An Approach to the Design of Distributed Systems with B AMN (Extended Version)[*†]

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

December 18, 1996

Abstract

In this paper, we describe an approach to the design of distributed systems with B AMN. The approach is based on the action-system formalism which provides a framework for developing state-based parallel reactive systems. More specifically, we use the so-called CSP approach to action systems in which interaction between subsystems is by synchronised message passing and there is no sharing of state. We show that the abstract machines of B may be regarded as action systems and show how reactive refinement and decomposition of action systems may be applied to abstract machines. The approach fits in closely with the stepwise refinement method of B. We illustrate the approach by the abstract specification of an email service as a single machine and it's subsequent refinement into a store-and-forward network.

1 Introduction

In the B method [1], a system is specified as an abstract machine consisting of some state and some operations acting on that state. This is essentially the same structure as an action system [5] which describes the behaviour of a parallel reactive system in terms of the atomic actions (i.e., operations) that can take place during its execution. The operations of both B machines and action systems are described using notations based on Dijkstra's guarded command language [11]. As with B machines, action systems may be refined in a stepwise manner. Techniques for refining the atomicity of operations and for composing systems in parallel have been developed for action systems and such techniques are important for the development of parallel/distributed systems.

Different views as to what constitutes the observable behaviour of an action system may be taken. In the state-based view, the evolution of the state during execution is observable but not the identity of the operations that cause the state transitions. In the event-based view, the execution of an operation is regarded as an event, but only the the identity of the event is observable and the state is regarded as being internal and not observable. The event-based view corresponds to the way in which system behaviour is modelled in various process algebras such as ACP [7], CCS [14] and CSP [12]. An exact correspondence between action systems and CSP was made by Morgan [15]. Using this correspondence, techniques for event-based refinement and parallel composition of action systems have been developed in [8, 9]. In this paper, we shall use the event-based view of action systems, applying the techniques of [8, 9] to B abstract machines. For a description of the state-based view of action systems see [6].

We illustrate the approach by the design of a distributed email system. The design starts with the abstract specification of an email service as a single machine with a simple state structure. The goal is to

[*] A shorter version of this paper appears in the proceedings of the 10th International Conference of Z Users (ZUM'97), 3-4th April 1997, University of Reading, UK.

[†] Support from the Defence Research Ageny is acknowledged.

MACHINE M
SETS SS
CONSTANTS C
PROPERTIES P
VARIABLES v
INVARIANT I
INITIALISATION $init$
OPERATIONS
  $y_1 \longleftarrow op_1(x_1) \;\hat{=}\; S_1$
  $y_2 \longleftarrow op_2(x_2) \;\hat{=}\; S_2$
  $\ldots$
END

Figure 1: Abstract machine outline.

implement this abstract service on a store-and-forward network, where not all nodes are directly connected and messages may pass through a number of intermediate nodes before reaching their recipient. The first refinement of the abstract machine involves the introduction of data structures more closely resembling the store-and-forward architecture. Subsequent design steps involve the parallel decomposition of the system into subsystems representing the agents at each node in the network and the direct communications links in network.

The paper is organised as follows: In Sections 2 and 3 respectively, we show how action systems may be described using B AMN and compare action system refinement with refinement in B. In Section 4, we describe the notion of internal actions that are outside the control of a system's environment and show how such actions may be introduced to a system in a refinement step. In Section 5, we describe a technique for combining action systems in parallel in which interaction between subsystems is based on shared actions. The development of the email system is described in Section 6. Section 7 outlines the correspondence between CSP and the event-based view of action systems.

2 Abstract Machines and Actions Systems

B Abstract Machine Notation (AMN)

B AMN is a model-oriented formal notation and is part of the B-method developed by Abrial [1]. A system in B is specified as an abstract machine which has the form outlined in Fig. 1. An abstract machine consists of some sets (SS), constants (C) and variables (v) which are modelled using standard set-theoretic constructs. Properties (P) and invariants (I) are first-order predicates. Operations act on the variables while preserving the invariant and have input parameters (x_i) and output parameters (y_i). The initialisation ($init$) and operations (S_i) are written in the generalised substitution notation of B AMN which includes constructs such as *assignment* ($v := E$), *precondition statements* (PRE Q THEN S END), *guarded statements* (SELECT Q THEN S END) and *unbounded choice* (ANY x WHERE Q THEN S END).

The semantics of generalised substitutions is defined by weakest-precondition formulae: for statement S and postcondition P, the formula $[S]P$ (weakest precondition of S w.r.t. P) characterises those initial states from which S is guaranteed to terminate in a state satisfying P. The semantics of several AMN constructs are specified in Fig. 2. Note that this is only a subset of the full language.

Actions and Guarding

Actions will be specified as statements in the generalised substitution notation of B AMN. For action S, the formula $[S]\,false$ represents those initial states from which S is guaranteed to establish any postcondition; to see this, we have that for any predicate P,

$$false \;\Rightarrow\; P,$$

$$
\begin{array}{rcl}
[v := E]P & \hat{=} & P[E/v] \\
[\text{PRE } Q \text{ THEN } S \text{ END}]P & \hat{=} & Q \wedge [S]P \\
[\text{SELECT } Q \text{ THEN } S \text{ END}]P & \hat{=} & Q \Rightarrow [S]P \\
[\text{ANY } x \text{ WHERE } Q \text{ THEN } S \text{ END}]P & \hat{=} & (\forall x \bullet Q \Rightarrow [S]P) \\
[S;\, T]P & \hat{=} & [S]([T]P)
\end{array}
$$

Figure 2: AMN semantics.

and, since $[S]$ is monotonic, we get

$$[S]\,\mathit{false} \quad \Rightarrow \quad [S]P.$$

That is, S behaves miraculously in an initial state satisfying $[S]\,\mathit{false}$ since it can establish any postcondition P. For example, the statement

$$\text{SELECT } \mathit{false} \text{ THEN } T \text{ END}$$

is miraculous in any initial state since

$$[\text{SELECT } \mathit{false} \text{ THEN } T \text{ END}]\,\mathit{false} \quad = \quad \mathit{true}.$$

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

$$gd(S) \;\hat{=}\; \neg\,([S]\,\mathit{false}).$$

From this we get the following rules for calculating the guards of guarded statements, unbounded choice statements and assignment statements:

$$
\begin{array}{rcl}
gd(\text{ SELECT } G \text{ THEN } S \text{ END }) & = & G \wedge gd(S) \\
gd(\text{ ANY } x \text{ WHERE } P \text{ THEN } S \text{ END }) & = & (\exists x \cdot P \wedge gd(S)) \\
gd(\ x := E\ ) & = & \mathit{true}.
\end{array}
$$

For example, we get

$$
\begin{array}{rl}
& gd(\text{ ANY } x \text{ WHERE } x \in a \text{ THEN } a := a \setminus \{x\} \text{ END }) \\
= & (\exists x \cdot x \in a \wedge \mathit{true}) \\
= & a \neq \{\},
\end{array}
$$

which means this statement is enabled only when $a \neq \{\}$.

Note the difference between PRE and SELECT statements: a PRE statement aborts (i.e., is not guaranteed to terminate) when Q is not satisfied, while a SELECT statement is disabled and hence cannot be executed when Q is not satisfied.

Action Systems

An action system consists of some state variables, a set of actions, each with its own unique name, and an initialisation statement. An action system proceeds by firstly executing the initialisation. Then, repeatedly, an enabled action is selected and executed. An action system deadlocks if no action is enabled, and diverges (behaves chaotically) whenever some action aborts.

Fig. 3 contains an action system, called *VM1*, specified as a B abstract machine. This is intended to represent a simple vending machine. The state of the machine is represented by the variable n. The

MACHINE

$\quad VM1$

VARIABLES

$\quad n$

INVARIANT

$\quad n \in \{0, 1\}$

INITIALISATION

$\quad n := 0$

OPERATIONS

$\quad coin \;\hat{=}\; \text{SELECT } n = 0 \text{ THEN } n := 1 \text{ END}$

$\quad choc \;\hat{=}\; \text{SELECT } n = 1 \text{ THEN } n := 0 \text{ END}$

END

Figure 3: Simple vending machine.

machine has two actions represented by the operations *coin* and *choc*. Initially n is set to 0 so that only the *coin* action is enabled. When the *coin* action is executed, n is set to 1, and only the *choc* action is enabled. Execution of the *choc* action then results in *coin* being enabled again and so on. Thus *VM1* describes a system that alternatively engages in an *coin* action then a *choc* action forever.

As mentioned already, we are taking an event-based view of action systems. This means that the environment of an abstract machine can only interact with the machine through its actions and has no direct access to a machine's state. The environment of a machine can also control the execution of actions by blocking them. This will be seen clearly in Section 5, where parallel composition of action systems is described.

For any abstract machine M, we write $\alpha(M)$ for the set of action names in M. For example,

$$\alpha(VM1) \;=\; \{\, coin, choc \,\}.$$

We write $M.a$ for the action named a in machine M. For example,

$$VM1.coin \;=\; \text{SELECT } n = 0 \text{ THEN } n := 1 \text{ END}.$$

Parameter Passing

The actions of an action system can be *input* actions, with associated input parameters, or *output* actions, with associated output parameters. An input action will be represented by a B AMN operation of the form

$$name(x) \;\hat{=}\; S,$$

where x represents the input parameter(s). An input action models a channel through which a machine is willing to accept an input value whenever that action is enabled. An output action will be represented by a B AMN operation of the form

$$y \longleftarrow name \;\hat{=}\; S,$$

MACHINE

Buffer1

VARIABLES

s

INVARIANT

$s \in seq\ T$

INITIALISATION

$s := \langle\rangle$

OPERATIONS

$left(x) \;\hat{=}\;$ SELECT $x \in T$ THEN $s := s \frown \langle x \rangle$ END

$y \longleftarrow right \;\hat{=}\;$ SELECT $s \neq \langle\rangle$ THEN $y, s := head(s), tail(s)$ END

END

Figure 4: Ordered buffer.

where y represents the output parameter(s). An output action models a channel through which a machine is willing to deliver an output value whenever that action is enabled.

We shall assume that no action can be both an input action and and output action. See Section 8 for a discussion of this issue.

Fig. 4 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.

3 Refinement

Specification machines usually contain abstract data structures that are not directly implementable in a programming language. *Data refinement* is used in order to bring abstract specifications towards implementations by replacing abstract variables with concrete variables that are more easily implemented.

Data Refinement of Actions

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 a statement that acts on variables a, T is a statement that acts on variables c, and AI is an abstraction invariant then we write

$$S \sqsubseteq_{AI} T$$

for "S is data-refined by T under abstraction invariant AI".

The weakest-precondition definition of data refinement [1, 3, 16, 17] is as follows:

Definition 1 (Data Refinement) *$S \sqsubseteq_{AI} T$ iff for each postcondition P independent of concrete variable c,*

$$AI \wedge\ [S]P \quad \Rightarrow \quad [T](\exists a \bullet AI \wedge P).$$

Aside: Definition 1 involves quantification over predicates (P). Abrial [1] shows that the following conditions, which do not involve quantification over predicates, are sufficient to ensure data refinement:

$$\begin{array}{lcl} AI \wedge pre(S) & \Rightarrow & pre(T) \\ AI \wedge pre(S) & \Rightarrow & [T](\neg [S] \neg AI). \end{array}$$

Here, $pre(S) \;\hat{=}\; [S]true$, i.e., $pre(S)$ represents the initial condition under which S is guaranteed to terminate. Obligations such as these can be checked using the theorem-proving environments associated with the B-method [1, 18].

Data Refinement of Initialisations

$S \sqsubseteq_{AI} T$ means that T data-refines S in such a way that the abstraction invariant AI is preserved from initial to final states. However, no assumption is made about the state of action systems before initialisation. This means that a concrete initialisation must refine an abstract initialisation in such a way that it establishes the abstraction invariant rather than simply preserving it. We write $S \sqsubseteq_{AI}^{init} T$ to deal with this, where

Definition 2 (Data Refinement of Initialisations) *$S \sqsubseteq_{AI}^{init} T$ if for each postcondition P independent of concrete variable c,*

$$[S]P \quad \Rightarrow \quad [T](\exists a \bullet AI \wedge P).$$

Simulation

A *simulation* is a relation between action systems M and N with the same alphabet but possibly different state-spaces:

Definition 3 (Simulation) *For abstract action system M and concrete action system N, where $\alpha(M) = \alpha(N)$, M is simulated by N with abstraction invariant AI, denoted $M \sqsubseteq_{AI} N$, provided each of the following conditions hold:*

1. $M.init \sqsubseteq_{AI}^{init} N.init$
2. $M.a \sqsubseteq_{AI} N.a, \qquad$ *each* $a \in \alpha(M)$
3. $AI \wedge gd(M.a) \Rightarrow gd(N.a), \qquad$ *each* $a \in \alpha(M)$.

Conditions 1 and 2 ensure that each action of N is a refinement of its counterpart in M, and are referred to as data-refinement conditions. These are precisely the conditions that define refinement of machines in B AMN [1]. Condition 3 ensures that N may only refuse an action when M may refuse it, and is referred to as a progress condition. Intuitively, $M \sqsubseteq_{AI} N$ means that any observable behaviour of N is also an observable behaviour of M. See Section 7 for a more precise definition of what this means.

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 is 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 *UBuffer1* of Fig. 5 describes an unordered buffer that communicates values of type T. The initialisation statement of *UBuffer1* sets 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, removes it from a and outputs it as y.

It can be shown that *UBuffer1* is refined by *Buffer1* of Fig. 4. As an abstraction invariant, we use

$$AI \quad \hat{=} \quad a = bag(s),$$

MACHINE

$\quad UBuffer1$

VARIABLES

$\quad a$

INVARIANT

$\quad a \in bag\ T$

INITIALISATION

$\quad a := \prec\succ$

OPERATIONS

$$left(x) \ \hat{=} \ \text{SELECT } x \in T \text{ THEN } a := a + \prec x \succ \text{ END}$$

$$y \longleftarrow right \ \hat{=} \ \text{ANY } y' \text{ WHERE } y' \in a \text{ THEN } a, y := a - \prec y' \succ, y' \text{ END}$$

END

Figure 5: Unordered buffer.

where $bag(s)$ represents the bag of elements in sequence s. The proof obligations generated by Definition 3 are as follows:

- $UBuffer1.init \ \sqsubseteq_{AI}^{init} \ Buffer1.init$
- $UBuffer1.left \ \sqsubseteq_{AI} \ Buffer1.left$
- $UBuffer1.right \ \sqsubseteq_{AI} \ Buffer1.right$
- $AI \wedge gd(UBuffer1.left) \ \Rightarrow \ gd(Buffer1.left)$
- $AI \wedge gd(UBuffer1.right) \ \Rightarrow \ gd(Buffer1.right)$

4 Internal Actions

In this section, action systems are extended to include internal actions. Internal actions are not visible to the environment of a machine and are thus outside the control of the environment. Any number of executions of an internal action may occur in between each execution of a visible action. If the action system reaches a state where internal actions can be executed infinitely, then the action system diverges. Internal actions do not have input or output parameters.

An example of an action system with internal actions is given in Fig. 6. *UBuffer2* 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, *UBuffer2* 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.

We write $\beta(M)$ for the set of internal actions in system M.

MACHINE

$\quad$ *UBuffer2*

VARIABLES

$\quad b, c$

INVARIANT

$\quad b \in bag\ T \ \wedge\ c \in bag\ T$

INITIALISATION

$\quad b, c := \prec\succ, \prec\succ$

OPERATIONS

$\quad left(x) \ \hat{=}\ $ SELECT $x \in T$ THEN $b := b + \prec x \succ$ END

$\quad y \longleftarrow right \ \hat{=}\ $ ANY y' WHERE $y' \in c$ THEN $c, y := c - \prec y' \succ, y'$ END

INTERNAL OPERATIONS

$\quad mid \ \hat{=}\ $ ANY z WHERE $z \in b$ THEN $b, c := b - \prec z \succ, c + \prec z \succ$ END

END

Figure 6: Unordered buffer with internal action.

Refinement with Internal Actions

Intuitively it can be seen that *UBuffer2* behaves in the same way as *UBuffer1* of Fig. 5. We shall introduce a proof rule that allows us to verify that *UBuffer1* $\sqsubseteq$ *UBuffer2*. 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:

Definition 4 *Let M and N be action systems where $\alpha(M) = \alpha(N)$ and $\beta(M) = \{\}$. M is simulated by N with abstraction invariant AI, well-founded set WF, and variant E, denoted $M \sqsubseteq_{(AI,WF,E)} N$, provided each of the following conditions hold:*

1. $M.init \ \sqsubseteq_{AI}^{init}\ N.init$
2. $M.a \ \sqsubseteq_{AI}\ N.a, \qquad$ *each* $a \in \alpha(M)$
3. $skip \ \sqsubseteq_{AI}\ N.h, \qquad$ *each* $h \in \beta(N)$
4. $AI \ \Rightarrow\ E \in WF$
5. $AI \wedge E = e \ \Rightarrow\ [N.h](E < e), \qquad$ *each* $h \in \beta(N)$

6. $AI \wedge gd(M.a) \;\Rightarrow\; gd(N.a) \vee (\exists h \in \beta(N) \bullet gd(N.h)), \qquad each\ a \in \alpha(M)$

Conditions 1, 2, and 3 are data-refinement conditions. Conditions 1 and 2 are the same as in Definition 3. Condition 3 ensures that each internal action of N causes no change to the corresponding abstract state. Conditions 4 and 5 are referred to as non-divergence conditions. Condition 4 ensures that the variant E is an element of WF, while Condition 5 ensures that the internal actions of N always decrease E when executed. Together, Conditions 4 and 5 ensure that the internal actions of N are eventually disabled and so cannot introduce divergence. Condition 6 is a progress condition and ensures that, whenever an action of M is enabled, either the corresponding action of N is enabled, or some internal action of N is enabled.

Example

To show that $UBuffer1 \sqsubseteq UBuffer2$, we use the abstraction invariant

$$AI \quad \hat{=} \quad a = b + c.$$

We use the size of bag b, written $\#b$, as a variant, with $\mathbb{N}$ as a well-founded set. Note that *UBuffer2.mid* is a refinement of *skip* under this abstraction invariant since the bag sum $b+c$ is unchanged by its execution. Also *UBuffer2.mid* decreases the variant $\#b$.

Hiding Operator

Let M be an action system, and C be a set of operation names, with $C \subseteq \alpha(M)$. We write $M \backslash C$ for the machine M with each action named in C converted into an internal action. The input/output parameters of an internalised action should be localised using the VAR $x \bullet S$ END construct of B AMN [1]. Note that action hiding is simply a syntactic transformation of M.

Action hiding is monotonic: if M is refined by N, then $M \backslash C$ is refined by $N \backslash C$.

5 Parallel Composition

In this section, we describe a parallel composition operator for action systems. The parallel composition of action systems M and N is written $M \parallel N$. M and N must not have any common state variables. Instead they interact by synchronising over shared actions (i.e., actions with common names). They may also pass values on synchronisation. We look first at basic parallel composition and later look at parallel composition with value passing.

Basic Parallel Composition of Actions

To achieve the synchronisation effect, shared actions are 'fused' using a parallel operator for actions $(S \parallel T)$. This operator satisfies the following properties:

- $x := E \;\parallel\; y := F \quad = \quad x, y := E, F$
- SELECT G THEN S END $\parallel$ SELECT H THEN T END $=$
  SELECT $G \wedge H$ THEN $S \parallel T$ END.

The parallel operator models simultaneous execution of statements and the composite action is enabled exactly when both component actions are enabled. Note that the variables assigned to by constituent actions must be independent. The weakest-precondition semantics of this operator is described in [1, 4].

Basic Parallel Composition of Action Systems

The parallel composition of action systems M and N is an action system constructed by fusing shared actions of M and N and leaving independent actions independent. The state variables of the composite system $M \parallel N$ are simply the union of the variables of M and N.

MACHINE

N1

VARIABLES

m

INVARIANT

$m \in \{0,1\}$

INITIALISATION

$m := 0$

OPERATIONS

$a \;\hat{=}$
SELECT
$m = 0$
THEN
$m := 1$
END

$c \;\hat{=}$
SELECT
$m = 1$
THEN
$m := 0$
END

END

MACHINE

N2

VARIABLES

n

INVARIANT

$n \in \{0,1\}$

INITIALISATION

$n := 0$

OPERATIONS

$b \;\hat{=}$
SELECT
$n = 0$
THEN
$n := 1$
END

$c \;\hat{=}$
SELECT
$n = 1$
THEN
$n := 0$
END

END

Figure 7: Action systems with common actions.

As an illustration of this, consider *N1* and *N2* of Fig. 7. *N1* alternates between an a-action and a c-action, while *N2* alternates between an b-action and a c-action. The system *N1* || *N2* is shown in Fig. 8. The a- and b-actions of *N1* || *N2* come directly from *N1* and *N2* respectively, while the c-action is the fusion of the c-actions of *N1* and *N2*. The initialisations of *N1* and *N2* are also fused to form the initialisation of *N1* || *N2*. The effect of *N1* || *N2* is that, repeatedly, the a- or the b-actions can occur in either order, then both systems must synchronise on the c-action.

Parallel Composition with Value-Passing

We extend the parallel operator to deal with parameterised actions and value-passing. An output action from one system is composed with a similarly labelled input action form another in such a way that the output value generated by the first is passed on as the input value for the second. For example, given an output action of the form

$$y \longleftarrow name \;\hat{=}\; \text{SELECT } G \text{ THEN } u, y := U, Y \text{ END}$$

and an output action if of the form

$$name(x) \;\hat{=}\; \text{SELECT } x \in A \wedge H \text{ THEN } v := F(x) \text{ END},$$

MACHINE

$\quad N1 \,||\, N2$

VARIABLES

$\quad m, n$

INVARIANT

$\quad m, n \in \{0, 1\}$

INITIALISATION

$\quad m, n := 0, 0$

OPERATIONS

$$a \;\hat{=}\; \text{SELECT } m = 0 \text{ THEN } m := 1 \text{ END}$$

$$b \;\hat{=}\; \text{SELECT } n = 0 \text{ THEN } n := 1 \text{ END}$$

$$c \;\hat{=}\; \text{SELECT } m = 1 \wedge n = 1 \text{ THEN } m, n := 0, 0 \text{ END}$$

END

Figure 8: Parallel composition of action systems.

their value-passing fusion is represented as:

$$y \longleftarrow \mathit{name} \;\hat{=}\; \text{SELECT } H \wedge G \text{ THEN } u, y, v := U, Y, F(Y) \text{ END}.$$

Notice how $F(x)$ becomes $F(Y)$, modelling the passing of the output value from the output action to the input action. Notice also that the fused action is itself an output action. This allows us to fuse further input actions with the fused action, thereby modelling broadcast communications.

More generally, let $M.\mathit{name}$ be an output action of machine M and $N.\mathit{name}$ be an input action of N. We shall assume[1] that $M.\mathit{name}$ has the form:

$$y \longleftarrow \mathit{name} \;\hat{=}\; \text{ANY } u', y' \text{ WHERE } P \text{ THEN } u, y := u', y' \text{ END}$$

and that $N.\mathit{name}$ has the form:

$$\mathit{name}(x) \;\hat{=}\; \text{SELECT } x \in A \wedge H \text{ THEN } v := F(x) \text{ END},$$

where H is independent of x. The value-passing fusion of these two action is defined by:

Definition 5 (Value-passing Fusion)

$$y \longleftarrow \mathit{name} \;\hat{=}\; \text{ANY } u', y' \text{ WHERE } P \wedge H \text{ THEN } u, y, v := u', y', F(y') \text{ END}.$$

Furthermore, the composition of $M.name$ an $N.name$ is only permitted provided

$$I_M \;\Rightarrow\; [M.\mathit{name}](y \in A),$$

where M is the system to which the output action belongs and I_M is the invariant of M.

[1] We only make these assumptions on actions that are to be composed in parallel with other actions.

This restriction ensures that the output value generated by the output action is always acceptable by the input action. Since this restriction guarantees that $y \in A$, the predicate $x \in A$ in the guard of the input action is dropped from the composite action.

The composition of two systems M and N is then constructed by fusing commonly named input-output pairs of actions as described by Definition 5. As before, independently named actions remain independent. The fusion of input-input pairs of actions is also permitted: assume $M.name$ has the form

$$name(x) \;\hat{=}\; \text{SELECT } x \in A \wedge G \text{ THEN } u := F(x) \text{ END},$$

and that $N.name$ has the form:

$$name(x) \;\hat{=}\; \text{SELECT } x \in B \wedge H \text{ THEN } v := G(x) \text{ END},$$

The fusion of these two action is defined by:

Definition 6

$$name(x) \;\hat{=}\; \text{SELECT } x \in (A \cap B) \wedge G \wedge H \text{ THEN } u, v := F(x), G(x) \text{ END}.$$

Fusion of output-output pairs of actions is not permitted. This avoids the introduction of deadlock in situations where two output actions are not willing to output the same value.

Fig. 9 describes the action systems *UBufferL* and *UBufferR*. *UBufferL* is simply an unbounded buffer with *right* renamed to *mid*, while *UBufferR* has *left* renamed to *mid*. When *UBufferL* and *UBufferR* are placed in parallel, they interact via the *mid* channel, with values being passed from *UBufferL* to *UBufferR*. This can be seen by constructing the composite action system *UBufferL* || *UBufferR* as described above (see Fig. 10). The only proof obligation (from Definition 5) associated with this composition is that the *UBufferL.mid* is guaranteed to output a value of type T, i.e.,

$$b \in bag\, T \;\Rightarrow\; [UBufferL.mid](y \in T).$$

If the *mid* action of *UBufferL* || *UBufferR* is hidden, then the resultant action system is the same as *UBuffer2* of Fig. 6:

$$UBuffer2 \quad = \quad (UBufferL \,||\, UBufferR) \backslash \{mid\}.$$

Now, since $UBuffer1 \sqsubseteq UBuffer2$, we have that:

$$UBuffer1 \quad \sqsubseteq \quad (UBufferL \,||\, UBufferR) \backslash \{mid\}.$$

Design Technique

The derivation of the system $(UBufferL \,||\, UBufferR) \backslash \{mid\}$ illustrates a design technique that may be used to decompose an action system into parallel subsystems: refine the state variables so that they may be partitioned amongst the subsystems, introducing internal actions representing interaction between subsystems, then partition the system into subsystems using the parallel operator in reverse. The refinement of the single system can always be performed in a number of steps rather than a single step.

Most importantly, the parallel composition of action systems is monotonic: if M is refined by M' and N is refined by N', then $M \,||\, N$ is refined by $M' \,||\, N'$. This means that when we decompose a system into parallel subsystems, the subsystems may be refined and decomposed independently.

6 Case Study: Email System

The action systems of this case study contain replicated 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 a replicated set of input channels $\{\ i.in \mid i \in F\ \}$, with associated actions, the following notation is used:

$$\text{REPLICATE}\, i \;\mid\; i \in F \bullet\; i.in(x) \;\hat{=}\; S_i.$$

```
MACHINE
    UBufferL
VARIABLES
    b
INVARIANT
    b ∈ bag T
INITIALISATION
    b := ≺≻
OPERATIONS
    left(x) ≙
        SELECT
            x ∈ T
        THEN
            b := b + ≺x≻
        END

    y ⟵ mid ≙
        ANY y′ WHERE
            y′ ∈ b
        THEN
            b, y := b − ≺y′≻, y′
        END
END
```

```
MACHINE
    UBufferR
VARIABLES
    c
INVARIANT
    c ∈ bag T
INITIALISATION
    c := ≺≻
OPERATIONS
    y ⟵ right ≙
        ANY y′ WHERE
            y′ ∈ c
        THEN
            c, y := c − ≺y′≻, y′
        END

    mid(x) ≙
        SELECT
            x ∈ T
        THEN
            c := c + ≺x≻
        END
END
```

Figure 9: Buffers.

The intention is that the i-indexed statement represents a set of input actions. A replicated set of output channels is written:

$$\mathrm{REPLICATE}\, i \mid i \in F \bullet y \longleftarrow i.out \;\hat{=}\; S_i.$$

When a replicated input action such as $i.in(x)$ is internalised, it is collapsed into a single parameterless statement by transforming it to

$$left \;\hat{=}\; \text{ANY } i, x \text{ WHERE } i \in F \text{ THEN } S_i \text{ END}.$$

Similarly for a replicated output action.

Abstract Specification

We suppose that an email service 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*. 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*.

MACHINE

$UBufferL \mid\mid UBufferR$

VARIABLES

b, c

INVARIANT

$b \in bag\ T \ \wedge \ c \in bag\ T$

INITIALISATION

$b, c := \prec\succ, \prec\succ$

OPERATIONS

$left(x) \ \hat{=}$
SELECT
$x \in T$
THEN
$b := b + \prec x \succ$
END

$y \longleftarrow right \ \hat{=}$
ANY y' WHERE
$y' \in c$
THEN
$c, y := c - \prec y' \succ, y'$
END

$y \longleftarrow mid \ \hat{=}$
ANY y' WHERE
$y' \in b$
THEN
$b, c, y := b - \prec y' \succ, c + \prec y' \succ, y'$
END

END

Figure 10: Parallel buffers.

The initial specification of the email service, *MailSys1*, is given in Figure 11. Variable *mail* contains all messages sent but not yet received. Initially *mail* is empty. For each node n, there is a *send* action and a *receive* action. Action $s.send$ 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 $r.receive$ chooses one of these messages and outputs it.

First Refinement of MailSys

Our goal is to implement *MailSys1* 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.

MACHINE

MailSys1

SETS

$NODE, MESS,$
$ENV = NODE \times MESS$

VARIABLES

mail

INVARIANT

$mail \in bag\ ENV$

INITIALISATION

$mail := \prec\succ$

OPERATIONS

$\mathrm{REPLICATE}\ s \mid s \in NODE \bullet$
$s.send(r,m) \ \hat{=}$
SELECT
$(r,m) \in ENV$
THEN
$mail := mail + \prec(r,m)\succ$
END

$\mathrm{REPLICATE}\ r \mid r \in NODE \bullet$
$m \longleftarrow r.receive \ \hat{=}$
ANY n WHERE
$(r,n) \in mail$
THEN
$mail := mail - \prec(r,n)\succ \ \| \ m := n$
END

END

Figure 11: Electronic mail service.

MACHINE

Routing

SETS

TAG

CONSTANTS

$net, route$

PROPERTIES

$net \in NODE \leftrightarrow NODE \ \wedge$
$route \in TAG \rightarrow (NODE \leftrightarrow NODE) \ \wedge$
$\mathrm{ran}\, route \ \subseteq \ Routes(net) \ \wedge$
$(\cup i \in TAG \bullet route(i)^*) \ = \ NODE \times NODE$

END

Figure 12: Routing Information.

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.

In *MailSys2*, *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. 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 7 *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 $a \neq c$,*

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

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 route R, the elongation to the destination node c decreases.

MailSys2 will use a fixed set of routes, each one uniquely identified by a tag from a set *TAG*. These routes are represented by the constant function

$$route \quad \in \quad TAG \rightarrow Routes(net).$$

In order that each distinct pair of nodes be connected by at least one route, we shall assume that the constant function *route* satisfies:

$$(\cup i \in TAG \bullet route(i)^*) \quad = \quad NODE \times NODE.$$

For convenience, the constants associated with routing are collected in the machine *Routing* of Figure 12

MACHINE

MailSys2

REFINES

MailSys1

SEES

Routing

VARIABLES

$store, link$

INVARIANT

$$
\begin{aligned}
& store \in NODE \rightarrow bag\ (TAG \times ENV)\ \wedge \\
& link \in net \rightarrow bag\ (TAG \times ENV)\ \wedge \\
& (\forall (i,r,m) \in TAG \times ENV;\ a,b \in NODE \bullet \\
& \qquad (i,r,m) \in store(a)\ \Rightarrow\ (a,r) \in route(i)^*\ \wedge \\
& \qquad (a,b) \in net \wedge (i,r,m) \in link(a,b)\ \Rightarrow\ (b,r) \in route(i)^*)\ \wedge \\
& mail = \ (\Sigma a \in NODE \bullet env(store(a))) + (\Sigma (a,b) \in net \bullet env(link(a,b)))
\end{aligned}
$$

VARIANT

$$
\begin{aligned}
& (\Sigma a \in NODE \bullet (\Sigma (i,r,m) \in store(a) \bullet e_i(a,r) * 2)) + \\
& (\Sigma (a,b) \in net \bullet (\Sigma (i,r,m) \in link(a,b) \bullet (e_i(b,r) * 2) + 1))
\end{aligned}
$$

Figure 13: Refined email system.

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. Elongations are used as a variant to ensure that all envelopes eventually reach their destination.

MailSys2 is then specified in Figures 13 and 14. 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 invariant states that there is always a path from the current position of an envelope to its recipient in the assigned route.

All stores and links are initially empty. The action $s.send$ 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 $r.receive$ 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 *MailSys1* is refined by *MailSys2*, which may be checked using Definition 4, as follows:

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 $env(b)$

INITIALISATION

$store := (\lambda\, a \mid a \in NODE \bullet \prec\succ) \;\; ||$
$link := (\lambda\, a, b \mid (a,b) \in net \bullet \prec\succ)$

OPERATIONS

$\mathrm{REPLICATE}\; s \;\mid\; s \in NODE \bullet$
$s.send(r,m) \;\hat{=}$
ANY i WHERE
$(r,m) \in ENV \wedge i \in TAG \wedge (s, r?) \in route(i)^*$
THEN
$store(s) := store(s) + \prec(i,r,m)\succ$
END

$\mathrm{REPLICATE}\; r \;\mid\; r \in NODE \bullet$
$m \longleftarrow r.receive \;\hat{=}$
ANY i, n WHERE
$(i,r,n) \in store(r)$
THEN
$store(r) := store(r) - \prec(i,r,n)\succ \;||\; m := n$
END

INTERNAL OPERATIONS

$forward \;\hat{=}$
ANY a, b, i, r, m WHERE
$(i,r,m) \in store(a) \;\wedge\; r \neq a \;\wedge$
$(a,b) \in route(i) \;\wedge\; (b,r) \in route(i)^*$
THEN
$store(a) := store(a) - \prec(i,r,m)\succ \;\; ||$
$link(a,b) := link(a,b) + \prec(i,r,m)\succ$
END

$relay \;\hat{=}$
ANY a, b, i, r, m WHERE
$(a,b) \in net \;\wedge\; (i,r,m) \in link(a,b)$
THEN
$link(a,b) := link(a,b) - \prec(i,r,m)\succ \;\; ||$
$store(b) := store(b) + \prec(i,r,m)\succ$
END

END

Figure 14: Operations of the refined email system.

is the corresponding bag of untagged envelopes. The abstraction invariant, *AI*, is then the conjunction of the invariants of *MailSys1* and *MailSys2*.

The variant must be shown to be decreased by both *forward2* and *relay2*. We use the elongation from the current position of each envelope to its destination in its route to define the 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 in Figure 13.

Using *AI* and *E*, the simulation conditions of Definition 4 can then be discharged. Note that in order to show that simulation conditions hold between replicated actions

$$\mathrm{REPLICATE}\, i \mid i \in F \bullet\ i.name \quad \text{and} \quad \mathrm{REPLICATE}\, i \mid i \in F \bullet\ i.name',$$

we only need to show that they hold between

$$i.name \quad \text{and} \quad i.name',$$

under the assumption that $i \in F$.

Parallel Decomposition of MailSys

In this step, *MailSys2* is decomposed into two parallel systems, *Agents* and *Media*, specified in Figures 15 and 16. *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 direct 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

$$\begin{aligned} &MailSys2 \ = \\ &\qquad (Agents1 \parallel Media1) \setminus \{\, a.forward \mid a \in NODE \,\} \cup \{\, b.relay \mid b \in NODE \,\}. \end{aligned}$$

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. In order to do this, we first perform a very simple refinement step involving the replacement of the single variable *store* by a replicated set of variables

$$\mathrm{REPLICATE}\, a \mid a \in NODE \bullet\ a.lstore$$

where each $a.lstore$ has type $bag\,(TAG \times ENV)$. In the refined system, *Agents2* (Figure 17), each $a.lstore$ is equated with with $store(a)$. Since *NODE* is finite, the generalised parallel composition of statements used in the initialisation is defined by iterated use of the binary operator.

Each action, $a.name$, of *Agents2* only refers to the variable $a.store$, so that *Agents2* may be partitioned into a set of independent parallel subsystems:

$$Agents2 \ = \ (\parallel a \mid a \in NODE \cdot LocalAgent1(a)),$$

where $LocalAgent1(a)$ is specified in Figure 18. The only statements that are fused in the construction of $(\parallel a \mid a \in NODE \cdot LocalAgent1(a))$ are the initialisations; otherwise the decomposition simply involves the syntactic partitioning of the replicated *lstores* and the replicated actions.

7 CSP Correspondence

In CSP [12], the behaviour of a process is viewed in terms of the events it can engage in. Value-passing is modelled by grouping events into input channels and output channels. Each process P has an alphabet of events A, and its behaviour is modelled by a set of *failures* F and a set of *divergences* D. A failure is a

MACHINE

Agents1

VARIABLES

store

INVARIANT

$store \in NODE \to bag\,(TAG \times ENV)$

INITIALISATION

$store := (\lambda\, a \mid a \in NODE \bullet \prec\succ)$

OPERATIONS

$\mathrm{REPLICATE}\ s \mid s \in NODE \bullet$
$s.send(r,m) \;\hat{=}$
ANY i WHERE
$(r,m) \in ENV \wedge i \in TAG \wedge (s,r?) \in route(i)^*$
THEN
$store(s) := store(s) + \prec(i,r,m)\succ$
END

$\mathrm{REPLICATE}\ r \mid r \in NODE \bullet$
$m \longleftarrow r.receive \;\hat{=}$
ANY i,n WHERE
$(i,r,n) \in store(r)$
THEN
$store(r) := store(r) - \prec(i,r,n)\succ \parallel m := n$
END

$\mathrm{REPLICATE}\ a \mid a \in NODE \bullet$
$(b',i',r',m') \longleftarrow a.forward \;\hat{=}$
ANY b,i,r,m WHERE
$(i,r,m) \in store(a) \;\wedge\; r \neq a \;\wedge$
$(a,b) \in route(i) \;\wedge\; (b,r) \in route(i)^*$
THEN
$store(a) := store(a) - \prec(i,r,m)\succ \;\parallel$
$b',i',r',m' := b,i,r,m$
END

$\mathrm{REPLICATE}\ b \mid b \in NODE \bullet$
$b.relay(a,i,r,m) \;\hat{=}$
SELECT
$a \in NODE \;\wedge\; i \in TAG \;\wedge\; (r,m) \in ENV$
THEN
$store(b) := store(b) + \prec(i,r,m)\succ$
END

END

Figure 15: Network agents.

MACHINE

$Media1$

VARIABLES

$link$

INVARIANT

$link \in net \to bag\,(TAG \times ENV)$

INITIALISATION

$link := (\lambda\, a, b \mid (a, b) \in net \bullet \prec\succ)$

OPERATIONS

$\mathrm{REPLICATE}\ a \mid a \in NODE \bullet$
$a.forward(b, i, r, m) \;\hat{=}$
SELECT
$b \in NODE \;\wedge\; i \in TAG \;\wedge\; (r, m) \in ENV$
THEN
$link(a, b) := link(a, b) + \prec(i, r, m)\succ$
END

$\mathrm{REPLICATE}\ b \mid b \in NODE \bullet$
$(a', i', r', m') \longleftarrow b.relay \;\hat{=}$
ANY a, i, r, m WHERE
$(a, b) \in net \;\wedge\; (i, r, m) \in link(a, b)$
THEN
$link(a, b) := link(a, b) - \prec(i, r, m)\succ \;\|$
$a', i', r', m' := a, i, r, m$
END

END

Figure 16: Network media.

```
MACHINE

    Agents2

REFINES

    Agents1

VARIABLES

    REPLICATE a | a ∈ NODE • a.lstore

INVARIANT

    (∀a | a ∈ NODE · a.lstore = store(a))

INITIALISATION

    (‖ a | a ∈ NODE · a.lstore := ≺≻)

OPERATIONS

    REPLICATE s | s ∈ NODE •
     s.send(r,m) ≙
       ANY i WHERE
         (r,m) ∈ ENV ∧ i ∈ TAG ∧ (s,r?) ∈ route(i)*
       THEN
         s.lstore := s.lstore + ≺(i,r,m)≻
       END


    REPLICATE r | r ∈ NODE •
     m ⟵ r.receive ≙
       ANY i,n WHERE
         (i,r,n) ∈ r.lstore
       THEN
         r.lstore := r.lstore − ≺(i,r,n)≻ ‖ m := n
       END


    REPLICATE a | a ∈ NODE •
     (b′,i′,r′,m′) ⟵ a.forward ≙
       ANY b,i,r,m WHERE
         (i,r,m) ∈ a.lstore ∧ r ≠ a ∧
         (a,b) ∈ route(i) ∧ (b,r) ∈ route(i)*
       THEN
         a.lstore := a.lstore − ≺(i,r,m)≻ ‖
         b′,i′,r′,m′ := b,i,r,m
       END


    REPLICATE b | b ∈ NODE •
     b.relay(a,i,r,m) ≙
       SELECT
         a ∈ NODE ∧ i ∈ TAG ∧ (r,m) ∈ ENV
       THEN
         b.lstore := b.lstore + ≺(i,r,m)≻
       END

END
```

Figure 17: Refined network agents.

MACHINE

$\quad LocalAgent1(a)$

CONSTRAINTS

$\quad a \in NODE$

VARIABLES

$\quad a.lstore$

INVARIANT

$\quad a.lstore \in bag\,(TAG \times ENV)$

INITIALISATION

$\quad a.lstore := \prec\succ$

OPERATIONS

$a.send(r,m) \;\hat{=}$
ANY i WHERE
$(r,m) \in ENV \wedge i \in TAG \wedge (a,r?) \in route(i)^*$
THEN
$a.lstore := a.lstore + \prec(i,r,m)\succ$
END

$m \longleftarrow a.receive \;\hat{=}$
ANY i,n WHERE
$(i,r,n) \in a.lstore$
THEN
$a.lstore := a.lstore - \prec(i,r,n)\succ \parallel m := n$
END

$(b',i',r',m') \longleftarrow a.forward \;\hat{=}$
ANY b,i,r,m WHERE
$(i,r,m) \in a.lstore \;\wedge\; r \neq a \;\wedge$
$(a,b) \in route(i) \;\wedge\; (b,r) \in route(i)^*$
THEN
$a.lstore := a.lstore - \prec(i,r,m)\succ \;\parallel$
$b',i',r',m' := b,i,r,m$
END

$a.relay(a',i,r,m) \;\hat{=}$
SELECT
$a' \in NODE \;\wedge\; i \in TAG \;\wedge\; (r,m) \in ENV$
THEN
$a.lstore := a.lstore + \prec(i,r,m)\succ$
END

END

Figure 18: Individual agent.

pair (t, X), where t is a trace of events and X is a set of events; $(t, X) \in F$ means that P may engage in the trace of events t and then refuse all the events in X. A divergence is a trace of events d, and $d \in D$ means that, after engaging the trace d, P may diverge (behave chaotically). Process (A, F, D) is refined by process (A, F', D'), written $(A, F, D) \sqsubseteq (A, F', D')$, if

$$F \supseteq F' \text{ and } D \supseteq D'.$$

In [15], a correspondence between CSP and an event-based view of action systems is described. This involves giving a failures-divergence semantics to action systems, with the execution of actions corresponding to the occurrence of CSP-like communication events. Let $\{\!\![M]\!\!\}$ represent the failures-divergence semantics of action system M. The definition of $\{\!\![M]\!\!\}$ may be found in [8, 15]. Previously we claimed that if M is simulated by N (Definitions 3 and 4), then any observable behaviour of N is an observable behavior of M. The observable behaviour of an action system is represented by its failures-divergence semantics and it can be shown [8, 20] that if M is simulated by N, then

$$\{\!\![M]\!\!\} \sqsubseteq \{\!\![N]\!\!\}.$$

CSP has both a hiding operator $(P \setminus C)$ for internalising events and a parallel composition operator $(P \parallel Q)$ for composing processes based on shared events. Both operators are defined in terms of failures-divergence semantics: Let $[\![P]\!]$ be the failures-divergence semantics of a process P. Then $[\![P \setminus C]\!]$ is defined by $HIDE([\![P]\!], C)$ and $[\![P \parallel Q]\!]$ is defined by $PAR([\![P]\!], [\![Q]\!])$, where $HIDE$ and PAR are described in [12]. It can be shown [8] that the hiding and parallel operators for action systems correspond to the CSP operators; that is, for action systems M and N:

$$\begin{aligned} \{\!\![M \setminus C]\!\!\} &= HIDE(\{\!\![M]\!\!\}, C) \\ \{\!\![M \parallel N]\!\!\} &= PAR(\{\!\![M]\!\!\}, \{\!\![N]\!\!\}). \end{aligned}$$

Since $HIDE$ and PAR are monotonic w.r.t. (failures-divergence) refinement [12], our earlier claim that the hiding and parallel operators for action systems are monotonic is justified.

8 Concluding

Although operations in B AMN can have both input and output parameters, it was stated earlier that actions can either be input actions or output actions, but not both. Consider an AMN action of the form

$$y \longleftarrow name(x) \;\hat{=}\; S.$$

In the implementation of this operation, we would expect a delay between receipt of x and the delivery of y. In particular, we may want to push the computation of y into some internal actions. In order to do this using simulation (Definition 4), the operation should be broken into an input action, representing receipt of x, and an output action, representing delivery of y. In this way, we can introduce internal actions that are executed in between receipt of x and delivery of y, contributing towards the computation of y. It also allows us to interleave other visible actions between receipt of x and delivery of y.

Abrial has proposed an approach to the design of protocols using the B method [2]. With this approach, a protocol is specified as a single operation which is subsequently decomposed into a sequence of steps through a series of refinements. The introduction of each new step in the protocol is justified by showing that it is a data-refinement of the *skip* action. This is the same as our data-refinement condition on internal actions being introduced by a simulation step (Condition 3, Definition 4). Of course, we also require that the nondivergence and progress conditions be satisfied (Conditions 4, 5, 6, Definition 4).

In this paper, we have taken an event-based view of action systems. In [10, 19], B is combined with a state-based view of action systems. With the state-based view, refinement is similar to the definitions presented here, but the parallel composition of machines is somewhat different; actions are not fused, but machines may share variables through which they interact. The choice between an event-based and a state-based view will depend on the nature of the application being developed. The event-based view is more suited to the design of message-passing distributed systems, while the state-based view is more suited to the design of parallel algorithms. Lano and Dick have also developed an approach to dealing with concurrency in B by combining it with a form of temporal logic [13].

We have seen the close correspondence between action systems and the abstract machines of B and seen the similarity between their notions of refinement. Because of this close correspondence, we are able to apply action system techniques such as internalisation of actions and parallel composition to abstract machines. These techniques provide a powerful abstraction mechanism since they allow us to abstract away from the distributed architecture of a system and the complex interactions between its subsystems; a system, such as the email service, can be specified as a single abstract machine and only in later refinement steps do we need to introduce explicit subsystems and interactions between them. The reasoning required to use these techniques involves refinement arguments and variant arguments, which is the sort of reasoning already used in B. The techniques are also very modular since the parallel components of a distributed system can be refined and decomposed separately without making any assumptions about the rest of the system.

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *First B Conference*, November 1996.

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

[4] R.J.R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume LNCS 947, pages 128–158. Springer–Verlag, 1995.

[5] 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.

[6] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comp. Prog.*, 13:133–180, 1989.

[7] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoret. Comp. Sci.*, 37:77–121, 1985.

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

[9] M.J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, September 1996.

[10] M.J. Butler and M. Waldén. Distributed system development in B. In H. Habrias, editor, *First B Conference*, November 1996.

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

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

[13] K. Lano and J.Dick. Development of concurrent systems in B AMN. In He Jifeng, editor, *7th BCS-FACS Refinement Workshop*, Workshops in Computing. Springer–Verlag, 1996.

[14] R. Milner. *Communication and Concurrency*. Prentice–Hall, 1989.

[15] 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.

[16] C.C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Formal Approaches to Computing and Information Technology. Springer, 1994.

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

[18] D.S. Nielson and I.H. Sorensen. The B-technologies: a system for computer aided programming. In U.H. Engberg, K.G. Larsen, and P.D. Mosses, editors, *6th Nordic Workshop on Programming Theory*. BRICS, October 1994.

[19] M. Waldén and K. Sere. Refining action systems within B-Tool. In *Formal Methods Europe (FME'96)*, volume LNCS 1051, pages 85 – 104. Springer–Verlag, March 1996.

[20] 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, pages 340–351. Springer–Verlag, 1990.