(v) $(\exists\, a \bullet AI) \wedge E = e \ \Rrightarrow\ wp(Q_g, E < e)$, *each* $g \in G$

(vi) $(\exists\, a \bullet AI \wedge gd(P_\alpha)) \ \Rrightarrow\ gd(Q_\alpha) \vee (\exists\, g \in G \bullet gd(Q_g))$, *each* $\alpha \in A$.

Conditions 1, 2, and 3 are data-refinement conditions. Conditions 1 and 2 are the same as in Definition 12. Condition 3 ensures that each internal action of Q causes no change to the corresponding abstract state. Condition 4 ensures that the variant E is an element of WF, while Condition 5 ensures that the internal actions of Q always decrease E when executed. Together, Conditions 4 and 5 ensure that the internal actions of Q are eventually disabled and so cannot introduce divergence. Conditions 4 and 5 are referred to as non-divergence conditions. Condition 6 is a progress condition and ensures that, whenever an action of P is enabled, either the corresponding action of Q is enabled, or some internal action of Q is enabled. It is demonstrated in [8] that $P \sqsubseteq Q$ follows from $P \sqsubseteq_{(AI,WF,E)} Q$.

Note that, in certain cases, because of the monotonicity of the hiding operator, the hiding refinement rule can be used even if the abstract action system P has a non-empty set of internal actions: Definition 20 can always be applied if P and Q have the form

$$\begin{aligned} P &= (A,\ v,\ P_\imath,\ P_A,\ P_H,\ dir,\ type) \\ Q &= (A,\ w,\ Q_\imath,\ Q_A,\ Q_{H \cup H'},\ dir,\ type). \end{aligned}$$

Notice how each internal action of P has a correspondingly labelled internal action in Q. To show $P \sqsubseteq Q$, construct action systems P' and Q' by making the H actions visible as follows:

$$\begin{aligned} P' &\mathrel{\hat=} (A \cup H,\ v,\ P_\imath,\ P_{A \cup H},\ \{\},\ dir,\ type) \\ Q' &\mathrel{\hat=} (A \cup H,\ w,\ Q_\imath,\ Q_{A \cup H},\ Q_{H'},\ dir,\ type), \end{aligned}$$

Now, using Definition 20, show that $P' \sqsubseteq Q'$. Hence, by monotonicity, $P' \setminus H \sqsubseteq Q' \setminus H$, i.e., $P \sqsubseteq Q$.

Example

To show that $\mathit{UBuffer}^1 \sqsubseteq \mathit{UBuffer}^2$, we use the abstraction invariant

$$AI \mathrel{\hat=} c, b \in \text{bag } T \wedge a = b + c.$$

We use the size of bag b, written $\# b$, as a variant, with $\mathbb{N}$ as a well-founded set. The data-refinement conditions of Definition 20 are easily checked using refinement calculus rules. For example, Condition 3 is checked as follows:

$$
\begin{aligned}
\mathbf{skip} = \quad & a : [\ a = a_0\ ] \\
\preceq_{AI}\ & b, c : \left[\begin{array}{l} b_0, c_0, b, c \in \text{bag}\ T \wedge \\ b + c = b_0 + c_0 \end{array} \right] \\
\preceq \quad & b, c : \left[\begin{array}{l} b_0, c_0, b, c \in \text{bag}\ T \wedge \\ (\exists\, x \in b_0 \bullet b = b_0 - \prec x? \succ \wedge\ c = c_0 + \prec x? \succ) \end{array} \right] \\
\preceq \quad & \mathit{UBuffer}^2_{mid}.
\end{aligned}
$$

The non-divergence conditions are satisfied since $b \in \text{bag}\ T \Rrightarrow \#b \in \mathbb{N}$, and mid always decreases $\#b$ (which may be checked using Definition 1). Finally, the progress condition is easily checked; for example:

$$
\begin{aligned}
& (\exists\, a \bullet AI \wedge gd(\mathit{UBuffer}^1_{right})) \\
\Rrightarrow\ & (\exists\, a \bullet AI \wedge a \in \text{bag}\ T \wedge a \neq \prec\succ) \\
\Rrightarrow\ & b, c \in \text{bag}\ T \wedge b + c \neq \prec\succ && \textit{1-point rule} \\
\Rrightarrow\ & c \in \text{bag}\ T \wedge c \neq \prec\succ \quad \vee \quad b \in \text{bag}\ T \wedge b \neq \prec\succ \\
\Rrightarrow\ & gd(\mathit{UBuffer}^2_{right}) \quad \vee \quad gd(\mathit{UBuffer}^2_{mid}).
\end{aligned}
$$

Note that Definition 20 does not allow us to show the reverse refinement, i.e. $\mathit{UBuffer}^2 \sqsubseteq \mathit{UBuffer}^1$. However, if we assume that design proceeds by the introduction rather then elimination of internal actions, then $\mathit{UBuffer}^1 \sqsubseteq \mathit{UBuffer}^2$ is all we require.

9 Basic Parallel Composition

In this section, we introduce a parallel operator for action systems without value-passing. Firstly, we look at parallel composition in CSP. The CSP process $\mathcal{P} \parallel \mathcal{Q}$ represents the parallel composition of the processes $\mathcal{P}$ and $\mathcal{Q}$. Operationally, $\mathcal{P}$ and $\mathcal{Q}$ interact by synchronising over common events in $alphabet(\mathcal{P}) \cap alphabet(\mathcal{Q})$, while events not in $alphabet(\mathcal{P}) \cap alphabet(\mathcal{Q})$ can occur independently. An event common to both $\mathcal{P}$ and $\mathcal{Q}$ becomes a single event in $\mathcal{P} \parallel \mathcal{Q}$, and can be offered by $\mathcal{P} \parallel \mathcal{Q}$ only when both $\mathcal{P}$ and $\mathcal{Q}$ are prepared to offer it.

As an example, consider the CSP processes

$$
\mathcal{N}1 \ \hat{=}\ (\mu\, X \bullet a \to c \to X), \quad \mathcal{N}2 \ \hat{=}\ (\mu\, X \bullet b \to c \to X).
$$

Here c is the only event common to $\mathcal{N}1$ and $\mathcal{N}2$, and using the algebraic laws of CSP it can be shown that

$$\begin{aligned} \mathcal{N}1 \parallel \mathcal{N}2 \;=\; (\mu\, X \bullet \quad & a \to b \to c \to X \\ & \square\;\; b \to a \to c \to X \;). \end{aligned}$$

That is, the a or the b events can occur in either order, then both processes must synchronise on event c.

Correspondingly, we write the parallel composition of action systems P and Q as $P \parallel Q$. The actions of $P \parallel Q$ are formed from the actions of P and Q: commonly labelled actions are composed in parallel, while independently labelled actions remain independent. An important assumption we make is that action systems P and Q have no common state-variables, so that interaction between P and Q is based purely on synchronisation.

In order to construct $P \parallel Q$, we introduce an operator for combining individual actions in parallel. The parallel composition of actions S and S' is written $S \parallel S'$, and is only valid when the variables changed by S and S' are independent. The operator is (partially) defined as follows:

Definition 21 *Let x and x' be independent variables. Let $E, G, Com, post$ be independent of x' and $E', G', Com', post'$ be independent of x. The parallel operator for actions is given by:*

$$\begin{aligned} (x := E) \;\parallel\; (x' := E') \;&\hat{=}\; (x, x' := E, E') \\ (G \to Com) \;\parallel\; (G' \to Com') \;&\hat{=}\; (G \wedge G' \to Com \parallel Com') \\ x : [post] \;\parallel\; x' : [post'] \;&\hat{=}\; x, x' : [post \wedge post']. \end{aligned}$$

Since the variables changed by constituent actions are independent, the only effect of the parallel operator for actions is to ensure that the composite action is enabled exactly when both component actions are enabled.

Parallel composition of action systems without value-passing is then defined as follows (assume from here on that v and w are distinct):

Definition 22 *For action systems*

$$P = (A, v, P_\imath, P_A, P_H) \;\; and \;\; Q = (B, w, Q_\imath, Q_B, Q_G)$$

$$P \parallel Q \;\hat{=}\; (A \cup B, \quad (v, w), \quad P_\imath \parallel Q_\imath, \quad par(P_A, Q_B), \quad P_H \cup Q_G)$$

system *N1*
  var $m : \mathbb{N}$
  initially $m := 0$
  action $a \bullet \quad m = 0 \to m := 1$
  action $c \bullet \quad m = 1 \to m := 0$
end

system *N2*
  var $n : \mathbb{N}$
  initially $n := 0$
  action $b \bullet \quad n = 0 \to n := 1$
  action $c \bullet \quad n = 1 \to n := 0$
end

system $N1 \parallel N2$
  var $m, n : \mathbb{N}$
  initially $m, n := 0, 0$
  action $a \bullet \quad m = 0 \to m := 1$
  action $b \bullet \quad n = 0 \to n := 1$
  action $c \bullet \quad m = 1 \wedge n = 1 \to m, n := 0, 0$
end

Fig. 6. Example parallel action systems.

$$where \quad par(P_A, Q_B) \ \hat{=}\ P_{A-B} \ \cup\ Q_{B-A} \ \cup\ \{\ P_c \parallel Q_c \mid c \in A \cap B\ \}.$$

The alphabet of $P \parallel Q$ is the union of A and B. The state variable is the pair (v, w). The initialisations and the commonly labelled actions are respectively composed using the parallel operator for actions. Independent actions remain independent. The respective internal actions are simply bunched together.

As an example of parallel composition, consider the action systems *N1*, *N2*, and $N1 \parallel N2$ of Figure 6. $N1 \parallel N2$ has been constructed from *N1* and *N2* using Definition 22. Notice the correspondence with the CSP processes described at the beginning of this section.

As with hiding, the semantics of CSP process $\mathcal{P} \parallel \mathcal{Q}$ is defined as a function, PAR, of the semantics of $\mathcal{P}$ and $\mathcal{Q}$:

$$[\![\mathcal{P} \parallel \mathcal{Q}]\!] \ \hat{=}\ \mathsf{PAR}([\![\mathcal{P}]\!], [\![\mathcal{Q}]\!]),$$

where PAR is defined in [17]. It can be shown that parallel composition for action systems corresponds to CSP parallel composition:

Theorem 23 $\{\![P \parallel Q]\!\} \ =\ \mathsf{PAR}(\{\![P]\!\}, \{\![Q]\!\})$.

Theorem 23 is proven in [8]. The proof is based on using properties of the parallel operator for actions to show that $\{\!\![P \parallel Q]\!\!\}$ can be characterised in terms of $\{\!\![P]\!\!\}$ and $\{\!\![Q]\!\!\}$. Because of Theorem 23 the parallel operator for action systems enjoys the same properties as the CSP parallel operator such as commutativity, associativity, and monotonicity w.r.t. refinement (P, Q, and R are action systems):

Rule 24 $P \parallel Q \;=\; Q \parallel P.$

Rule 25 $(P \parallel Q) \parallel R \;=\; P \parallel (Q \parallel R).$

Rule 26 *If* $\;P \sqsubseteq P'\;$ *then* $\;P \parallel Q \;\sqsubseteq\; P' \parallel Q,\;$ *for any* Q.

Because of Rules 24 and 25, we write the parallel composition of a finite collection of action systems P_i as $(\parallel\, i \bullet P_i)$, where $(\parallel\, i \bullet P_i)$ is calculated by successive application of the binary parallel operator. Such calculation can deal with multi-party interaction where more than two action systems share some action label.

Rule 26 (along with Rules 24 and 25) means that any parallel component of a distributed system can be further refined and decomposed independently of the rest of the system.

10 Parallel Composition with Value-Passing

In this section, we extend the parallel operator to deal with parameterised actions and value-passing. First, we look at CSP value-passing. In CSP, when an output channel is placed in parallel with a similarly named input channel, passing of values from one process to the other is possible. Assume process $\mathcal{P}$ is ready for $c!e$, and process $\mathcal{Q}$ is ready for $c?x$. If $\mathcal{P}$ and $\mathcal{Q}$ are placed in parallel, then they can engage simultaneously in the event $c.e$ and, in this way, the value e is passed from $\mathcal{P}$ to $\mathcal{Q}$. This is represented by the following algebraic law:

$$(c!e \to \mathcal{P}) \parallel (c?x \to \mathcal{Q}_x) = c.e \to (\mathcal{P} \parallel \mathcal{Q}_e).$$

Note that the right-hand side of this law can itself be written $c!e \to (\mathcal{P} \parallel \mathcal{Q}_e)$.

In order to achieve a similar effect in action systems, we introduce an operator for composing an output action with an input action. If S is an output action and S' is an input action, then their composition is written $S \overrightarrow{\parallel} S'$. The effect of $S \overrightarrow{\parallel} S'$ is to synchronise S and S', and to pass on the output parameter $y!$ of S

as the input parameter $x?$ of S'. We shall assume that S and S' are specified using $v : [post]$ notation. The construction of $S \vec{\|} S'$ is then defined as follows:

Definition 27 *Let post be independent of w and $x?$, and let post′ be independent of v and $y!$. Then*

$$\begin{array}{l}(\textbf{action}\ \alpha\ \textbf{out}\ y! : T \bullet\ \ v : [post])\ \vec{\|}\ (\textbf{action}\ \alpha\ \textbf{in}\ x? : T' \bullet\ \ w : [post']) \\ \quad \hat{=}\ \ (\textbf{action}\ \alpha\ \textbf{out}\ y! : T \bullet\ \ \ v, w : [\ post \land post'[x? \backslash y!]\ ]).\end{array}$$

Note that, as with the CSP case, the composite action is itself an output action.

We shall place a further restriction on the construction of $S \vec{\|} S'$, as described by the following rule:

Rule 28 *Assume S and S' have the form:*

$$\begin{array}{rcl} S & = & \textbf{action}\ \alpha\ \textbf{out}\ y! : T \bullet \quad v : [post] \\ S' & = & \textbf{action}\ \alpha\ \textbf{in}\ x? : T' \bullet \quad w : [post'] \end{array}$$

Then the construction of $S \vec{\|} S'$ is allowed only if the following conditions are satisfied:

(i) $(\exists\, x?, w \bullet post') \land x? \in T' \ \Rrightarrow\ (\exists\, w \bullet post')$
(ii) $T \subseteq T'$.

The first condition means that whenever S' is enabled for some value of $x?$, it is enabled for all $x?$ of type T'. For example,

$$\textbf{action}\ \alpha\ \textbf{in}\ x? : \mathbb{N} \bullet \quad s : [s = s_0 \frown \langle x? \rangle]$$

satisfies Condition 1, though

$$\textbf{action}\ \alpha\ \textbf{in}\ x? : \mathbb{N} \bullet \quad s : [x? \geq 10 \land s = s_0 \frown \langle x? \rangle]$$

does not since it won't accept naturals less than 10. The purpose of Rule 28 is to ensure that an output value produced by S is always acceptable as an input value by S'. Notice from Definition 27 that the output type of a composite output action is exactly the type of the constituent output action.

As an example of value-passing composition, assume

$$\begin{array}{rcl} S & = & \textbf{action}\ mid\ \textbf{out}\ y! : T \bullet \quad b : [\ y! \in b_0 \land b = b_0 - \prec y! \succ\ ] \\ S' & = & \textbf{action}\ mid\ \textbf{in}\ x? : T \bullet \quad c : [\ c = c_0 + \prec x? \succ\ ]. \end{array}$$

Then

$$S \vec{\|} S' \;=\; \textbf{action}\ mid\ \textbf{out}\ y! : T \bullet \quad b, c\ :\ \begin{bmatrix} y! \in b_0 \wedge b = b_0 - \prec y! \succ \wedge \\ c = c_0 + \prec y! \succ \end{bmatrix}.$$

We also permit the parallel composition of two input actions. Both actions share the same input parameter and the composite action is itself an input action. Parallel composition of two input actions is defined as follows:

Definition 29 *Let post be independent of w, and let $post'$ be independent of v. Then*

$$\begin{array}{l} (\textbf{action}\ \alpha\ \textbf{in}\ x? : T \bullet \quad v : [post]) \;\|\; (\textbf{action}\ \alpha\ \textbf{in}\ x? : T' \bullet \quad w : [post']) \\ \quad \hat{=}\ \ (\textbf{action}\ \alpha\ \textbf{in}\ x? : (T \cap T') \bullet \quad v, w : [\ post \wedge post'\ ]). \end{array}$$

Notice that the input type of the composite action is the intersection of the types of the constituent actions.

The parallel composition of two output actions is not permitted.

Parallel composition of value-passing action systems is defined as follows:

Definition 30 *For value-passing action systems*

$$\begin{array}{l} P = (A,\ v,\ P_\imath,\ P_A,\ P_H,\ dir_P,\ type_P) \\ Q = (B,\ w,\ Q_\imath,\ Q_B,\ Q_G,\ dir_Q,\ type_Q) \end{array}$$

If the following conditions are satisfied:

(i) P and Q have no commonly labelled output actions
(ii) Commonly labelled input-output pairs from P and Q satisfy Rule 28,

then $P \parallel Q$ is defined as follows:

$$P \parallel Q \;\hat{=}\; (A \cup B,\ (v, w),\ P_\imath \parallel Q_\imath,\ par(P_A, Q_B),\ P_H \cup Q_G,\ dir,\ type)$$

where

$$\begin{array}{rl} par(P_A, Q_B) \;\hat{=} & P_{A-B} \cup Q_{B-A} \\ & \cup \{\ P_c \vec{\|} Q_c \mid c \in OUT_P\ \} \\ & \cup \{\ Q_c \vec{\|} P_c \mid c \in OUT_Q\ \} \\ & \cup \{\ P_c \parallel Q_c \mid c \in (A \cup B) - (OUT_P \cup OUT_Q)\ \} \end{array}$$

system $UBuffer^l$
  var b : bag T
  initially $b : [\ b = \prec\succ\ ]$
  action *left* **in** $x? : T \bullet$ $\quad b : [\ b = b_0 + \prec x?\succ\ ]$
  action *mid* **out** $y! : T \bullet$ $\quad b : [\ y! \in b_0 \wedge b = b_0 - \prec y!\succ\ ]$
end

system $UBuffer^r$
  var c : bag T
  initially $c : [\ c = \prec\succ\ ]$
  action *mid* **in** $x? : T \bullet$ $\quad c : [\ c = c_0 + \prec x?\succ\ ]$
  action *right* **out** $y! : T \bullet$ $\quad c : [\ y! \in c_0 \wedge c = c_0 - \prec y!\succ\ ]$
end

system $UBuffer^l \parallel UBuffer^r$
  var b, c : bag T
  initially $b, c : [\ b = c = \prec\succ\ ]$
  action *left* **in** $x? : T \bullet$ $\quad b : [\ b = b_0 + \prec x?\succ\ ]$
  action *right* **out** $y! : T \bullet$ $\quad c : [\ y! \in c_0 \wedge c = c_0 - \prec y!\succ\ ]$
  action *mid* **out** $y! : T \bullet$ $\quad b, c : \begin{bmatrix} y! \in b_0 \wedge b = b_0 - \prec y!\succ \wedge \\ c = c_0 + \prec y!\succ \end{bmatrix}$
end

Fig. 7. Parallel buffers.

$$\begin{array}{lcl} OUT_P & \hat{=} & \{\ c \in A \cap B \mid dir_P(c) = out \wedge dir_Q(c) = in\ \} \\ OUT_Q & \hat{=} & \{\ c \in A \cap B \mid dir_Q(c) = out \wedge dir_P(c) = in\ \} \end{array}$$

and dir and type are as entailed by Definitions 27 and 29.

Theorem 23 also holds for parallel composition of value-passing action systems so that Rules 24, 25, and 26 are valid for value-passing action systems.

Figure 7 describes the action systems $UBuffer^l$ and $UBuffer^r$. $UBuffer^l$ is simply an unbounded buffer with *right* renamed to *mid*, while $UBuffer^r$ has *left* renamed to *mid*. When $UBuffer^l$ and $UBuffer^r$ are placed in parallel, they interact via the *mid* channel, with values being passed from $UBuffer^l$ to $UBuffer^r$. This can be seen by constructing the composite action system $UBuffer^l \parallel UBuffer^r$ using Definition 30 (see Figure 7).

If the *mid* action of $UBuffer^l \parallel UBuffer^r$ is hidden, then the resultant action system is the same as $UBuffer^2$ of Figure 5. In order to internalise *mid*, the output parameter must be localised. This may be achieved using the following rule (derived from Definitions 16 and 1):

Rule 31 $(\mathbf{var}\ y! \bullet v : [\ post\ ]) \ = \ v : [\ (\exists\, y! \bullet post)\ ].$

Now, using Definition 15 and Rule 31, it is easy to show that

$$UBuffer^{2} = (\,UBuffer^{l} \parallel UBuffer^{r}) \setminus \{mid\}.$$

Since $UBuffer^{1} \sqsubseteq UBuffer^{2}$, we have that:

$$UBuffer^{1} \sqsubseteq (\,UBuffer^{l} \parallel UBuffer^{r}) \setminus \{mid\}.$$

Note on Restrictions

In CSP terms, violation of conditions (i) and (ii) of Definition 29 may introduce deadlock; in the first case, because the input action may not be willing to accept the value being output by the output action, and in the second case because both output actions may not be willing to output the same value; in both cases the choice between whether or not deadlock occurs may be nondeterministic. But we cannot model the nondeterministic choice between being enabled and not being enabled in the guard of an action. Rather we would have to represent the nondeterministic choice between whether or not deadlock occurs by the nondeterministic setting of some flag and adding a test on that flag to the guard of the composite action. Thus, without conditions (i) and (ii), the composite action system $P \parallel Q$ cannot be constructed from P and Q by simply composing commonly-labelled action, but rather would require a much more complicated construction.

11 Message-Passing System

The action systems of this case study contain indexed sets of channels, each one offering similar behaviour. An indexed statement is used to specify the actions associated with such channel sets. For example, to specify an indexed set of input channels $\{\ left_i \mid i \in F\ \}$, with associated types and actions, the following notation is used:

$$\textbf{for } i \in F \textbf{ action } left_i \textbf{ in } x? : T \bullet \qquad u_i : [\ post_i\ ].$$

The intention is that the i-indexed statement represents a set of input actions. Similarly for an indexed set of output channels. An indexed set of internal actions $\{\ h_i \mid i \in F\ \}$ is specified by

$$\textbf{for } i \in F \textbf{ internal } \ h_i \bullet \qquad u_i : [\ post_i\ ].$$

system MPS^1
var $mail$: bag Env
initially $mail : [\ mail = \prec\succ\ ]$

for $s \in Node$ **action** $send_s$ **in** $(r?, m?) : Env$ $\bullet$
$mail : [\ mail = mail_0 + \prec(r?, m?)\succ\ ]$

for $r \in Node$ **action** $receive_r$ **out** $m! : Mess$ $\bullet$
$mail : [\ (r, m!) \in mail_0 \wedge mail = mail_0 - \prec(r, m!)\succ\ ]$
end

Fig. 8. Message-passing system.

We suppose that a message-passing system allows a set of users to exchange messages amongst each other. Each user resides at a node, and each user may engage in either a *send* action, or a *receive* action. Let *Node* represent the set of nodes in the system. We shall assume that *Node* is finite. Let *Mess* represent the type of messages that may be exchanged, and let *Env* be the cartesian product of *Node* and *Mess*, i.e.

$$Env \ \hat{=}\ Node \times Mess.$$

In the pair $(r, m) \in Env$, r is the recipient node, m is the message, and we say that (r, m) is an *envelope*.

The initial specification of the message-passing system, MPS^1, is given in Figure 8. Variable *mail* contains all messages sent but not yet received. Initially *mail* is empty. For each node n, there is a $send_n$ action and a $receive_n$ action. Action $send_s$ accepts an envelope $(r?, m?)$ at sending node s and adds it to the bag *mail*. If there is at least one message for recipient node r in *mail*, then action $receive_r$ chooses one of these messages and outputs it.

Our goal is to implement MPS^1 as a *store-and-forward* network, where not all nodes are directly connected, and envelopes must pass through a number of intermediate nodes before reaching their recipient. In the first refinement step, we introduce data structures more closely resembling the store-and-forward architecture, and introduce internal actions for passing envelopes between these data structures. Then the system is decomposed into a set of *agents*, one residing at each node, and a set of *media*, by which the agents communicate.

Before proceeding with this case study, we introduce some notational conventions concerning invariants and array variables.

Invariants

We shall sometimes require state variables to satisfy invariants. A variable v with invariant I is declared by

$$\textbf{var } v \textbf{ where } I.$$

We assume that I is an implicit part of the initialisation and of each action as follows: the predicate *post* of an initialisation is replaced by $I \wedge post$, while the predicate *post* of an action is replaced by $I[v_0 \backslash v] \wedge I \wedge post$. Also, we write "$\textbf{var } v : T \textbf{ where } I$" as short for "$\textbf{var } v \textbf{ where } I \wedge v \in T$".

Z *schemas* [34] can be used to specify the state variables and invariant of an action system. We shall use schemas of the form

```
┌─ Sch ──────────────────────────
│ v : T
├───────────
│ I
└────────────────────────────────
```

where v is a state variable, T is a type, and I is a predicate. If *Sch* is defined as shown, then the declaration **var** *Sch* is short for $\textbf{var } v : T \textbf{ where } I$.

Array Variables

An array of type T with index set D may be represented by a function $f \in D \rightarrow T$. A specification statement can be made to update f at index i by containing the formula

$$f = f_0 \oplus \{i \mapsto E\},$$

where $f_0 \oplus \{i \mapsto E\}$ represents the overriding of function f_0 with the function $\{i \mapsto E\}$. For convenience, $f = f_0 \oplus \{i \mapsto E\}$ will be written simply as $f(i) = E$.

11.1 First Refinement of MPS

In MPS^2, *mail* is replaced by a set of stores, one per node, and a set of buffers representing direct links between nodes. The constant relation $net \in Node \leftrightarrow Node$ represents the connectivity of the network: $(a, b) \in net$ means there is a direct communications link from node a to node b.

Routing relations are used to determine which intermediate nodes an envelope may pass through. Before defining a *route*, we present some simple graph theory concepts [32]. We say that a *graph* G is a relation on a set of nodes N (e.g., *net* is a graph on *Node*). A *path* from a to b in G is a non-empty sequence p of nodes from N, such that $p_i G p_{i+1}$, for each $0 \leq i < \#p - 1$, and $p_0 = a$ and $p_{\#p-1} = b$. Let G^* be the reflexive transitive closure of G. Then aG^*b means there is a path from a to b in G. Note that there is always a path from a to a in G. An *arc* from a to b in G is a path from a to b in which all nodes are distinct. If N is finite, then the *elongation* from a to b in G, written $e_G(a, b)$, is the length of the longest arc from a to b in G. Since the only arc from a to a is $\langle a \rangle$, we have $e_G(a, a) = 1$. We define *routes* as follows:

Definition 32 *Let G be a graph on nodes N. Then $Routes(G)$, the set of* routes *of G, is the set of subgraphs of G such that for all $R \in Routes(G)$, and all $a, b, c \in N$, where $a \neq c$,*

$$aRb \wedge bR^*c \;\Rightarrow\; e_R(a, c) > e_R(b, c).$$

Here, each $R \in Routes(G)$ is a routing relation, (a, b) is a single step in R, and c is a destination node. The definition says that as we move from node a to node b on the route, the elongation to the destination node c decreases.

MPS^2 will contain a fixed set of routes, each one uniquely identified by a tag from a set *Tag*. These routes will be represented by the constant function

$$route \;\in\; Tag \rightarrow Routes(net).$$

On input, each envelope will be assigned one of these routes by being tagged with the route identifier. At any point on its journey the choice of the next node to which an envelope is sent will be determined by its destination and its assigned route. Since a route is a relation, the choice of next node may be nondeterministic. We shall use elongations as a variant to ensure that all envelopes eventually reach their destination.

The state variables of MPS^2, with invariant, are declared by the following schema:

$$
\begin{array}{|l}
\hline
\textit{INV}^2 \\
store \;:\; Node \rightarrow \mathrm{bag}\ (Tag \times Env) \\
link \;:\; net \rightarrow \mathrm{bag}\ (Tag \times Env) \\
\hline
(\forall (i, r, m) \in Tag \times Env;\ a, b \in Node \bullet \\
\quad (i, r, m) \in store(a) \;\Rightarrow\; (a, r) \in route(i)^* \\
\quad (a, b) \in net \wedge (i, r, m) \in link(a, b) \;\Rightarrow\; (b, r) \in route(i)^*) \\
\hline
\end{array}
$$

system MPS^2
var INV^2

$$\textbf{initially} \quad \begin{matrix} store, \\ link \end{matrix} : \left[\begin{array}{l} (\forall\, a \in Node \bullet store(a) = \prec\succ) \\ \wedge\ (\forall (a,b) \in net \bullet link(a,b) = \prec\succ) \end{array}\right]$$

for $s \in Node$ **action** $send_s$ **in** $(r?, m?) : Env \bullet$

$$store : \left[\begin{array}{l} (\exists\, i \in Tag \bullet (s, r?) \in route(i)^* \\ \qquad \wedge\ store(s) = store_0(s) + \prec(i, r?, m?)\succ\ ) \end{array}\right]$$

for $r \in Node$ **action** $receive_r$ **out** $m! : Mess \bullet$

$$store : \left[\begin{array}{l} (\exists\, i \bullet (i, r, m!) \in store_0(r) \\ \qquad \wedge\ store(r) = store_0(r) - \prec(i, r, m!)\succ\ ) \end{array}\right]$$

internal $forward \bullet$

$$\begin{matrix} store, \\ link \end{matrix} : \left[\begin{array}{l} (\exists (a,b) \in net;\ (i, r, m) \in store_0(a) \bullet \\ \qquad r \neq a \wedge (a,b) \in route(i) \wedge (b, r) \in route(i)^* \\ \qquad \wedge\ store(a) = store_0(a) - \prec(i, r, m)\succ \\ \qquad \wedge\ link(a,b) = link_0(a,b) + \prec(i, r, m)\succ\ ) \end{array}\right]$$

internal $relay \bullet$

$$\begin{matrix} store, \\ link \end{matrix} : \left[\begin{array}{l} (\exists (a,b) \in net;\ (i, r, m) \in link_0(a,b) \bullet \\ \qquad \wedge\ link(a,b) = link_0(a,b) - \prec(i, r, m)\succ \\ \qquad \wedge\ store(b) = store_0(b) + \prec(i, r, m)\succ\ ) \end{array}\right]$$

end

Fig. 9. Message-passing system with internal actions.

Corresponding to each node in the network, there is a store (bag) of tagged envelopes. These are modelled by the variable *store*. Corresponding to each direct link in the network, there is an unordered buffer of tagged envelopes. These are modelled by the variable *link*. The predicate part of INV^2 means that there is always a path from the current position of an envelope to its recipient in the assigned route. In order that each distinct pair of nodes be connected by at least one route, we shall assume that the constant function *route* satisfies:

$$(\cup i \in Tag \bullet route(i)^*) = Node \times Node.$$

MPS^2 is then specified by Figure 9. All stores and links are initially empty. The action $send_s$ accepts an envelope $(r?, m?)$, chooses a route i that (directly or indirectly) connects s to $r?$, and adds $(i, r?, m?)$ to the bag $store(s)$. If there is at least one message for recipient r in $store(r)$, then action $receive_r$ chooses one of those messages and outputs it.

The internal action *forward* takes a tagged envelope that has not yet reached its recipient from some $store(a)$, chooses the next node b to forward the envelope to, and places the envelope in $link(a, b)$. The internal action *relay* simply takes an envelope from some $link(a, b)$ and places it in $store(b)$.

By a sequence of *forward* and *relay* actions, a message sent at node s is eventually delivered to the store of its recipient node r. This is the case since $MPS^1 \sqsubseteq MPS^2$, which may be checked using Definition 20, as follows:

Data Refinement Conditions

The abstract and the concrete variables are related by equating *mail* with the sum of envelopes in each store and each link. We write $(\Sigma i \bullet b_i)$ for the summation of a set of bags b_i. Let *env* be the function that removes tags from tagged envelopes, i.e. $env(i, r, m) = (r, m)$. If b is a bag of tagged envelopes, then $env(b)$ is the corresponding bag of untagged envelopes. The abstract invariant AI is defined as follows:

$$\begin{aligned} AI \;\hat{=}\; INV^2 \;\wedge\; mail = {} & (\Sigma a \in Node \bullet env(store(a))) \\ & +(\Sigma (a, b) \in net \bullet env(link(a, b))). \end{aligned}$$

The data refinement obligations from Definition 20 are as follows (let α^i be the α action of MPS^i):

$$\begin{aligned} init^1 &\preceq'_{AI} init^2 \\ send^1_s &\preceq_{AI} send^2_s \\ receive^1_r &\preceq_{AI} receive^2_r \\ \mathbf{skip} &\preceq_{AI} forward^2 \\ \mathbf{skip} &\preceq_{AI} relay^2. \end{aligned}$$

These are easily proven using Rules 8 and 10 and properties of bags.

Non-divergence Conditions

A variant that is decreased by both $forward^2$ and $relay^2$ must be chosen. We will use the elongation from the current position of each envelope to its destination in its route to define a variant. Let $(\Sigma j \bullet n_j)$ represent the summation of a set of naturals n_j, and let $e_i(a, b)$ be the elongation from a to b on $route(i)$, i.e. $e_{route(i)}(a, b)$. The variant E is then defined as follows:

$$E \;\hat{=}\; (\Sigma a \in Node \bullet (\Sigma (i, r, m) \in store(a) \bullet e_i(a, r) * 2))$$

$$+(\Sigma(a,b) \in net \bullet (\Sigma(i,r,m) \in link(a,b) \bullet (e_i(b,r) * 2) + 1)).$$

It is easy to show that $INV^2 \Rrightarrow E \in \mathbb{N}$. Definition 1 and the definition of routes may be used to show that

$$\begin{aligned} INV^2 \wedge E = e &\Rrightarrow wp(forward^2, E < e) \\ INV^2 \wedge E = e &\Rrightarrow wp(relay^2, E < e). \end{aligned}$$

So the non-divergence conditions are satisfied.

Progress Conditions

Finally, it is easy to show, for each s, r, that

$$\begin{aligned} (\exists\, mail \bullet AI \wedge gd(send_s^1)) &\Rrightarrow gd(send_s^2) \\ (\exists\, mail \bullet AI \wedge gd(receive_r^1)) &\Rrightarrow gd(receive_r^2) \vee gd(forward^2) \vee gd(relay^2). \end{aligned}$$

Thus all the simulation conditions of Definition 20 are satisfied and $MPS^1 \sqsubseteq MPS^2$. Our next design step will be to decompose MPS^2. However, before doing this we shall simplify the invariant of MPS^2 in order to simplify the obligations of further design steps.

11.2 Invariant Simplification

A statement S is said to *establish* invariant I if $true \Rrightarrow wp(S, I)$. A statement S is said to *preserve* invariant I if $I \Rrightarrow wp(S, I)$. If an invariant is established by the initialisation statement of an action system, and preserved by each action, then that invariant can be eliminated, as the following rule shows:

Rule 33 *Assume action system P has been defined to have invariant $I \wedge J$. Assume A is the channel set of P, and H is the label set of the internal actions of P. Let action system P' be the same as P but without the invariant I. Then $P \sqsubseteq P'$ provided*

(i) P'_i *establishes* I
(ii) P'_α *preserves* I, *each* $\alpha \in A \cup H$.

Rule 33 can be used to simplify the state invariant of MPS^2. The specification of MPS^3 is the same as that of MPS^2, except that its state invariant is simply:

$$\begin{array}{|l} \hline \textit{INV}^3 \\ store : Node \rightarrow \text{bag } (Tag \times Env) \\ link : net \rightarrow \text{bag } (Tag \times Env) \\ \hline \end{array}$$

It can easily be shown that the initialisation of MPS^3 establishes INV^2, and that each action of MPS^3 preserves INV^2. Hence, by Rule 33, with INV^2 for I and INV^3 for J, we have

$$MPS^2 \sqsubseteq MPS^3.$$

11.3 Parallel Decomposition of MPS

In this step, MPS^3 is decomposed into two parallel action systems: *Agents* and *Media*, specified in Figures 10 and 11. *Agents* represents the behaviour of all the nodes of the network, and has a *send*, *receive*, *forward*, and *relay* channel for each network node. *Agents* only has the state variable *store*. *Media* represents the communications links of the network, and has a *forward* and a *relay* channel for each network node. *Media* only has the state variable *link*. *Agents* and *Media* communicate via *forward* and *relay* channels, and we have that

$$\begin{array}{l} MPS^3 \;=\; \\ (Agents \parallel Media) \setminus \{\, forward_a \mid a \in Node \,\} \cup \{\, relay_b \mid b \in Node \,\}. \end{array}$$

This decomposition step is an example of what Hoare [17] terms *subordination*: the channel set of *Media* is a subset of the channel set of *Agents*, and the common channels to be hidden are exactly those of *Media*. *Media* is said to be subordinate to *Agents*, since its behaviour is exactly under the control of *Agents*.

To prove this decomposition step correct, we first replace the internal actions of MPS^3 with corresponding indexed sets of internal actions using the following rule:

Rule 34 *An internal action*

$$\textbf{internal}\ \ h \bullet \qquad u : [\ (\exists\, i \in F \bullet post_i)\ ]$$

is the same as the set of internal actions

$$\textbf{for}\ i \in F\ \textbf{internal}\ \ h_i \bullet \qquad u : [\ post_i\ ].$$

system *Agents*

$$
\begin{array}{l}
\textbf{var } store : Node \to \text{bag } (Tag \times Env) \\
\textbf{initially} \quad store : \left[(\forall\, a \in Node \bullet store(a) = \prec\succ) \right]
\end{array}
$$

for $s \in Node$ **action** $send_s$ **in** $(r?, m?) : Env \bullet$

$$
store : \begin{bmatrix} (\exists\, i \in Tag \bullet (s, r?) \in route(i)^* \\ \qquad \wedge\ store(s) = store_0(s) + \prec(i, r?, m?)\succ\ ) \end{bmatrix}
$$

for $r \in Node$ **action** $receive_r$ **out** $m! : Mess \bullet$

$$
store : \begin{bmatrix} (\exists\, i \bullet (i, r, m!) \in store_0(r) \\ \qquad \wedge\ store(r) = store_0(r) - \prec(i, r, m!)\succ\ ) \end{bmatrix}
$$

for $a \in Node$ **action** $forward_a$ **out** $b! : Node;\ i! : Tag;\ (r!, m!) : Env \bullet$

$$
store : \begin{bmatrix} (i!, r!, m!) \in store_0(a) \wedge r! \neq a \\ \wedge\ (a, b!) \in route(i!) \wedge (b!, r!) \in route(i!)^* \\ \wedge\ store(a) = store_0(a) - \prec(i!, r!, m!)\succ \end{bmatrix}
$$

for $b \in Node$ **action** $relay_b$ **in** $a? : Node;\ i? : Tag;\ (r?, m?) : Env \bullet$

$$
store : \begin{bmatrix} (a?, b) \in net \Rightarrow \\ \qquad store(b) = store_0(b) + \prec(i?, r?, m?)\succ \end{bmatrix}
$$

end

Fig. 10. Network agents.

system *Media*

$$
\begin{array}{l}
\textbf{var } link : net \to \text{bag } (Tag \times Env) \\
\textbf{initially} \quad link : \left[(\forall (a, b) \in net \bullet link(a, b) = \prec\succ) \right]
\end{array}
$$

for $a \in Node$ **action** $forward_a$ **in** $b? : Node;\ i? : Tag;\ (r?, m?) : Env \bullet$

$$
link : \begin{bmatrix} (a, b?) \in net \Rightarrow \\ \qquad link(a, b?) = link_0(a, b?) + \prec(i?, r?, m?)\succ \end{bmatrix}
$$

for $b \in Node$ **action** $relay_b$ **out** $a! : Node;\ i! : Tag;\ (r!, m!) : Env \bullet$

$$
link : \begin{bmatrix} (a!, b) \in net \wedge (i!, r!, m!) \in link_0(a!, b) \\ \wedge\ link(a!, b) = link_0(a!, b) - \prec(i!, r!, m!)\succ \end{bmatrix}
$$

end

Fig. 11. Network media.

For example, **internal** *relay* is replaced in MPS^3 by

for $b \in Node$ **internal** $relay_b \bullet$

$$
\begin{array}{c} store, \\ link \end{array} : \begin{bmatrix} (\exists\, a \in Node;\ (i, r, m) \in link \mid (a, b) \in net \bullet \\ \qquad \wedge\ link(a, b) = link_0(a, b) - \prec(i, r, m)\succ \\ \qquad \wedge\ store(b) = store_0(b) + \prec(i, r, m)\succ\ ) \end{bmatrix}
$$

Now, when each $relay_b$ action of *Agents* is composed with the corresponding $relay_b$ action of *Media* using Definition 27, and then internalised using Definition 15 and Rule 31, the result is the same as the $relay_b$ action of MPS^3. Similarly for the $forward_a$ actions.

Note 1: The first attempted specification of the $relay_b$ action of *Agents* was

$$store : \begin{bmatrix} (a?, b) \in net \wedge \\ store(b) = store_0(b) + \prec(i?, r?, m?)\succ \end{bmatrix}$$

However, this action violates Rule 28, since it only accepts input values $a?$ if $(a?, b) \in net$, and therefore, it cannot be placed in parallel with the $relay_b$ action of *Media*. The $relay_b$ action of *Agents* shown in Figure 10 accepts all input values $a? \in Node$, though it's behaviour is unspecified if $(a?, b) \notin net$. It could be refined, for example, by

$$store : \begin{bmatrix} (a?, b) \in net \Rightarrow store(b) = store_0(b) + \prec(i?, r?, m?)\succ \\ (a?, b) \notin net \Rightarrow store = store_0 \end{bmatrix}$$

The same is true of the $forward_a$ actions of *Media*.

Note 2: The $relay_b$ action of *Agents* doesn't check if b is on the route of the tagged envelope $(i?, r?, m?)$ that it accepts as input. So, on its own, *Agents* could accept a tagged envelope on channel $relay_b$ for which it has no routing information, i.e., INV^2 could be violated. Similarly, *Media* never checks tagged envelopes for violation of INV^2. However, since MPS^3 preserves INV^2, the combined system $Agents \parallel Media$ also preserves it, and so envelopes for which there is no routing information can never be communicated between *Agents* and *Media*.

11.4 Parallel Decomposition of Agents

In this step, *Agents* is decomposed into a set of parallel action systems, each one representing the behaviour of an individual node of the network. We have

$$Agents = (\parallel n \in Node \bullet Agent_n).$$

$Agent_n$ is specified in Figure 12. Each $Agent_n$ only has the state variable $store_n$, which is simply a bag of tagged envelopes.

The decomposition is easily checked by generalised application of the binary parallel operator: Because the individual agents do not share any actions, no

$$
\begin{array}{l}
\textbf{system} \;\; Agent_n \\
\quad \textbf{var}\; store_n : \text{bag}\; (Tag \times Env) \\
\quad \textbf{initially} \;\; store_n : \left[store_n = \prec\succ \right] \\
\\
\quad \textbf{action}\; send_n \;\textbf{in}\; (r?, m?) : Env \bullet \\
\qquad store_n : \left[\begin{array}{l} (\exists\, i \in Tag \bullet (n, r?) \in route(i)^* \\ \qquad \wedge\; store_n = (store_n)_0 + \prec(i, r?, m?)\succ\;) \end{array} \right] \\
\\
\quad \textbf{action}\; receive_n \;\textbf{out}\; m! : Mess \bullet \\
\qquad store_n : \left[\begin{array}{l} (\exists\, i \bullet (i, n, m!) \in (store_n)_0 \\ \qquad \wedge\; store_n = (store_n)_0 - \prec(i, n, m!)\succ\;) \end{array} \right] \\
\\
\quad \textbf{action}\; forward_n \;\textbf{out}\; b! : Node;\; i! : Tag;\; (r!, m!) : Env \bullet \\
\qquad store_n : \left[\begin{array}{l} (i!, r!, m!) \in (store_n)_0 \wedge r! \neq n \\ \wedge\; (n, b!) \in route(i) \wedge (b!, r!) \in route(i)^* \\ \wedge\; store_n = (store_n)_0 - \prec(i!, r!, m!)\succ \end{array} \right] \\
\\
\quad \textbf{action}\; relay_n \;\textbf{in}\; a? : Node;\; i? : Tag;\; (r?, m?) : Env \bullet \\
\qquad store_n : \left[\begin{array}{l} (a?, n) \in net \Rightarrow \\ \qquad store_n = (store_n)_0 + \prec(i?, r?, m?)\succ \end{array} \right] \\
\textbf{end}
\end{array}
$$

Fig. 12. Individual agent.

labelled actions are composed when constructing $(\| \; n \in Node \bullet Agent_n)$. Since the function *store* is well-defined for all $n \in Node$, we equate a statement such as: $store : [\; store(n) = E \;]$ with $store_n : [\; store_n = E \;]$. Hence, we equate an action α_n of $Agent_n$ with action α_n of *Agents*. The only statements that need to be composed when constructing $(\| \; n \in Node \bullet Agent_n)$ are the respective initialisations. These are composed using the following rule:

Rule 35 *Let* $\langle\; v_i \mid i \in F \;\rangle$ *be a partition of* v. *Then*

$$(\| \; i \in F \bullet v_i : [\; post_i \;]) = v : [\; (\forall\, i \in F \bullet post_i) \;],$$

provided each $post_i$ *is independent of* v_j*, for* $j \neq i$.

Using this rule we get:

$$
\begin{array}{l}
\quad (\| \in Node \bullet store_n : [\; store_n = \prec\succ \;]) \\
= (\cup n \in Node \bullet store_n) : [\; (\forall\, n \in Node \bullet store_n = \prec\succ) \;].
\end{array}
$$

We equate this with $store : [\; (\forall\, n \in Node \bullet store(n) = \prec\succ) \;]$.

The fixed routing relations *route* may be replaced by fixed routing tables in a subsequent refinement step. For $n, r \in Node$, $i \in Tag$, we define $table_n(r, i)$ as follows:

$$table_n(r,i) \;\hat{=}\; \{\; b \in Node \mid (n,b) \in route(i) \wedge (b,r) \in route(i)^* \;\}.$$

At each node n, $table_n(r, i)$ is the set of next nodes to which messages for recipient r on $route(i)$ may be sent. The $forward_n$ action of $Agent_n$ may then be replaced by the following:

$$\begin{array}{l}
\textbf{action } forward_n \textbf{ out } b! : Node;\ i! : Tag;\ (r!, m!) : Env \bullet \\
\qquad store_n : \left[\begin{array}{l} (i!, r!, m!) \in (store_n)_0 \wedge r! \neq n \\ \wedge\ b! \in table_n(r!, i!) \\ \wedge\ store_n = (store_n)_0 - \prec(i!, r!, m!)\succ \end{array}\right]
\end{array}$$

The choice of which next node to pass a message to is nondeterministic if $table_n(r, i)$ contains more than one element. A possible further refinement would be to introduce an ordering on elements of $table_n(r, i)$ and choose elements from $table_n(r, i)$ for successive messages on a "round-robin" basis.

In [8], an adaptive routing mechanism is introduced, where congestion information is passed around the network, and the node chosen from $table_n(i, r)$ is on the "least-congested" path to recipient r.

12 Related Work

Josephs [22] and He Jifeng [15] have defined the CSP failures of transition systems in which actions are relations between states rather than predicate transformers. He Jifeng defines divergences as well. Both define simulation, hiding and parallel composition in the relational approach. In [15], the divergent state needs to be modelled explicitly, which complicates the definitions. This is avoided, in our case, by the use of predicate transformers rather than relations.

Morgan has already defined the CSP failure-divergences semantics of (non-value-passing) action systems (without internal actions) in predicate-transformer terms [26]. This idea was developed further to deal with unbounded nondeterminism, value-passing, internal actions, and parallel composition in [8] and these developments are outlined here in the form of definitions and rules.

The two simulation rules presented here are forms of *downward* simulation

[16]. Woodcock and Morgan [37] have shown how downward simulation together with the complementary notion of *upward* simulation provide a complete method for (failures-divergences) refinement of action systems. However, this completeness result does not hold for action systems with internal actions, nor does it allow for unbounded nondeterminism.

In several state-based approaches to concurrent refinement [1,5,35], more emphasis is placed on the state of a system rather than its actions. A reactive system is modelled by a set of state-traces describing the possible evolution of the state and simulation between systems implies state-trace refinement. Several authors have treated the concurrent composition of state-based systems [2,5,11,12,18]. In all these approaches, interaction between sub-systems is based on access to shared state. Concurrent components of a system cannot be refined independently of the rest of the system. Rather there are certain assumptions that have to be made about the environment, and such assumptions are used and composed in certain ways [2].

I/O-automata, introduced by Lynch and Tuttle [24], are similar to action systems but with some fairness requirements. The actions of an I/O-automaton are partitioned into input events, output events, and internal events. An I/O-automaton is modelled by a set of finite and infinite traces of input and output events, and refinement is defined in terms of trace inclusion. Unlike the CSP failures-divergences model, the traces model for I/O-automaton does not distinguish internal and external nondeterminism, though this is obviated by the use of fairness requirements. Lynch [23] has defined a form of refinement mapping for I/O-automata called a *multivalued possibilities mapping* which is similar to our simulation relation. The parallel composition operator for I/O-automata is also based on interaction via shared events. Jonsson [21] takes a similar approach to Lynch and Tuttle. However, instead of using multivalued possibilities mappings, he uses downwards and upwards simulation relations, thus providing a complete method for checking refinement between boundedly-nondeterministic I/O-automata.

In the *interacting processes* (IP) formalism of Francez and Forman [14], reactive systems are modelled as parallel state-based processes that interact through synchronised multi-party interactions. The language for describing processes is essentially Dijkstra's guarded command language augmented with multi-party interactions. In IP, each process is described as a single program with interactions embedded in it, whereas in our approach, a process is modelled as a set of actions with each action corresponding to an interaction. The semantics for IP is operational.

Later stages of system development involve implementation in programming languages. Action systems cannot be written directly in any programming language, since they contain no explicit flow of control, but rather any action may be executed if it is enabled. However, it should be possible to develop proof rules for implementing action systems as sequential programs in languages that provide synchronised communication such as occam [20] and Ada [36]. The designers of the specification language SL_0 [31] have achieved a similar aim. In SL_0, a combination of CSP algebraic-notation and the transition-system approach of [15,22] is used to specify a communicating system. A set of compositional proof-rules are provided for transforming SL_0 specifications into occam programs. Back & Sere [7] have also investigated the transformation of action systems into occam programs.

13 Conclusions

The definitions and rules presented here may be used to decompose action systems into distributed implementations in which interaction is based on synchronised value-passing. Because the failures-divergences semantics of action systems is defined in terms of weakest preconditions, the specification statements and refinement rules of the refinement calculus may be readily used in our approach. In order to decompose an action system into parallel sub-systems, we refine the state variables so that they may be partitioned amongst the sub-systems, and introduce internal actions representing interaction between sub-systems.

Unlike the style of concurrent composition of state-based systems used by Abadi and Lamport, etc [2,5,11,12,18], where interaction between subcomponents is based on shared state, in our case, interaction is based solely on shared actions representing synchronised value-passing. One important consequence of this is that parallel components of a system can be refined and decomposed separately without making any assumptions about the rest of the system (see Rule 26). So, although our approach cannot be used for as large a range of applications as that of [2], it is simpler to use since we don't have to carry around assumptions about a component's environment. The examples presented here suggest that our approach can be applied to a reasonably large class of distributed systems. Another feature of our approach is that the simulation rule distinguishes internal and external nondeterminism, so that our simulation rule relates less action systems than the simulation rules presented in [21,24].

When using the CSP algebraic notation, explicit state is described by parameterising processes, For example, an ordered buffer may be described as follows:

$$Buffer(\langle\rangle) \;\hat{=}\; (in?x \rightarrow Buffer(\langle x\rangle))$$

$$Buffer(\langle x\rangle^\frown s) \;\hat{=}\; (in?y \rightarrow Buffer(\langle x\rangle^\frown s^\frown\langle y\rangle)\ []\ (out!x \rightarrow Buffer(s)).$$

With this approach, event behaviour and state behaviour are refined separately, whereas in our approach, refinement is more uniform. Also, in the CSP algebraic approach, communications events for the same channel may appear in more than one place in a process description which complicates reasoning about parallel composition. In contrast, with our approach, all communications on a channel are represented by a single action, and when composing systems, we only have to compose pairs of corresponding actions, which is just a syntactic operation. However, in the CSP algebraic approach, it is easy to describe sequential ordering of events, whereas with action systems, it is necessary to introduce some form of program counter variable and encode the sequential ordering into the actions.

The ability to use specification statements to describe actions means that our approach is related to the B [3], VDM [19], and Z [34] methods. In these methods, *operations* may be described using predicates on the initial and final values of state variables, just like specification statements. A system is specified by a set of state variables and a set of operations on those variables, which is the same as the structure of an action system. Our approach provides a means of composing such systems using synchronisation-based parallelism, thus supporting the use of B, VDM, and Z in the development of certain forms of distributed system.

Acknowledgements

This work is based on my D.Phil. thesis undertaken at the Programming Research Group in Oxford. Thanks to my supervisor, Carroll Morgan, for his constant support while I was working on the D.Phil. Thanks to Broadcom Éireann Research Ltd., Ireland for funding my D.Phil. This paper was mostly written while I was a postdoctoral research fellow at Åbo Akademi University, Finland supported by the Academy of Finland IRENE Project and the EU Human Capital Mobility Fund.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. and Sys.*, 17(3):507–534, 1995.

[3] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* To be published by Cambridge University Press, 1996.

[4] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[5] R.J.R. Back. Refinement of parallel and reactive programs. In *Marktoberdorf International Summer School — Program Design Calculi*, July 1992.

[6] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[7] R.J.R. Back and K. Sere. Deriving an occam implementation of action systems. In C.C. Morgan and J.C.P. Woodcock, editors, *3rd BCS-FACS Refinement Workshop.* Springer–Verlag, 1990.

[8] M.J. Butler. *A CSP Approach To Action Systems.* D.Phil. Thesis, Programming Research Group, Oxford University, 1992.

[9] M.J. Butler. Refinement and decomposition of value-passing action systems. In E. Best, editor, *CONCUR'93*, volume LNCS 715. Springer–Verlag, 1993.

[10] M.J. Butler and C.C. Morgan. Action systems, unbounded nondeterminism, and infinite traces. *Formal Aspects of Comput.*, 7:37–53, 1995.

[11] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison–Wesley, 1988.

[12] P. Collette. Composition of assumption-commitment specifications in a UNITY style. *Sci. Comp. Prog.*, 23:107–125, 1994.

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

[14] N. Francez and I. Forman. *Interacting Processes - A multiparty approach to coordinated distributed programming.* Addison–Wesley/ACM Press, 1996.

[15] J. He. Process refinement. In J. McDermid, editor, *The Theory and Practice of Refinement.* Butterworths, 1989.

[16] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *European Symposium on Programming*, volume LNCS 213. Springer–Verlag, 1986.

[17] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[18] C.B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing '83.* IFIP, North Holland, 1983.

[19] C.B. Jones. *Systematic Software Development using VDM.* Prentice–Hall, 1986.

[20] G. Jones and M. Goldsmith. *Programming in occam 2.* Prentice–Hall, 1988.

[21] B. Jonsson. On decomposing and refining specifications of distributed systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume LNCS 430. Springer–Verlag, 1990.

[22] M.B. Josephs. A state-based approach to communicating sequential processes. *Distrib. Comput.*, 3:9–18, 1988.

[23] N.A. Lynch. Multivalued possibilities mappings. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume LNCS 430. Springer–Verlag, 1990.

[24] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[25] C.C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3), 1988. Reprinted in [28].

[26] C.C. Morgan. Of wp and CSP. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer–Verlag, 1990.

[27] C.C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.

[28] C.C. Morgan, K.A. Robinson, and P.H.B. Gardiner. *On the Refinement Calculus*. Technical Monograph PRG–70, Programming Research Group, Oxford University, October 1988.

[29] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.

[30] J.M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[31] E.-R. Olderog. Towards a design calculus for communicating programs. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings CONCUR '91*, volume LNCS 527. Springer-Verlag, 1991.

[32] O. Ore. *Theory of Graphs*, volume XXXVIII of *American Math. Soc. Colloquium Publications*. American Mathematical Society, 1962.

[33] A.W. Roscoe. Unbounded nondeterminism in CSP. *J. Logic Computat.*, 3(2):131–172, 1993.

[34] J.M. Spivey. *The Z Notation - A Reference Manual*. Prentice–Hall, 1989.

[35] R. Udink and J. Kok. Two fully-abstract models for UNITY. In E. Best, editor, *CONCUR'93*, volume LNCS 715. Springer–Verlag, 1993.

[36] D.A. Watt, B.A. Wichmann, and W. Findlay. *ADA: Language and Methodology*. Prentice–Hall, 1987.

[37] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90*, volume LNCS 428. Springer–Verlag, 1990.