Event Ordering in Action Systems

Michael Butler
`mjb@ecs.soton.ac.uk`

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-98-3
May 1998

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Event Ordering in Action Systems*

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

May 1998

Abstract

Action systems are state machines that describe the behaviour of a distributed system in terms of the atomic actions that can take place during its operation. In this paper, techniques for constraining the order in which actions occur are introduced. It is shown how an event-ordering term may be added to an action system to constrain the ordering of events and it is shown how such a term may be translated into a standard action system to aid refinement.

1 Introduction

The action system formalism, introduced by Back and Kurki-Suonio [5], is a state-based approach to distributed computing. An action system describes the behaviour of a distributed system in terms of the atomic actions that can take place during its operation, specifying under what conditions an action can be executed and what the effect of execution is.

Action systems may be refined using standard techniques for program refinement and data refinement. This includes the possibility of introducing auxiliary, or internal, actions in a refinement step. This feature is useful in the design of distributed systems as it allows one to start with a specification that is independent of the distributed architecture, in which all information is globally available, and then use data refinement to partition the information, introducing internal actions to transfer information between partitions. An example of this may be found in [8], where a mail service is specified using a single mail bag to which users add addressed messages when sending mail and from which they may read mail; this simple 'global view' is then refined into a system with a more complicated data structure representing a network of nodes while internal actions are introduced to model the routing of messages through the network.

The ability to decompose action systems into parallel subsystems is also important for the design of distributed systems. This can be done in two ways: partition the actions amongst the subsystems and allow the subsystems to interact using shared state [6], or partition the state and allow interaction through shared actions [8]. In this paper, we follow the latter approach of allowing for interaction based on

*To appear in proceedings of *IRW/FMP'98 International Refinement Workshop and Formal Methods Pacific 1998.*

shared actions. We refer to it as the 'CSP approach' as its notion of action system behaviour is based on the semantic model used for Hoare's CSP [12].

Action system refinement and decomposition provide a powerful and practical approach to the correct design of distributed systems and they may be used to enhance industrial-strength state-based methods such as Z and B (see, for example, [7, 9]). In particular, parallel composition/decomposition is essentially a syntactic procedure.

However, while action systems are good at modelling parallel activity, they are weaker at modelling sequential activity. Typically one has to introduce an abstract 'program counter' to order the execution of actions. This can be a lot less transparent than the way in which one orders action execution in process algebras such as CSP and CCS [14]. The designers of the languages SL_0 [19] and CSP-OZ [11] have gone some way towards addressing this by combining CSP-like process specifications with action system-like specifications. However, while they allow for the separate refinement of the process parts and the action system parts, they do not support refinement steps involving the introduction or elimination of process parts nor allow one to make use of the process part when refining the action system part.

This ability to mix the process and the action system parts in refinements, as well as specification, is the contribution of this paper. We present a way of adding a process term, a so-called *event-ordering term*, to an action system to constrain the occurrence of actions. We use a subset of CCS to describe event-ordering terms. We show how an event-ordering term may be eliminated by calculating an equivalent action system without that term. In Section 4, we show how this technique may be used to show refinements between systems with event-ordering terms.

The extensions to the action system approach are modest and pragmatic. The theory is not extended in any way. The novelty is the provision of systematic ways to use these ideas effectively in conjunction with action systems. Before describing the extensions, we proceed with some background details on action systems.

2 Action Systems Overview

An action system M is essentially a state machine consisting of some state variables, an alphabet of action names αM, an action $M.a$ for each $a \in \alpha M$ and an initialisation action $M.init$. Each action $M.a$ has a guard, or enabling condition, written $gd(M.a)$. 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.

Actions are written in an extended version of Dijkstra's guarded command language [10]. Along with the *skip* statement, sequential composition $(A; B)$ and assignment $(x, y := E, F)$, we make use of the following language constructs:

- *Guarding*, $G \longrightarrow A$: execute A only if condition G holds.
- *Guarding with arbitrary value*, $@x \cdot G \longrightarrow A$: guarded action with body A making use of an arbitrary value x whose choice may be constrained by G.
- *Generalised assignment*, $x :\in S$: assign an arbitrary value from the set S to variable x. This is the same as $@x' \cdot x' \in S \longrightarrow x := x'$. Also, $x :=?$ is short for $x :\in T$, where T is the type of x.
- *Choice*, $A \mathbin{[\,]} B$: execute whichever is enabled of A and B. If both are enabled, choose from them nondeterministically.
- *Parallel execution*, $A \parallel B$: execute A and B in parallel. A and B must write to separate variables.

$$
\begin{array}{l}
\text{SYSTEM} \quad \mathit{MailService} \\
\text{VAR} \quad \mathit{mail} : \mathit{User} \rightarrow \mathit{bag}\ (\mathit{User} \times \mathit{Msg}) \\
\text{INIT} \quad \mathit{mail} := (\lambda b : \mathit{User} \cdot \{\}) \\
\text{ACTIONS} \\
\qquad \mathit{Send}(a, b, m) \ \hat{=} \ \ \mathit{mail}(b) := \mathit{mail}(b) + \{(a, m)\} \\
\qquad (a, m) \leftarrow \mathit{Recv}(b) \ \hat{=} \ \ (a, m) :\in \mathit{mail}(b) \\
\text{END}
\end{array}
$$

Figure 1: Mail service action system.

Statements in Dijkstra's guarded command language are defined using *weakest precondition* formulae: for action A and postcondition R, the formula $wp(A, R)$ (weakest precondition of A w.r.t. R) characterises those initial states from which action A is guaranteed to terminate in a state satisfying R. A weakest-precondition definition of parallel execution can be found in [2, 4]. Weakest-precondition definitions of all the other extensions can be found in [2, 3, 16, 17].

The guard of an action is defined as follows[1] [15]:

$$gd(A) \quad \hat{=} \quad \neg wp(A, \mathit{false}).$$

From this, the following properties may be derived:

$$
\begin{array}{rcl}
gd(\mathit{skip}) & = & \mathit{true} \\
gd(x, y := E, F) & = & \mathit{true} \\
gd(G \longrightarrow A) & = & G \wedge gd(A) \\
gd(@x \cdot G \ \longrightarrow \ A) & = & (\exists x \cdot G \wedge gd(A)) \\
gd(x :\in S) & = & S \neq \{\} \\
gd(A \mathbin{[\!]} B) & = & gd(A) \vee gd(B) \\
gd(A \parallel B) & = & gd(A) \wedge gd(B).
\end{array}
$$

As well as updating the state variables, an action may have some input parameters and output parameters. The specification of an action named a has the form

$$y \leftarrow a(x) \ \hat{=} \ A,$$

where x represents input parameter(s), y represents output parameter(s) and A is a statement in the action notation. Action A may write to the state variables and to variable y and it may depend on the value of x though it does not write to x.

Fig. 1 gives an example of an action system. This specifies a mail service allowing users to exchange messages. The state variable contains a bag (or multiset) of user-message pairs for each user b. Action $\mathit{Send}(a, b, m)$ is invoked when user a wishes to send message m to user b resulting in (a, m) being added to the bag $\mathit{mail}(b)$. User b may receive a message m from user a provided $(a, m) \in \mathit{mail}(b)$; b reads such messages by invoking $\mathit{Recv}(b)$ which generates some pair (a, m) as output provided $\mathit{mail}(b) \neq \{\}$.

Following the approach of [8], the environment interacts with an action system through the actions and the state is not directly visible to the environment. An action system may also contain *internal* actions. These are actions that can change the state of the system but whose occurrence is not directly observable by the environment of the system. Examples of such actions will be used in Section 4.

[1] $wp(A, \mathit{false})$ represents those initial states in which A could establish any postcondition, i.e., behave miraculously. An action is said to be enabled when it cannot behave miraculously, i.e., when $\neg wp(A, \mathit{false})$ holds.

$$P + P = P, \quad P + Q = Q + P, \quad (P+Q)+R = P+(Q+R),$$
$$P(x) + P(y) = \Sigma i \in \{x, y\} \cdot P(i), \quad P + \emptyset = P,$$
$$(\Sigma i \in S \cdot P(i)) \; + \; (\Sigma i \in T \cdot P(i)) \;=\; (\Sigma i \in S \cup T \cdot P(i)),$$
$$P = Q \;\Leftrightarrow\; a.P = a.Q$$

Figure 2: Algebraic laws for event-ordering terms.

3 Event-ordering Terms

We use a simple subset of CCS with event prefixing ($a.P$: engage in event a then behave as P) and choice ($P + Q$: offer the choice of P and Q) to describe event-ordering. Our simple term language has the following syntax:

$$P \;::=\; a.P \;\mid\; P + Q \;\mid\; \Sigma i \in S \cdot P(i).$$

The term $\Sigma i \in S \cdot P(i)$ represents a generalised choice of possible behaviours. We write $\emptyset$ for $\Sigma i \in \{\} \cdot P(i)$, the empty choice of possible behaviours.

The laws of Fig. 2 define equality between terms. Note that prefixing does not distribute through choice. The transition relation $P \xrightarrow{a} P'$ says that it is possible for P to evolve to P' by engaging in event a. This is defined by the following three rules:

$$a.P \xrightarrow{a} P \qquad \frac{P \xrightarrow{a} P'}{P + Q \;\xrightarrow{a}\; P'} \qquad \frac{P(i) \xrightarrow{a} P', \quad i \in S}{\Sigma i \in S \cdot P(i) \;\xrightarrow{a}\; P'}$$

Note that $\emptyset$ cannot evolve at all. It is easy to show that equality implies behavioural equivalence, i.e., if $P = Q$ and $P' = Q'$, then

$$P \xrightarrow{a} P' \;\Leftrightarrow\; Q \xrightarrow{a} Q'.$$

Fig. 3 gives an example of an action system with an event-ordering term. (We assume that event prefixing binds more tightly than choice.) The system provides actions to increment and decrement a counter. The counter is regarded as a shared resource so, before it can be changed, it must first be claimed for exclusive use by executing a *Lock* action. After operating on the counter, it is freed using the *Unlock* action. These ordering constraints are imposed by the *initial* event-ordering term $P(0)$. The *Dec* action is further constrained with the explicit guard $c > 0$ so, even though the event-ordering term says *Dec* may happen after the first *Lock* (when $c = 0$), the guard prevents it from occurring.

3.1 Action System Normalisation

An event-ordering term may be described by a mutually recursive set of indexed terms $P(i) \mathrel{\hat{=}} \ldots, P(j) \mathrel{\hat{=}} \ldots$, etc, with an initial term given by $P(0_{Index})$. Such a set of terms will define a function $next : Index \times Event \to \mathbb{P}\ Index$:

$$next(i, a) \;=\; \{\; i' \mid P(i) \xrightarrow{a} P(i') \;\}.$$

This function may be used to give meaning to an event-ordering term in terms of a standard action system. An action system M with variables v, actions $M.a \mathrel{\hat{=}} A$, $M.b \mathrel{\hat{=}} B$, etc and event-ordering term P (which defines a function *next* as above) is really a short for the following 'normalised' machine:

SYSTEM $\quad Counter1$
VAR $\quad c : \mathbb{N}$
INIT $\quad c := 0$
ACTIONS

$$Lock \;\hat{=}\; skip$$

$$Inc \;\hat{=}\; c := c + 1$$

$$Dec \;\hat{=}\; c > 0 \longrightarrow c := c - 1$$

$$Unlock \;\hat{=}\; skip$$

EVENT_ORDERING $\quad P(0)\quad$ WHERE

$$\begin{array}{lcl} P(0) & \hat{=} & Lock\,.\,P(1) \\ P(1) & \hat{=} & Inc\,.\,P(1) \;+\; Dec\,.\,P(1) \;+\; Unlock\,.\,P(0) \end{array}$$

END

Figure 3: Action system with event-ordering term.

SYSTEM $\quad M$
VAR $\quad v : V,\; p : Index$
INIT $\quad M.init \;\|\; p := 0_{Index}$
ACTIONS

$$a \;\hat{=}\; A \;\|\; p :\in next(p, a)$$

$$b \;\hat{=}\; B \;\|\; p :\in next(p, b)$$

...

END

An indexing variable p is added to the state, initialised to the initial event-ordering index 0_{Index}, and each action is augmented with a generalised assignment that selects a new value for p.

We have the following rule for normalisation of actions:

Rule 1 (Normalisation I) *Given a set of event-ordering terms $P(i)$, an action of the form $a \;\hat{=}\; A$ is normalised to*

$$a \;\hat{=}\; A \;\|\; p :\in next(p, a),$$

where $next(p, a)$ is as given above.

The set $next(p, a)$ must be non-empty for the normalised action to be enabled:

$$\begin{array}{rl} & gd(\; A \;\;\|\;\; p :\in next(p, a)\;) \\ = & gd(A) \;\wedge\; gd(p :\in next(p, a)) \\ = & gd(A) \;\wedge\; next(p, a) \neq \{\}. \end{array}$$

Note that Rule 1 only deals with unparameterised actions. Parameterised actions will be dealt with in Section 3.4

3.2 Simplifying Normalised Actions

A *prefixed choice*[2] is a term of the form $(\Sigma i \in S \cdot a_i\,.\,P(i))$ where each choice is prefixed by an event. This definition applies even when S is empty. If Q is a

[2] In [12, 14] this is called *guarded* choice. We use the term *prefixed* to avoid confusion with guarded actions.

prefixed choice term, then an equation of the form

$$P(i) \quad \hat{=} \quad a.Q + R$$

may be replaced by the pair of equations

$$P(i) \;\; \hat{=} \;\; a.P(j) + R, \qquad P(j) \;\; \hat{=} \;\; Q,$$

where j is some fresh index. For example,

$$P(0) \;\; \hat{=} \;\; a.b.P(0) \; + \; c.P(0)$$

may be rewritten to

$$P(0) \;\; \hat{=} \;\; a.P(1) \; + \; c.P(0), \qquad P(1) \;\; \hat{=} \;\; b.P(0)$$

This allows us to rewrite any event-ordering term, in which all choices are prefixed and finite, as a set of equations of the form

$$\begin{array}{lcl} P(i) & \hat{=} & a_1.P(i_1) \; + \; \cdots \; + \; a_n.P(i_n), \\ \text{or} \;\; P(i) & \hat{=} & \emptyset. \end{array}$$

Here, a_m and a_n may denote different occurrences of the same event.

This form provides a systematic way of simplifying $p :\in next(p, a)$:

Rule 2 (Simplifying $p :\in next(p, a)$ **)** *For an event a, assume each $P(i)$ is described by a set of equations of the form:*

$$\begin{array}{lcl} P(i_1) & \hat{=} & \cdots + \; a.P(i_1') \; + \cdots + \; a.P(i_1'') \; + \cdots \\ P(i_2) & \hat{=} & \cdots + \; a.P(i_2') \; + \cdots \\ P(i_3) & \hat{=} & \cdots \end{array}$$

Then $p :\in next(p, a)$ is equivalent to the following action:

$$\begin{array}{ll} & p = i_1 \longrightarrow p :\in \{i_1', i_1'', \ldots\} \\ [] & p = i_2 \longrightarrow p :\in \{i_2', \ldots\} \\ [] & \ldots \end{array}$$

This action contains a choice branch for each i_n for which some occurrence of $a.P(i_n')$ appears in the right-hand-side. For each such occurrence in an equation $P(i_n) \;\hat{=}\; \cdots$, the choice set $\{i_n', i_n'', \ldots\}$ for the next value of p contains the index of the next state i_n'.

For example, *Lock* appears in one place in the equations for the event-ordering term of Fig. 3, i.e., on the right-hand-side of $P(0)$, so we get:

$$p :\in next(p, Lock) \quad = \quad p = 0 \longrightarrow p :\in \{1\}.$$

This may be simplified further:

$$= \quad p = 0 \longrightarrow p := 1.$$

The justification for this and other simplifications of normalised actions is provided by various algebraic laws for actions such as those of Fig. 4.

The normalised *Lock* action of *Counter*1 is now further simplified as follows:

$$\begin{array}{lll} & Lock \;\; \| \;\; p :\in next(p, Lock) & \\ = & skip \;\; \| \;\; p = 0 \longrightarrow p := 1 & \\ = & p = 0 \longrightarrow p := 1 & \textit{(skip is identity).} \end{array}$$

$$
\begin{array}{rcl}
x :\in \{E\} & = & x := E \\
x = E \longrightarrow x := E & = & x = E \longrightarrow \mathit{skip} \\
x := E \;\; \| \;\; y := F & = & x, y := E, F \\
G_1 \longrightarrow A_1 \;\; \| \;\; G_2 \longrightarrow A_2 & = & G_1 \wedge G_2 \;\; \longrightarrow \;\; A_1 \| A_2 \\
\mathit{skip} \| A & = & A.
\end{array}
$$

Figure 4: Some algebraic laws for actions.

Similarly, Dec appears once in the term equations for $Counter1$, and we get:

$$p :\in \mathit{next}(p, \mathit{Dec}) \quad = \quad p = 1 \longrightarrow p :\in \{1\}.$$

This may be further simplified:

$$
\begin{array}{cll}
= & p = 1 \longrightarrow p := 1 & \\
= & p = 1 \longrightarrow \mathit{skip} & \quad \textit{(redundant assignment).}
\end{array}
$$

The normalised Dec action is then further simplified:

$$
\begin{array}{cll}
 & \mathit{Dec} \;\; \| \;\; p :\in \mathit{next}(p, \mathit{Dec}) & \\
= & c > 0 \longrightarrow c := c - 1 \;\; \| \;\; p = 1 \longrightarrow \mathit{skip} & \\
= & c > 0 \wedge p = 1 \;\; \longrightarrow \;\; c := c - 1 \| \mathit{skip} & \quad \textit{(merge guards)} \\
= & c > 0 \wedge p = 1 \;\; \longrightarrow \;\; c := c - 1. &
\end{array}
$$

3.3 More Examples

We look at three variations of a simple vending machine which accepts coins and delivers coffee. In each case, the system has two actions *Coin* and *Coffee*, both defined as *skip*:

$$\mathit{Coin} \;\hat{=}\; \mathit{skip} \qquad \mathit{Coffee} \;\hat{=}\; \mathit{skip}.$$

The event-ordering term in the first example involves the following set of event-ordering equations in which the *Coin* event appears in more than one right-hand-side:

$$
\begin{array}{rcl}
P(0) & \hat{=} & \mathit{Coin}.P(1) \\
P(1) & \hat{=} & \mathit{Coffee}.P(0) \; + \; \mathit{Coin}.P(1).
\end{array}
$$

These give rise to a *Coin* action with a choice of enabling conditions:

$$p :\in \mathit{next}(p, \mathit{Coin}) \quad = \quad p = 0 \longrightarrow p := 1 \;\; [] \;\; p = 1 \longrightarrow p := 1.$$

In the second example, the *Coin* event appears twice in one right-hand-side of an event-ordering equation:

$$
\begin{array}{rcl}
P(0) & \hat{=} & \mathit{Coin}.P(0) \; + \; \mathit{Coin}.P(1) \\
P(1) & \hat{=} & \mathit{Coffee}.P(0).
\end{array}
$$

These give rise to a *Coin* action with nondeterministic choice of next index:

$$p :\in \mathit{next}(p, \mathit{Coin}) \quad = \quad p = 0 \longrightarrow p :\in \{0, 1\}.$$

SYSTEM $\quad$ *Counter*2
VAR $\quad c : \mathbb{N}$
INIT $\quad c := 0$
ACTIONS

$$Lock(u : User) \;\hat{=}\; skip$$

$$Inc(u : User,\ m : \mathbb{N}) \;\hat{=}\; x := c + m$$

$$Dec(u : User,\ m : \mathbb{N}) \;\hat{=}\; c \geq m \longrightarrow c := c - m$$

$$Unlock(u : User) \;\hat{=}\; skip$$

EVENT_ORDERING $\;P(0, \_)\;$ WHERE

$$P(0, v) \;\hat{=}\; \Sigma u \cdot Lock(u)\mathbf{.}P(1, u)$$
$$P(1, u) \;\hat{=}\; Inc(u, \_)\mathbf{.}P(1, u) \;+\; Dec(u, \_)\mathbf{.}P(1, u) \;+\; Unlock(u)\mathbf{.}P(0, u)$$

END

Figure 5: Counter with parameterised actions.

The third example involves a vending machine which only delivers one coffee and then stops:

$$P(0) \;\hat{=}\; Coin\mathbf{.}Coffee\mathbf{.}\emptyset\ .$$

This event-ordering term is rewritten as:

$$P(0) \,\hat{=}\, Coin\mathbf{.}P(1), \qquad P(1) \,\hat{=}\, Coffee\mathbf{.}P(2), \qquad P(2) \,\hat{=}\, \emptyset.$$

This gives rise to a system that reaches a deadlocked state:

SYSTEM $\quad VM2$
VAR $\quad p : \{0, 1, 2\}$
INIT $\quad p := 0$
ACTIONS

$$Coin \;\hat{=}\; p = 0 \longrightarrow p := 1$$

$$Coffee \;\hat{=}\; p = 1 \longrightarrow p := 2$$

END

3.4 Parameterised Events

Sometimes it is convenient to use the parameters of an event in an event-ordering term. For example, consider the counter system of Fig. 5. Here each action is parameterised with the identity of the invoking user u. (The type of a parameter is given explicitly where it cannot be inferred from the context.) The *Inc* and *Dec* actions also have a parameter m for the amount to increment/decrement the counter by.

Because of the event-ordering term, once the counter has been locked by user u, only that user can change it or unlock it. Generalised choice is used in $P(0, v)$ to represent a choice of input parameter u. The underscore (_) signifies an arbitrary value; thus the event-ordering term places no constraint on the parameter m of *Inc* and *Dec*.

In the above example, each event-ordering term is indexed by a pair of values. To construct a normalised system, we introduce two extra variables pi and pj respectively for each of the index values. Note that pj may, in general, be a tuple, but typically pi will represent a flow-of-control value while pj will represent some data value. We introduce the following rule for action normalisation:

Rule 3 (Normalisation II) *Given a set of event-ordering terms $P(i,j)$, an action of the form $a(x) \;\hat{=}\; A$ is normalised to*
$a(x) \;\hat{=}\; A \;\|\; pi, pj :\in next(pi, pj, a(x))$,
where $next(pi, pj, a(x))$ is given by:

$$next(i, j, a(x)) \;=\; \{\; i', j' \mid P(i,j) \stackrel{a(x)}{\rightarrow} P(i', j') \;\}.$$

We consider how to simplify $pi, pj :\in next(pi, pj, a(x))$ using the event-ordering term. We shall assume that terms are written using equations of the following form[3]:

$$\begin{array}{rcl} P(i,j) & \hat{=} & \cdots + (\Sigma x \mid C_m \cdot a_m(x).P(i_m, j_m)) + \cdots, \\ \text{or } P(i,j) & \hat{=} & \emptyset. \end{array}$$

Here, a_m and a_n may denote different occurrences of the same event. Each C_m is a predicate that may constrain the choice of x and may depend on i and j. Also, for choice term $(\Sigma x \mid C_n \cdot a_n(x).P(i_n, j_n))$, we assume that the next indices i_n and j_n will be some functions of i, j and x, i.e., such a term will be of the form $(\Sigma x \mid C_n \cdot a_n(x).P(f(i,j,x), g(i,j,x)))$

For example, the event-ordering term of $Counter2$ may be written as:

$$\begin{array}{rcll} P(0,j) & \hat{=} & \Sigma u \mid true \cdot Lock(u).P(1,u) & (1) \\ P(1,j) & \hat{=} & \Sigma u, m \mid u = j \cdot Inc(u,m).P(1,j) \;+ & (2) \\ & & \Sigma u, m \mid u = j \cdot Dec(u,m).P(1,j) \;+ & (3) \\ & & \Sigma u \mid u = j \cdot Unlock(u).P(0,j). & (4) \end{array}$$

In general, a term definition of the form $P(i,j) \;\hat{=}\; Q$, where each choice in Q is prefixed may be normalised to the above form. Let Q_j be a prefix choice term, then

$$P(i,j) \;\hat{=}\; (\; \Sigma x \mid C(i,j,x) \cdot a(x).Q_{f(i,j,x)} \;) \;+\; R$$

may be replaced by

$$\begin{array}{rcl} P(i,j) & \hat{=} & (\; \Sigma x \mid C(i,j,x) \cdot a(x).P(i', f(i,j,x)) \;) \;+\; R \\ P(i',j) & \hat{=} & Q_j, \end{array}$$

where i' is some fresh index.

The above normal form for term equations involving parameterised events and pairs of indices provides a systematic way of simplifying the action $pi, pj :\in next(pi, pj, a(x))$:

Rule 4 (Simplifying $pi, pj :\in next(pi, pj, a(x))$) *For an event $a(x)$, assume each $P(i,j)$ is described by a set of equations of the form:*

$$P(i_n, j) \;\hat{=}\; \cdots + (\; \Sigma x \mid C(i_n, j, x) \cdot a(x).P(f(i_n, j, x), g(i_n, j, x)) \;) \;+\; \cdots.$$

Then

$$\begin{array}{ll} pi, pj :\in next(pi, pj, a(x)) \;= & \\ & \cdots \\ [\!] & pi = i_n \;\wedge\; C(pi, pj, x) \longrightarrow pi, pj := f(pi, pj, x), g(pi, pj, x) \\ [\!] & \cdots \end{array}$$

[3] $\Sigma x \mid C \cdot P(x)$ is short for $\Sigma x \in \{\; x \mid C \;\} \cdot P(x)$.

SYSTEM *FileTransferService*
ACTIONS

$$Send(m : File) \;\hat{=}\; skip$$

$$(m : File) \leftarrow Recv \;\hat{=}\; m :=?$$

EVENT_ORDERING P WHERE

$$P \;=\; \Sigma m \cdot Send(m)\,.\,Recv(m)\,.\,P$$

END

Figure 6: File transfer service.

Here, for each occurrence of $\Sigma x \mid C(i_n, j, x) \cdot a(x)\,.\,P(f(i_n, j, x), g(i_n, j, x))$ *on the right-hand-side of an equation whose left-hand-side is* $P(i_n, j)$*, the action has an appropriate choice branch that is enabled when* $pi = i_n$ *and* $C(pi, pj, x)$ *holds and whose effect is* $pi, pj := f(pi, pj, x), g(pi, pj, x)$*.*

With Rule 2, a nondeterministic choice of next index was modelled by a generalised assignment. Here it is modelled by having several branches with overlapping guards.

From (1), (3) and (4) respectively, we get

$$\begin{array}{rcl} pi, pj :\in next(pi, pj, Lock(u)) & = & pi = 0 \longrightarrow pi, pj := 1, u \\ pi, pj :\in next(pi, pj, Dec(u, m)) & = & pi = 1 \wedge u = pj \longrightarrow pi, pj := 1, pj \\ pi, pj :\in next(pi, pj, Unlock(u)) & = & pi = 1 \wedge u = pj \longrightarrow pi, pj := 0, pj\ . \end{array}$$

The normalised $Dec(u, m)$ action, for example, is simplified as follows:

$$\begin{array}{rl} & Dec(u, m) \;\|\; pi, pj :\in next(pi, pj, Dec(u, m)) \\ = & c \geq m \longrightarrow c := c - m \;\;\|\;\; pi = 1 \wedge u = pj \longrightarrow pi, pj := 1, pj \\ = & c \geq m \longrightarrow c := c - m \;\;\|\;\; pi = 1 \wedge u = pj \longrightarrow skip \\ = & c \geq m \wedge pi = 1 \wedge u = pj \longrightarrow c := c - m. \end{array}$$

3.5 Events with Output

An event-ordering term may be used to constrain the output value generated by an action. This can be seen in the file transfer service of Fig. 6 in which a file m sent with a $Send(m)$ event is received with a following $Recv(m)$ event. The m in the $Recv(m)$ event is an output value, as can be seen from the specification of the $Recv$ action. While the $Recv$ action allows an arbitrary file to be output, the event-ordering term constrains the output file to be the file that was input with the preceding $Send$.

As both the action and the event-ordering term may constrain the choice of possible value for an output parameter, the parallel operator for actions is insufficient for action normalisation. Instead we use the so-called *fusion* operator introduced in [4]. The fusion of A and B is written $A \odot B$ and allows both A and B to write to the same variables such that the value written satisfies both A and B. The operator satisfies the following laws (P and Q are predicates rather than event-ordering terms):

$$
\begin{aligned}
(@x', y' \cdot P \longrightarrow x, y := x', y') \;\odot\; (@y', z' \cdot Q \longrightarrow y, z := y', z') \\
= \;\; @x', y', z' \cdot P \wedge Q \longrightarrow x, y, z := x', y', z'
\end{aligned}
$$

$$
\begin{array}{rcl}
x :\in S \;\odot\; x :\in T & = & x :\in S \cap T \\
x :=? \;\odot\; (@x', y' \cdot P \longrightarrow x, y := x', y') & = & (@x', y' \cdot P \longrightarrow x, y := x', y').
\end{array}
$$

We then introduce the following normalisation rule that uses a function $next_{out}$ which, as well as giving possible next values for the indices i and j, also gives possible values for the output parameter y:

Rule 5 (Normalisation III) *Given a set of event-ordering terms $P(i,j)$, an action of the form $y \leftarrow a(x) \;\hat{=}\; A$ is normalised to*

$$y \leftarrow a(x) \quad \hat{=} \quad A \;\odot\; y, pi, pj :\in next_{out}(pi, pj, a(x))$$

where $next_{out}(i,j,a(x))$ is given by:

$$next_{out}(i,j,a(x)) \quad = \quad \{\; y, i', j' \mid P(i,j) \stackrel{a(x)(y)}{\rightarrow} P(i', j') \;\}.$$

The difference in the treatment of input parameters and output parameters is significant. Where there is a choice of possible input values, that choice is offered to the environment. On the other hand, where there is a choice of possible output values, that choice is not offered to the environment but is made internally by the system. This internal choice of output value is modelled above by a nondeterministic assignment.

The action $y, pi, pj :\in next_{out}(pi, pj, a(x))$ may be simplified in a similar way to $pi, pj :\in next(pi, pj, a(x))$:

Rule 6 (Simplifying $y, pi, pj :\in next_{out}(pi, pj, a(x))$) *For an event $a(x)(y)$, assume each $P(i,j)$ is described by a set of equations of the form:*

$$
\begin{aligned}
P(i_n, j) \quad \hat{=} \quad \cdots + \; (\; \Sigma x, y \mid C(i_n, j, x, y) \cdot \\
a(x)(y) \,\mathbf{.}\, P(f(i_n, j, x, y), g(i_n, j, x, y)) \;) \; + \; \cdots.
\end{aligned}
$$

Then

$$
\begin{aligned}
& y, pi, pj :\in next_{out}(pi, pj, a(x)) \quad = \\
& \quad \cdots \;\; [] \;\; @y' \cdot pi = i_n \;\wedge\; C(pi, pj, x, y') \\
& \qquad\qquad \longrightarrow \;\; y, pi, pj := y', f(pi, pj, x, y'), g(pi, pj, x, y') \;\; [] \;\; \cdots
\end{aligned}
$$

Here, the choice of output value is represented by y' and may be constrained by C. For example, the event-ordering term in Fig. 6 may be rewritten as:

$$
\begin{array}{rcl}
P(0,j) & \hat{=} & \Sigma m \cdot Send(m) \,\mathbf{.}\, P(1,m) \\
P(1,j) & \hat{=} & \Sigma m \mid m = j \cdot Recv(m) \,\mathbf{.}\, P(0,j).
\end{array}
$$

Thus, $m, pi, pj :\in next_{out}(pi, pj, Recv)$ may be simplified as follows:

$$
\begin{array}{rl}
& m, pi, pj :\in next_{out}(pi, pj, Recv) \\
= & @m' \cdot pi = 1 \wedge m' = pj \;\longrightarrow\; m, pi, pj := m', 0, pj \\
= & pi = 1 \;\longrightarrow\; m, pi, pj := pj, 0, pj \\
= & pi = 1 \;\longrightarrow\; m, pi := pj, 0.
\end{array}
$$

The *Recv* action is then normalised as follows:

$$\begin{array}{ll} & Recv \;\; \| \;\; m, pi, pj :\in next_{out}(pi, pj, Recv) \\ = & m :=? \;\; \| \;\; pi = 1 \longrightarrow m, pi := pj, 0 \\ = & pi = 1 \longrightarrow m, pi := pj, 0. \end{array}$$

In the above example, the output value m was not constrained by the *Recv* action, only by the event-ordering term. It is possible for both an action and an event-ordering to constrain an output value. For example, suppose we have an action as follows:

$$y \leftarrow Out \;\; \hat{=} \;\; y :\in S,$$

and that *Out* is further constrained by the following term:

$$P \;\; \hat{=} \;\; \Sigma y \mid y \in T \cdot Out(y).P.$$

From the definition of P we get:

$$pi, pj, y :\in next_{out}(pi, pj, Out) \;\; = \;\; y :\in T.$$

Thus the *Out* action is normalised to

$$y :\in S \;\odot\; y :\in T \quad = \quad y :\in S \cap T.$$

So the output value y will be chosen from $S \cap T$. If $S \cap T$ is empty, the action will be disabled.

4 Refinement

An action system may be refined by applying the standard technique of data refinement to its state: an abstraction invariant is used to relate the state variables of the abstract system to those of the refined, or concrete, system and data refinement should hold between correspondingly-named actions in the abstract and concrete systems.

If A is an action that acts on variables a, C is an action that acts on variables c, and I is an abstraction invariant then we write

$$A \sqsubseteq_I C$$

for 'A is data-refined by C under abstraction invariant I'.

The weakest-precondition definition of $A \sqsubseteq_I C$ can be found in [2, 3, 16, 18]. The following rule for data refinement of guarded actions (with output) is easily derived from the definition:

Rule 7

$$\frac{Q \wedge I \;\Rightarrow\; P \wedge I[a, c \backslash E, F]}{@y' \cdot P \longrightarrow a, y := E, y' \;\; \sqsubseteq_I \;\; @y' \cdot Q \longrightarrow c, y := F, y'}$$

An action system M is refined by action system N under abstraction invariant I when M and N have the same set of action names ($\alpha(M) = \alpha(N)$) and the following conditions hold:

Rule 8 (Action System Refinement)

1. $M.init \;\sqsubseteq_I\; N.init$
2. $M.a \;\sqsubseteq_I\; N.a, \qquad each\ a \in \alpha(M)$
3. $I \wedge gd(M.a) \;\Rightarrow\; gd(N.a), \qquad each\ a \in \alpha(M).$

$$
\begin{array}{ll}
\text{SYSTEM} \quad \mathit{Calculator}1 & \text{SYSTEM} \quad \mathit{Calculator}2 \\
\text{VAR} \quad s : \mathit{bag}\ \mathbb{N} & \text{VAR} \quad t : \mathit{bag}\ \mathbb{N} \\
\text{INIT} \quad s :\in \mathit{bag}\ \mathbb{N} & \text{INIT} \quad t := \{\} \\
\text{ACTIONS} & \text{ACTIONS} \\
\qquad \mathit{Clear} \ \hat{=} \ s := \{\} & \qquad \mathit{Clear} \ \hat{=} \ \mathit{skip} \\
\qquad \mathit{Add}(x) \ \hat{=} \ s := s + \{x\} & \qquad \mathit{Add}(x) \ \hat{=} \ t := t + \{x\} \\
\qquad y \leftarrow \mathit{GetMean} \ \hat{=} \ y := \mathit{mean}(s) & \qquad y \leftarrow \mathit{GetMean} \ \hat{=} \ y := \mathit{mean}(t) \\
\text{EVENT\_ORDERING} \ P(0) \ \text{WHERE} & \text{EVENT\_ORDERING} \ P(0) \ \text{WHERE} \\
\qquad P(0) \ \hat{=} \ \mathit{Clear}.P(1) & \qquad P(0) \ \hat{=} \ \mathit{Clear}.P(1) \\
\qquad P(1) \ \hat{=} \ \mathit{Add}(\_).P(1) \ + & \qquad P(1) \ \hat{=} \ \mathit{Add}(\_).P(1) \ + \\
\qquad \qquad \mathit{GetMean}(\_).P(1) & \qquad \qquad \mathit{GetMean}(\_).P(1) \\
\text{END} & \text{END}
\end{array}
$$

Figure 7: Equivalent calculator systems.

4.1 Refinement with Event-ordering Terms

For any action system with an event-ordering term, it is easy to show that if we apply refinement to the action system part and leave the event-ordering term unchanged, then we get refinement of the overall (normalised) system also. However, there are many cases where it is not possible to show refinement between the state parts of two systems, even though there is refinement between the overall systems. In that case, we need to normalise the systems in order to be able to verify the refinement. In this section, we illustrate this with two examples.

The first example concerns a simple system for calculating the mean of a series of numbers. This may be specified by the two equivalent action systems shown in Fig. 7. Numbers are entered using the *Add* action and the mean of all numbers added so far is accessed using the *GetMean* action. However, according to the event-ordering term, the *Clear* action must be invoked before either *Add* or *GetMean* may be used. The difference between the two systems is that, in *Calculator*1, the bag is emptied by the *Clear* operation whereas, in *Calculator*2, it is emptied by the initialisation and the *Clear* action behaves as *skip*.

Ignoring the event-ordering term, it is not the case that *Calculator*1 is refined by *Calculator*2. For example, without the event-ordering term, we could engage in the following trace of events with *Calculator*2:

$$\mathit{Add}(2).\mathit{Clear}.\mathit{Add}(4).\mathit{GetMean}(3).$$

But this is not possible in *Calculator*1 where, again without the event-ordering term, we would instead get

$$\mathit{Add}(2).\mathit{Clear}.\mathit{Add}(4).\mathit{GetMean}(4).$$

Similarly the reverse refinement does not hold because with *Calculator*1, if we invoke *GetMean* as the first event, some arbitrary number will be returned (since s is initialised to an arbitrary bag of numbers) whereas *Calculator*2 only allows 0 (i.e., $\mathit{mean}(\{\})$) to be returned in that situation.

However, by normalising both systems as described in the previous section, it is easy to show that the refinement holds in both directions. Both systems are augmented with a variable p ranging over $\{0, 1\}$ and each action is guarded

SYSTEM $\quad FileTransferProtocol$
VAR $\quad afile, bfile : seq\ Byte; \quad bsize : \mathbb{N}$
ACTIONS

$$Send(m) \ \hat{=} \ afile := m$$

$$m \leftarrow Recv \ \hat{=} \ m := bfile$$

INTERNAL ACTIONS

$$OpenReq \ \hat{=} \ skip$$

$$OpenResp \ \hat{=} \ bsize :\in 1..\#afile \ \ \| \ \ bfile := \langle\rangle$$

$$TransBlock \ \hat{=} \ afile \neq \langle\rangle \longrightarrow \begin{array}{l} bfile := bfile ^\frown afile[0..bsize-1] \ \ \| \\ afile := afile[bsize..] \end{array}$$

$$EndTrans \ \hat{=} \ afile = \langle\rangle \longrightarrow skip$$

$$Ack \ \hat{=} \ skip$$

EVENT_ORDERING $R(0)$ WHERE

$$\begin{array}{lcl} R(0) & \hat{=} & Send(\_) \,.\, OpenReq \,.\, OpenResp \,.\, R(3) \\ R(3) & \hat{=} & TransBlock \,.\, R(3) \ + \ EndTrans \,.\, Recv(\_) \,.\, Ack \,.\, R(0) \end{array}$$

END

Figure 8: Simple block-wise file transfer protocol.

appropriately. The obvious abstraction invariant to use for showing the refinements is then:

$$p = 0 \ \Rightarrow \ t = \{\} \quad \wedge \quad p = 1 \ \Rightarrow \ s = t.$$

In the second example in this section, we use internal actions. Recall that these are actions that can change the state of the system but whose occurrence is not directly observable to the environment of the system. The example is a file transfer system and is given in Fig. 8. The system has two observable actions, *Send* and *Recv*, and five internal actions. A file is sent using *Send* and received using *Recv*. The *Send* action stores the input file m in the variable *afile* while the *Recv* action outputs the value of variable *bfile*. The internal actions are responsible for transferring the contents of *afile* to *bfile* as a series of blocks. Note: $afile[0..bsize-1]$ denotes the initial segment of sequence *afile* up to $bsize - 1$, while $afile[bsize..]$ denotes the remaining segment. Each event of the protocol is represented by an internal action. The idea of the protocol is that the sending side first informs the receiving side that it wishes to transmit a file (*OpenReq*). The receiving side responds, deciding what the maximum size of each block should be (*OpenResp*). Then each block is transmitted in turn (*TransBlock*) and, when all blocks have been transmitted, an *EndTrans* is issued. Once the file has been output at the receiving end, an acknowledgment is issued. The ordering of the protocol events is modelled by the event-ordering term while the data transfer is modelled by the actions.

Intuitively, it would seem that the file transfer protocol of Fig. 8 should be a refinement of the file transfer service of Fig. 6, and indeed that is the case (with the type *File* being instantiated to *seq Byte*). To show this, a more elaborate refinement rule than Rule 8 is required, details of which may be found in [8]. As well as requiring that data refinement hold between corresponding observable actions of the abstract and concrete systems, the rule has two further important requirements. Firstly, each internal action must be a data refinement of *skip* under the same abstraction

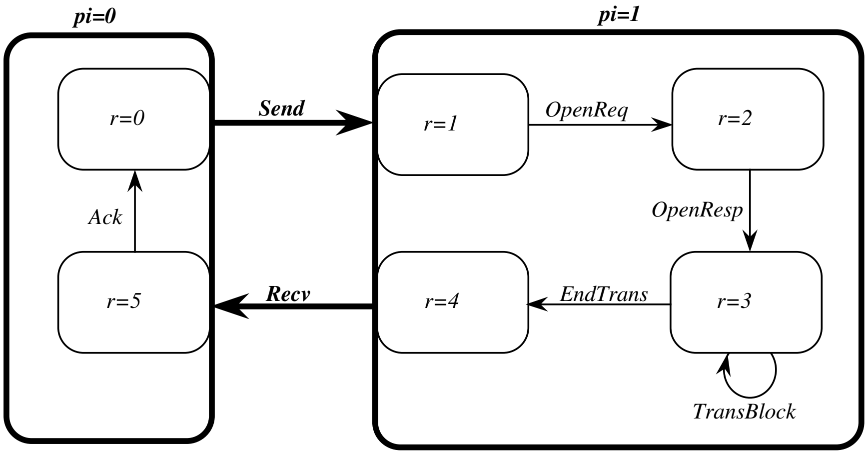

Figure 9: Abstract and concrete state transitions for the file transfer protocol.

invariant used for the observable actions. Secondly, in order to ensure that it is not possible to reach a state from which the internal actions may be executed infinitely often, thereby introducing 'livelock', there must be some variant expression that is decreased by each internal action under some well-founded ordering.

If we ignore the event-ordering terms, it is not possible to show that the system *FileTransferProtocol* is a refinement of *FileTransferService* since clearly the internal actions on their own can introduce livelock (e.g., the *OpenReq* action is always enabled). Even if the internal actions were such that they did not introduce livelock, we would need to know that *bfile* has the same value as the most recently sent file whenever *Recv* is invoked.

The refinement can easily be checked using the standard rule if both systems are normalised. The abstract system is normalised by introducing a pair of variables pi and pj with pi determining flow of control and pj being used to store the input file temporarily. The concrete system is normalised by introducing a single variable r which determines flow of control for the internal, as well as the observable, actions.

The abstraction invariant used to relate the abstract pi with the concrete r is illustrated by the state transition diagram of Fig. 9. Here, the heavy boxes represent the states of the abstract pi while the lighter boxes represent the states of the concrete r. The diagram shows that the abstract state $pi = 0$ corresponds to two concrete states, $r \in \{0, 5\}$, while the abstract state $pi = 1$ corresponds to four concrete states, $r \in \{1, 2, 3, 4\}$. The diagram also shows how the abstract and concrete states evolve through the various observable and internal actions.

To relate the data components of the systems, the following invariant is used:

$$\begin{array}{rcl} r \in \{1,2\} & \Rightarrow & pj = \mathit{afile} \\ r = 3 & \Rightarrow & pj = \mathit{bfile} ^\frown \mathit{afile} \\ r = 4 & \Rightarrow & pj = \mathit{bfile}. \end{array}$$

To ensure that the internal actions do not introduce livelock, the pair $(r, \#\mathit{afile})$, with lexicographic ordering, is used as a variant. The second, lower-order, component, $\#\mathit{afile}$, is ordered by the standard natural number ordering, while r is ordered by the following well-founded order:

$$1 \succ 2 \succ 3 \succ 4, \qquad 5 \succ 0.$$

Note that the observable actions, *Send* and *Receive*, are not required to decrease the invariant.

5 Conclusions

We introduced a simple language of event-ordering terms which may be added to an action system to constrain the ordering of events. Such a term provides a more transparent way of writing event-ordering constraints. We showed how an action system with an event-ordering term may be normalised to a standard action system by introducing abstract program counters to constrain flow of control. Although this normalisation may appear crude, it is straightforward (indeed, it should be easy to automate) and it is essential in order to be able to perform many useful refinements.

The approach of SL_0 and CSP-OZ mentioned in the introduction is related to ours though, unlike ours, it does not allow event ordering to be used when refining the state part. The idea of modelling a program counter more abstractly has been used in action systems elsewhere [20]. There, a binary sequencing operator is introduced which is really a shorthand for the appropriate use of a program counter. This allows for less general sequencing constraints than our approach. In Lamport's TLA [13], a state machine is constrained by temporal logic formulae rather than event-ordering terms and rules are provided for eliminating these in refinement. Abrial and Mussat [1] also provide rules for checking that a B machine satisfies simple temporal properties.

CCS and CSP provide operators for event hiding and parallel composition, as well as choice and prefixing, and CSP also provides a separate internal choice operator. For simplicity we have not considered supporting these in our term language. Although hiding, parallel composition, and internal choice are already available for the state part of an action system, it could be useful to make them available in the event-ordering term to be able to take advantage of existing algebraic laws from process algebras.

Acknowledgements

Thanks to Jean-Raymond Abrial, Stefan Hallerstede, Ric Hehner and Louis Mussat for some useful discussions.

References

[1] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *Second International B Conference*, April 1998.

[2] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 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. *Science of Computer Programming*, 13:133–180, 1989.

[7] M.J. Butler. *An Approach to the Design of Distributed Systems with B AMN (extended version)*. Technical Report DSSE-TR-96-6, http://www.dsse.ecs.soton.ac.uk/techreports/96-6.html, University of Southampton, 1996. Shorter version appears in proceedings of 10th International Conference of Z Users (ZUM'97), 3-4th April 1997, University of Reading, UK.

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

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

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

[11] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.

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

[13] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[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. *Programming from Specifications, Second Edition*. Prentice–Hall, 1994.

[17] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):298–306, 1987.

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

[19] 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.

[20] J. Plosila, R. Ruksenas, and K. Sere. *Action Systems Synthesis of DI Circuits*. Technical Report 149, http://www.tucs.abo.fi/publications/techreports/TR149.html, Turku Centre for Computer Science (TUCS), December 1997.

[21] 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.