Refinement of Parallel and Reactive Programs

R. J. R. Back

Computer Science Department
California Institute of Technology

Caltech-CS-TR-92-23

Refinement of Parallel and Reactive Programs

R.J.R. Back

Abstract

We show how to apply the refinement calculus to stepwise refinement of parallel and reactive programs. We use action systems as our basic program model. Action systems are sequential programs which can be implemented in a parallel fashion. Hence refinement calculus methods, originally developed for sequential programs, carry over to the derivation of parallel programs. Refinement of reactive programs is handled by data refinement techniques originally developed for the sequential refinement calculus. We exemplify the approach by a derivation of a mutual exclusion algorithm.

1 Introduction

The *action system* formalism [7, 8] describes the behavior of parallel and distributed programs in terms of the actions that can take place in the system. Two or more actions can be executed in parallel, as long as the actions do not have any variables in common. The actions are *atomic*: if an action is chosen for execution, it is executed to completion without any interference from the other actions in the system.

Atomicity guarantees that a parallel execution of an action system gives the same results as a sequential and nondeterministic execution. We can therefore treat a parallel action system as if it was a sequential program statement. This allows us to use the sequential refinement calculus introduced in [2] for stepwise refinement of parallel systems.

The refinement calculus is based on the assumption that the notion of correctness we want to preserve is total correctness. This is appropriate for *parallel algorithms*, i.e., programs that differ from sequential algorithms only in that they are executed in parallel, by co-operation of many processes. They are intended to terminate, and only the final results are of interest. Parallelism is introduced by superposition of action systems and refining the atomicity of actions. This approach to stepwise refinement of parallel algorithms has been put forward by Back and Sere [4, 10, 12].

In this paper we will show how the stepwise refinement method for action systems can be extended to stepwise refinement of reactive systems. Our starting point is the approach to refining reactive programs by refinement mappings put forward by Lamport in [27] and further developed by Abadi and Lamport [1], Stark [36], Jonsson [24, 25], Lynch and Tuttle [29] and Lam and Shankar [26]. We will show that refinement of reactive systems can be seen as a special case of the general method for data refinement [20, 23, 19]. Data refinement in the framework of refinement calculus is considered in [2, 33, 3, 34, 17, 13, 18]. The work described here is based on earlier work presented in [5].

The action system framework is described in Section 2. In Section 3 we consider parallel execution of action systems. In Section 4 we describe the basic operations for composing reactive action systems, parallel composition and hiding. Section 5 describes the method of data refinement for preserving total correctness, as applied to action systems. In Section 6 we consider refinement of reactive systems, where the behavior of an action system needs to be preserved in

a reactive context. Section 7 contains a case study of refinement of reactive systems. We show how to refine the atomicity of an action system by implementing a protocol that enforces mutual exclusion of the critical sections in the actions. We end with some concluding remarks in Section 8.

2 Action system

An *action system* is a statement of the form

$$\mathcal{A} = \texttt{begin var } x := x_0; \texttt{ do } A_1 \texttt{ [] } \ldots \texttt{ [] } A_m \texttt{ od end} : z. \tag{1}$$

Here x are the *local variables* of $\mathcal{A}$, initialized to x_0, z are the *global variables* of $\mathcal{A}$ and $A_1, \ldots, A_m$ are the *actions* (or *guarded commands*) of $\mathcal{A}$. Each action is of the form

$$A_i = g_i \rightarrow S_i,$$

where g_i is the *guard* of the action and S_i is the *statement* (or *body*) of the action. We denote the *guard* of action A by gA and the *statement* of it by sA, so $A = gA \rightarrow sA$.

The local and global variables are assumed to be distinct, i.e., $x \cap z = \emptyset$. The local and global variables together form the *state variables* y , $y = x \cup z$. The set of state variables accessed in action A is denoted vA.

An action system provides a global description of the system behavior. The state variables determine the state space of the system. The actions determine what can happen during an execution. The execution terminates when no action is enabled anymore. The initialization could be made more ellaborate, either permitting an arbitrary initialization statement, or then an assignment of values to the local values that nondeterministically establishes some condition, but the form chosen here has the advantage of simplicity.

We assume that the body of each action A is *strict*, i.e., $sA(\texttt{false}) = \texttt{false}$, and *positively conjunctive*, i.e., $sA(\bigwedge_i Q_i) = \bigwedge_i sA(Q_i)$ for any nonempty set $\{Q_i\}$ of predicates. The first assumption can be done without loss of generality. If sA is not strict, then we can write A in the equivalent form $g' \rightarrow S'$, where $g' = gA \wedge \neg sA(\texttt{false})$ and $S' = \{\neg sA(\texttt{false})\}; sA$, where S' is strict. The second assumption is a real restriction on the language of action systems.

The action system formalism is quite general: The body of an action may be an arbitrary, possibly nondeterministic statement and it may be nonterminating. The action system itself may or may not terminate.

Example Figure 1 shows an example of a simple sorting program (exchange sort) described as an action system. This program will sort the values of $x.1, \ldots, x.n$ in nondecreasing order. The $n-1$ sorting actions exchange neighboring values if they are out of order. The program terminates when all values are in nondecreasing order. All variables are global in this simple example. Figure 2 shows the *access relation* of the system, i.e., the way in which the actions access the state variables.

3 Parallel execution of action systems

Action systems may also be executed in parallel. If two actions A_i and A_j that have no state variables in common are both enabled, they may be executed in either order or at the same time. Such a parallel execution cannot produce any result that could not be produced by a sequential execution.

$\mathcal{A}$: **begin**
do
(‖ $x.i > x.(i+1) \rightarrow x.i, x.(i+1) := x.(i+1), x.i$ $[EX.i]$
for $i : 1, \ldots, n-1$)
od
end : $x.1, \ldots, x.n \in$ *integer*.

Figure 1: Exchange sorting

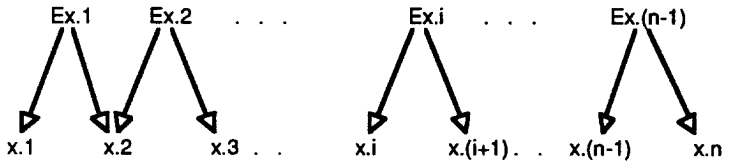

Figure 2: Access relation for exchange sorting

Distributed systems Consider again the action system $\mathcal{A}$ in (1). Let

$$\mathcal{P} = \{p_1, \ldots, p_r\}$$

be a partitioning of the state variables y of $\mathcal{A}$ into disjoint sets. We refer to each p_i as a *process*. Intuitively, we identify a process with the set of state variables local to the process. We refer to the pair $(\mathcal{A}, \mathcal{P})$ as a *partitioned action system*.

The action A is said to *involve* process p if $vA \cap p \neq \emptyset$, i.e., if A accesses some variable in p. Let pA be the set of processes involved in A, $pA = \{p \in \mathcal{P} | vA \cap p \neq \emptyset\}$.

An action A that involves only one process p is said to be *private* to p. If A involves two or more processes, it is said to be *shared* between these. Two actions A_i and A_j are said to be *independent*, if $pA_i \cap pA_j = \emptyset$. The actions are *competing* if they are not independent.

A shared action corresponds to a generalized handshake, executed jointly by all the processes involved in it. The processes must be synchronized for execution of such an action. Shared actions also provide communication between processes: a variable in one process may be updated in a way that depends on variables in other processes involved in the shared action. This model generalizes conventional synchronous message passing models for distributed systems such as CSP [21].

A *parallel execution* of a partitioned action system is any execution where only independent actions are executed in parallel. Independent actions do not have any processes in common, so they cannot have any state variables in common either. Different partitionings of the state variables will induce different parallel executions for the same action system.

As an example, consider the example program above, with the variables partitioned into the sets $\{x.1, x.2\}, \{x.3\}, \{x.4\}, \ldots, \{x.(n-2)\}, \{x.(n-1), x.n\}$ (Figure 3). Then the action $Ex.1$ is private to the first process and action $Ex.(n-1)$ is private to the last process. All other actions are shared between two neighboring processes and require a synchronizing handshake for execution.

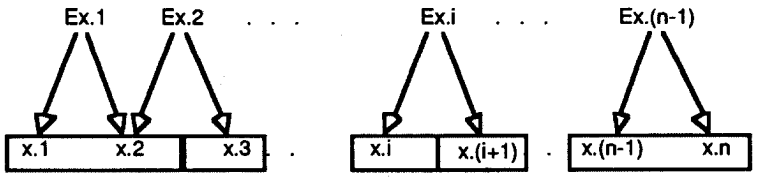

Figure 3: Distributed sorting

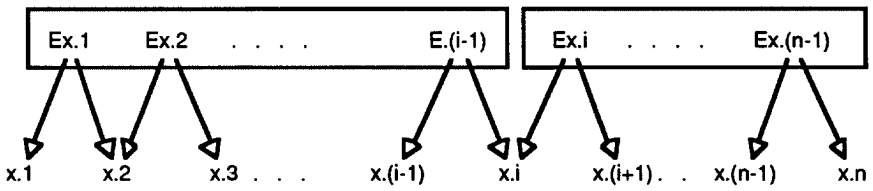

Figure 4: Shared variable sorting

Shared variable model By partitioning the actions rather than the variables, we get a *concurrent system* with shared variables. Let

$$\mathcal{P} = \{p_1, \ldots, p_r\},$$

where each p_i is a set of actions. A variable will be *shared* in the partitioned action system $(\mathcal{A}, \mathcal{P})$, if it is accessed by two or more actions, otherwise it is private. Shared variables may only be accessed under mutual exclusion and the actions must be executed atomically.

As an example, we partition the actions in the sorting program into two processes, one containing the actions $E_0, \ldots, E_{i-1}$ and the other containing actions $E_i, \ldots, E_{n-1}$. The first process sorts the low end of the array, and the second sorts the high end. They use a shared variable $x.i$ to exhange values between the low and high end.

Implementing action systems Partitioning is thus sufficient to describe different kinds of parallel execution models. The way in which these models are implemented may of course be very different depending on the view taken. A distributed implementation requires that synchronizing multiprocess handshakes are implemented, while a shared variable implementation requires a protocol that guarantees mutual exclusion for shared variables and atomicity of actions.

Distributed implementations of action systems are described by Back and Kurki-Suonio [7] for two-process actions in CSP with output guards. Efficient implementations of so-called *decentralized action systems* on broadcasting networks are presented in [8]. Implementations of action systems on point-to-point networks are described by Bagrodia [14]. Implementation of action systems in occam (which does not permit output guards) is described by Back and Sere in [11].

4 Composing action systems

We will take the shared variable partitioning model as the basis for structuring action systems into a hierarchy of interacting systems.

Parallel composition Given two action systems $\mathcal{A}$ and $\mathcal{B}$,

$$\begin{aligned} \mathcal{A} &= \textbf{begin var } x := x0; \textbf{ do } A_1 \,[\!]\, \ldots \,[\!]\, A_m \textbf{ od end} : z \\ \mathcal{B} &= \textbf{begin var } y := y0; \textbf{ do } B_1 \,[\!]\, \ldots \,[\!]\, B_k \textbf{ od end} : u, \end{aligned}$$

we define their *parallel composition* $\mathcal{A} \parallel \mathcal{B} : z \cup u$ to be

$$\textbf{begin var } x, y := x0, y0; \textbf{ do } A_1 \,[\!]\, \ldots \,[\!]\, A_m \,[\!]\, B_1 \,[\!]\, \ldots \,[\!]\, B_k \textbf{ od end} : z \cup u$$

This is the same as the union operator in UNITY [16], except that we also keep track of which variables are local and which are global (UNITY has only global variables). We assume that $x \cap y = \emptyset$ (this can always be achieved by renaming local variable). While the local variables are kept distinct, the global variables are shared among the processes in the parallel composition.

Renaming We may *rename* the global variables of an action systems. If $\mathcal{A}$ is an action system on global variables z, then $\mathcal{A}[z'/z]$ is the action system on the global variables z' (a list of distinct variables) which we get by replacing in $\mathcal{A}$ each occurrence of a variable in z by the corresponding variable in z', renaming local variables if necessary to avoid capture of global variables.

Hiding Given an action system $\mathcal{A} : z$ of the form above, we can *hide* some of its variables by making them local. If $z = u, v$, then the hiding of variables u with initialization u_0 produces the action system $\textbf{begin var } u := u0; \mathcal{A} \textbf{ end} : v$, defined as

$$\textbf{begin var } x, u := x0, u0; \textbf{ do } A_1 \,[\!]\, \ldots \,[\!]\, A_m \textbf{ od end} : v.$$

Hiding the variables u makes them inaccessible to actions outside $\mathcal{A}'$ in a parallel composition.

Decomposing action systems Given an action system

$$\mathcal{C} = \textbf{begin var } v := v0; \textbf{ do } C_1 \,[\!]\, \ldots \,[\!]\, C_n \textbf{od end} : z,$$

we can *decompose* it into smaller action systems by parallel composition and hiding. Let $AS = \{A_1, \ldots, A_m\}$ and $BS = \{B_1, \ldots, B_k\}$ be a partitioning of the actions in C. Let

$$\begin{aligned} x &= vAS - vBS - z \\ y &= vBS - vAS - z \\ w &= vAS \cap vBS - z \end{aligned}$$

We can then write $\mathcal{C}$ as

$$\mathcal{C} = \textbf{begin var } w := w0;\ \mathcal{A} \parallel \mathcal{B} \textbf{ end} : z,$$

where

$$\begin{aligned} \mathcal{A} &= \textbf{begin var } x := x0; \textbf{ do } AS \textbf{ od end} : z, w \\ \mathcal{B} &= \textbf{begin var } y := y0; \textbf{ do } BS \textbf{ od end} : z, w \end{aligned}$$

The main advantage of using blocks with local variables is that it permits us to clearly state which variables are used by which actions. The difference, as compared to the process algebra framework ([31, 22]) is that communication is by shared variables rather than by shared actions. Hence, hiding really means hiding variables, to prevent access to them, rather than hiding actions.

5 Refinement of action systems

Our purpose here is to show how the method of data refinement in the refinement calculus can be applied to the refinement of action systems. We will use data refinement in a form that permits *stuttering actions* to be introduced in a refinement and which also takes into account the context in which the action system occurs.

Notation The alternative action $A_1 \,[\!]\, \ldots \,[\!]\, A_m$ is an action itself,

$$A_1 \,[\!]\, \ldots \,[\!]\, A_m = \bigvee_{i=1}^{m} gA_i \rightarrow \texttt{if}\ \ A_1 \,[\!]\, \ldots \,[\!]\, A_m\ \texttt{fi}$$

We write $gA = \bigvee_i gA_i$ and $sA = \texttt{if}\ \ A_1 \,[\!]\, \ldots \,[\!]\, A_m\ \texttt{fi}$. This permits us to consider the whole action system A as consisting of a single action A, i.e., $\mathcal{A} = \texttt{begin var}\ \ x;\ S_0;\ \texttt{do}\ \ A\ \texttt{od end} : z$. We have that $vA = \bigcup_i vA_i$.

Data refinement Let A be a statement on the program variables x, z and A' a statement on the program variables x', z. Let $R(x, x', z)$ be a relation on these variables (*the abstraction relation*). Then A is *data refined* by A using R, denoted $A \leq_R A'$, if

$$(\forall q.\ R \wedge A\, q \Rightarrow A'\, (\exists x.\ R \wedge q)).$$

(where $(\exists x.\ R \wedge q)$ is understood to be a predicate on the program variables x', z.) When A and A' are actions, then this is equivalent to the following two conditions:

(i) *Refinement of guards:* $R \wedge gA' \Rightarrow gA$ and

(ii) *Refinement of bodies:* $(\forall q.\ R \wedge gA' \wedge sA\, q \Rightarrow sA'\, q)$.

A refinement may thus strengthen the guard of an action, decrease nondeterminism and increase termination of the body.

Data refinement of action systems The rule for data refinement of action systems is as follows: Let

$$\begin{aligned} \mathcal{A} &= \texttt{begin var}\ \ x := x_0;\ \texttt{do}\ \ A\ \texttt{od end} : z \\ \mathcal{A}' &= \texttt{begin var}\ \ x' := x_0';\ \texttt{do}\ \ A'\ \texttt{od end} : z. \end{aligned}$$

Then $\mathcal{A} \leq \mathcal{A}'$ if there exists a relation $R(x, x', z)$ such that

(i) *Initialization:* $R(x_0, x_0', z)$,

(ii) *Main action:* $A \leq_R A'$ and

(iii) *Exit conditions:* $R \wedge gA \Rightarrow gA'$.

The first condition requires that the abstraction relation is established by the initialization (for any initial value of z). The second condition requires that the action A' is data refined by the action A using R. The third condition requires that the continuation condition for $\mathcal{A}$ implies the continuation condition for $\mathcal{A}'$ whenever R holds (or, alternatively, that the exit condition of $\mathcal{A}'$ implies the exit condition of $\mathcal{A}$).

Permitting stuttering This relation of data refinement is, however, often too restrictive. The problem is that it requires a one to one correspondence between the actions executed by $\mathcal{A}$ and by $\mathcal{A}'$. In practice, executing a simple action in $\mathcal{A}$ will often correspond to executing a sequence of two or more actions in $\mathcal{A}'$, so that the one to one correspondence is not maintained.

We overcome this problem by permitting *stuttering actions* in $\mathcal{A}'$, actions which do not correspond to any global state change in $\mathcal{A}$. For any execution of an action system $\mathcal{A}$, the meaning of $\mathcal{A}$ is unchanged if we permit a finite number of *skip* actions (stutterings) to be inserted in the execution. We may not, however, add an infinite sequence of successive stutterings, because this would give rise to internal divergence, even when the original action system was guaranteed to terminate.

The rule for data refinement of action systems with stuttering is as follows. Let

$$\begin{aligned} \mathcal{A} &= \textbf{begin var } x := x_0; \textbf{ do } A \textbf{ od end} : z \\ \mathcal{A}' &= \textbf{begin var } x' := x_0'; \textbf{ do } A' \,[\!]\, H' \textbf{ od end} : z. \end{aligned}$$

Here H' is a stuttering action. Then $\mathcal{A} \leq \mathcal{A}'$ if there exists a relation $R(x, x', z)$ such that

(i) *Initialization:* $R(x_0, x_0', z)$,

(ii) *Main action:* $A \leq_R A'$,

(iii) *Exit conditions:* $R \wedge gA \Rightarrow gA' \vee gH'$,

(iv) *Auxiliary actions:* $skip \leq_R H'$ and

(v) *Internal convergence:* $R[\textbf{do } H' \textbf{ od}]true$.

We write $\mathcal{A} \preceq_R \mathcal{A}'$ when the conditions (i) —(v) hold.

The continuation condition is changed to reflect the fact that if the original action is enabled, then either the main action or the stuttering action in the refinement should be enabled. The fourth condition requires that the auxiliary actions are stuttering statements, in the sense that they act as skip statements on the global variables z. The last condition requires that the stuttering actions, when left to themselves, must necessarily terminate, to prevent internal divergence unless it is already present in the original action system. The correctness of this proof rule is established in [6, 38].

Data refinement in context When the action system $\mathcal{A}$ occurs in a parallel composition with other action systems, then the requirements above are not sufficient. They are based on the assumption that the action system is executed in isolation, as a closed system. To take the context into account, we have to add one more condition on the data refinement.

Let $\mathcal{A}$ and $\mathcal{A}'$ be as before, and let $\mathcal{B}$ be another action system,

$$\mathcal{B} = \textbf{begin var } y := y_0; \textbf{ do } B \textbf{ od end} : z$$

Then $\mathcal{A}||\mathcal{B} \leq \mathcal{A}'||\mathcal{B}$ if there exists a relation $R(x, x', z)$ such that $\mathcal{A} \preceq_R \mathcal{A}'$ and, in addition,

(vi) *Non-interference:* $R \wedge B\,true \Rightarrow B\,R$,

This condition guarantees that the interleaved execution of actions from B preserves the abstraction relation R. This requirement is analogous to the non-interference condition introduced in the Owicki-Gries proof theory for parallel programs [35].

Superposition Superposition refinement [7, 16, 12] of parallel systems is a special case of this more general notion of data refinement. In superposition, one may add new variables, but no old variables can be removed. The abstraction relation then degenerates to a invariant on the concrete variables.

6 Trace refinement

Traces of action systems Consider the (*initialized*) action system $\mathcal{A}$, defined as

$$z := z_0; \textbf{begin var } x := x_0; \textbf{ do } A \textbf{ od end} : z$$

A computation of the action system $\mathcal{A}$ is either a finite sequence

$$(x_0, z_0), (x_1, z_1), \ldots, (x_n, z_n)$$

where (x_n, z_n) satisfies the exit condition (a *successful computation*), a finite sequence

$$(x_0, z_0), (x_1, z_1), \ldots, (x_n, z_n), \bot,$$

where $\bot$ indicates that abortion occurred in state (x_n, z_n) (a *failed computation*), or an infinite sequence

$$(x_0, z_0), (x_1, z_1), (x_2, z_2), \ldots$$

(an *infinite computation*), where no abortion occurs and the exit condition is not satisfied in any state.

A computation c determines a *trace* of the action system. This is the sequence of (global) z values that we get by removing the hidden component x and also removing all finite sequences of repeated z values (finite stuttering), but leaving $\bot$ if it is present, as well as any infinite trailing sequence of stuttering values. Let us denote by $tr(\mathcal{A})$ the set of all traces of action system $\mathcal{A}$. We say that an action system is *robust*, if it cannot produce a failed trace.

Trace specifications A *trace specification* of an action system is a set T of sequences of z values, (without trailing $\bot$ element). We say that the action system $\mathcal{A}$ satisfies the specification T if $tr(\mathcal{A}) \subseteq T$.

A specification Q does not contain any failed traces, so an action system that has a failed trace will not satisfy any specification. Thus, only robust systems are accepted as implementations of a specification.

Trace refinement We say that the (initialized) action system $\mathcal{A}$ is *trace refined* by the (initialized) action system $\mathcal{A}'$, denoted $\mathcal{A} \sqsubseteq \mathcal{A}'$, if

$$(\forall T.\, tr(\mathcal{A}) \subseteq T \Rightarrow tr(\mathcal{A}') \subseteq T).$$

This is equivalent to the following condition:

$$\mathcal{A}\ robust \Rightarrow tr(\mathcal{A}) \supseteq tr(\mathcal{A}').$$

Thus robustness is preserved in a refinement, while the set of different traces of an action system may decrease. An initialized action system must always have at least one trace, so refinement cannot result in an empty set of traces.

Proving trace refinement Data refinement guarantees refinement of action systems, in the sense of preserving total correctness. In fact, data refinement is even stronger than this, because it will also preserve trace correctness. More precisely, if $\mathcal{A} \preceq_R \mathcal{A}'$, then $\mathcal{A} \sqsubseteq \mathcal{A}'$. This extends also to data refinement in context: the same conditions that guarantee that an action system refines another in a parallel context, are sufficient to guarantee that trace correctness is preserved.

Data refinement as described here is in fact *forward data refinement* (or *downward simulation.* A dual method is *backward data refinement* or (*upward simulation*) [15]. Under certain assumptions, these two methods together provide a complete method for trace refinement [25, 32, 37]

We may use data refinement for stepwise refinement of action systems both when we want to preserve total correctness, and when we want to preserve trace correctness. If we only need to preserve total correctness, then we may use other refinement steps besides data refinement, but if we have to preserve trace correctness, then we are restricted to data refinement.

7 Case study: Mutual exclusion

Initial system We apply the above techniques to show how to refine the atomicity of a system with the help of a classical mutual exclusion algorithm (Peterson's algorithm).

Let us consider the following action system:

$$
\begin{array}{ll}
MS_0: & \mathbf{begin}\,(\mathbf{var}\ y.i \in \mathit{Int}, cr.i \in \mathit{Bool}\ \mathbf{for}\ i = 0,1); \\
& (cr.i := \mathit{false}\ \mathbf{for}\ i = 0,1); \\
& \mathbf{do} \\
& (\ [\!]\ \ cr.i \rightarrow y.i := w + i + 1;\ w := y.i;\ cr.i := \mathit{false}\ \mathbf{for}\ i = 0,1) \quad [CS.i] \\
& (\ [\!]\ \ \neg cr.i \rightarrow N.i\ \mathbf{for}\ i = 0,1) \quad [NS.i] \\
& \mathbf{od} \\
& \mathbf{end} : z, w \in \mathit{Int}.
\end{array}
$$

Besides w, there may be some other globally observable variables z, which are not specificed further. The action $N.i$ may or may not set $cr.i$ to true, and does not affect $cr.j$, $j \neq i$. The effect is that $NS.i$ may execute any number of times before the corresponding $CS.i$ action is executed, and may also execute forever or terminate before the $CS.i$ action is executed.

Our task is to derive an implementation that preserves the trace correctness of the original solution, but updates w in two separate actions, one where $y.i := w + i + 1$ and the other where $w := y.i$. We assume that the action $NS.i$ does not access $w, y.0, y.1$, for $i = 0, 1$.

We will assume that the system MS_0 is closed, i.e., it is not executed in parallel with any other action system.

Problem with non-atomic update If the updates of $w.i$ are not performed atomically, then the sequence of updates

$$y.0 := w + 1;\ y.1 := w + 2;\ w := y.1;\ w := y.0$$

can take place. In an initial state $w = 0$, this would then give the sequence of w-values $0, 2, 1, \ldots$. Thus, we could observe a decrease in the w-value, which is not possible to observe in the original action system. To avoid this phenomenon, we need to treat $y.i := w + i + 1;\ w := y.i$, $i = 0, 1$, as critical sections that should be executed under mutual exclusion.

Identifying components We will start by partitioning the system into reactive (parallel) components.

MS_0 : **begin** (**var** $cr.i \in Bool$ **for** $i = 0, 1$);
$(cr.i := false$ **for** $i = 0, 1)$;
$CS_0 || NS_0$
end : z, w.

Here we define

CS_0 : **begin** (**var** $y.i \in Int$ **for** $i = 0, 1$);
do
(▌ $cr.i \rightarrow y.i := w + i + 1$; $w := y.i$; $cr.i := false$ **for** $i = 0, 1$) $[CS.i]$
od
end : $(cr.i$ **for** $i = 0, 1), w$.

and

NS_0 : **begin**
do
(▌ $\neg cr.i \rightarrow N.i$ **for** $i = 0, 1$); $[NS.i]$
od
end : $(cr.i$ **for** $i = 0, 1), z$.

Refining the critical section We refine the critical section part of the system. We add new variables and actions, in preparation for refining the atomicity of the system. This step is an example of a pure superposition step: no variables are reimplemented, only new variables are added. The auxiliary actions are $BS.i$, $TS.i$ and $BR.i$, $i = 0, 1$.

CS_1 : **begin** (**var** $b.i \in Bool, pc.i, y.i \in Int$ **for** $i = 0, 1$); $t : 0 \ldots 1$;
$(b.i, pc.i := false, 0$ **for** $i = 0, 1)$;
$(t := 0 \wedge t := 1)$;
do
(▌ $cr.i \wedge pc.i = 0 \rightarrow b.i := true$; $pc.i := 1$ **for** $i = 0, 1$) $[BS.i]$
(▌ $pc.i = 1 \rightarrow t := i$; $pc.i := 2$ **for** $i = 0, 1$) $[TS.i]$
(▌ $pc.i = 2 \wedge (\neg b.(1 - i) \vee t = 1 - i) \rightarrow y.i := w + i + 1$; $w := y.i$; $[CS'.i]$
$cr.i := false$; $pc.i := 3$ **for** $i = 0, 1$)
(▌ $pc.i = 3 \rightarrow pc.i := 0$; $b.i := false$ **for** $i = 0, 1$) $[BR.i]$
od
end : $(cr.i$ **for** $i = 0, 1), w$.

Correctness of refinement step We need to show that this refinement step is correct. The abstraction relation will be just an invariant on the new variables, because no variables in the old version are being replaced. The invariant R is described by the following table:

$pc.i$	$b.i$	$cr.i$	t
0	F	F, T	$0, 1$
1	T	T	$0, 1$
2	T	T	$0, 1$
3	T	F, T	$0, 1$

We check that the conditions for data refinement in context are satisfied.

(i) [*Initialization:* $R(x_0, x_0', z)$] The initialization obviously establishes the invariant.

(ii) [*Main action:* $A \leq_R A'$] The action $CS'.i$ is a data refinement of the original action: By the invariant, the guard of the new action implies the guard of the old. The effect is the same as that of the old action on global variables, and the invariant is preserved.

(iii) [*Exit conditions:* $R \wedge gA \Rightarrow gA' \vee gH'$] Assume that $cr.0$ is true. Then, if $pc.0 = 0$, we are done, as action $BS.0$ is enabled. Otherwise, if $pc.0 = 1$, action $TS.0$ is enabled and if $pc.0 = 3$, then action $BR.0$ is enabled. Assume that $pc.0 = 2$. If $\neg b.1 \vee t = 1$ holds, then $CS'.0$ is enabled. Assume therefore that $b.1 \wedge t = 0$ holds. By the invariant, this means that $pc.1 \neq 0$. If $pc.1 = 1$, then $TS.1$ is enabled and if $pc.1 = 3$, then $BR.1$ is enabled. Assume $pc.1 = 2$. Then $t = 0$, so action $CS'.1$ is enabled. The analogous argument can be made for the case when $cr.1$ is true. Hence, we have proved the exit condition requirement.

(iv) [*Auxiliary actions: skip* $\leq_R H'$] The action $BS.i$ refines a skip-statement, because only new variables are affected. It preserves the invariant. Action $TS.i$ also refines a skip statement, and preserves the invariant, and the same holds for action $BR : i$.

(v) [*Internal convergence:* $R[\texttt{do } H' \texttt{ od}]true$] Executing only auxiliary actions will eventually terminate in a state where $pc.0$ and $pc.1$ each is either set to 2 or 3.

(vi) [*Non-interference:* $R \wedge B\ true \Rightarrow B\ R$] The invariant refers to the global variable $cr.i$, so we need to show that it is preserved by the environment actions. The action $NS.i$ can only be enabled when $pc.i = 0$ or $pc.i = 3$, in which case the invariant is preserved trivially, as $pc.i, b.i, t$ are local to $CS.i$ and therefore not changed by $NS.i$.

Thus, we have shown that CS_1 is a data refinement of CS_0, in the context of NS_0. Hence, trace correctness is preserved if we replace CS_0 by CS_1 in MS_0.

Refining the atomicity Next, we refine the atomicity of the system. We split up the action $CS'.i$ into three actions, $CAS.i$, $CBS.i$ and $CCS.i$.

CS_2 : **begin** (**var** $b.i \in Bool, pc.i, y.i \in Int$ **for** $i = 0, 1$); $t : 0 \ldots 1$;
$(b.i, pc.i := false, 0$ **for** $i = 0, 1)$;
$(t := 0 \wedge t := 1)$;
do
$(\ [\!]\ cr.i \wedge pc.i = 0 \rightarrow b.i := true;\ pc.i := 1$ **for** $i = 0, 1)$ $[BS.i]$
$(\ [\!]\ pc.i = 1 \rightarrow t := i;\ pc.i := 2$ **for** $i = 0, 1)$ $[TS.i]$
$(\ [\!]\ pc.i = 2 \wedge (\neg b.(1-i) \vee t = 1-i) \rightarrow pc.i := 3$ **for** $i = 0, 1)$ $[CAS.i]$
$(\ [\!]\ pc.i = 3 \rightarrow y.i := w + i + 1;\ pc.i := 4$ **for** $i = 0, 1)$ $[CBS.i]$
$(\ [\!]\ pc.i = 4 \rightarrow w := y.i;\ cr.i := false;\ pc.i := 5$ **for** $i = 0, 1)$ $[CCS.i]$
$(\ [\!]\ pc.i = 5 \rightarrow pc.i := 0;\ b.i := false$ **for** $i = 0, 1)$ $[BR.i]$
od
end : $(cr.i$ **for** $i = 0, 1), w$.

Abstraction relation for refinement step We need to show that this refinement is correct also. We have changed the program counter $pc.i$ in CS_1 to another program counter $pc'.i$ in CS_2 (which will have the same name, but is distinguished below from the original by a dash).

We have the following relation between $pc.i$ and $pc'.i$, $P.i$:

$pc.i$	$pc'.i$
0	0
1	1
2	2
2	3
2	4
3	5

The other variables are unchanged in the abstraction. The following is an invariant $I.i$ of the resulting action system, for $i = 0, 1$:

$$\begin{aligned} & pc'.i = 1 \vee pc'.i = 2 \Rightarrow b.i \\ \wedge \quad & pc'.i = 3 \Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc'.j = 1) \\ \wedge \quad & pc'.i = 4 \vee pc'.i = 5 \Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc'.j = 1) \wedge y.i = w + i + 1 \end{aligned}$$

Preservation of invariant We need to show that each action preserves the invariant $I.0 \wedge I.1$. Consider first the conjunct $I.0$. The environment actions cannot change this, because the only global variable in it, w, is not changed by the noncritical section action. Action $CAS.0$ will establish the invariant, and action $CBS.0$ will preserve it. Also, the actions $BS.0$, $TS.0$ and $BR.0$ obviously preserve $I.0$. For the actions of process 1, we have that $BS.1$ will set $pc'.1 = 1$ and therefore preserves the invariant, even if $b.1$ is set to true. Action $TS.1$ sets $t = 1$, and therefore preserves the invariant, even if $pc'.1$ is set to 2. Action $BR.i$ will set $b.1$ to false, thus establishing the disjunction in $I.0$, if necessary.

This leaves the actions $CAS.1$, $CBS.1$ and $CCS.1$. We want to show that they can only be enabled when $pc'.0 \neq 3, 4, 5$ (the mutual exclusion property), so they cannot invalidate $I.0$ either. This is checked by an analysis of the enabling conditions and the invariant. Assume that $CAS.1$, $CBS.1$ or $CCS.1$ is enabled. Then, we have that

$$b.1 \wedge (\neg b.0 \vee t = 0)$$

must hold, by the invariant. However, by the assumption and the invariant, also

$$b.0 \wedge (\neg b.1 \vee t = 1)$$

must hold. But the conjunction of these two conditions is equivalent to *false*. Hence, actions $CAS.1$, $CBS.1$ and $CCS.1$ are only enabled when $pc'.1 = 0, 1$, in which case they preserve the condition $I.0$.

An analogous argument shows that each action establishes $I.1$, if initially $I.0 \wedge I.1$ holds, so the invariant is preserved by each action.

Correctness of refinement step We show that the conditions for data refinement in context are satisfied. The abstraction relation R is $P.0 \wedge P.1 \wedge I.0 \wedge I.1$.

(i) [*Initialization:* $R(x_0, x_0', z)$] The initialization obviously establishes the abstraction relation and the invariant.

(ii) [*Main action:* $A \leq_R A'$] The main actions are BS, i, $TS.i$, $BR.i$ and $CCS.i$, for $i = 0, 1$. The effect of these actions on the global variables is the same as the effect of the original actions, whenever the invariant and the relation between program counters holds. For $CCS.i$, this depends on the fact that $y.i = w + i + 1$ holds prior to execution. The fact that all actions preserve the invariant was shown above.

(iii) [*Exit conditions:* $R \wedge gA \Rightarrow gA' \vee gH'$] It is sufficient to show that if $pc.i = 2 \wedge (\neg b.j \vee t = j)$, then one of the actions $CAS.i$, $CBS.i$, $CCS.i$ is enabled. We have by the abstraction that either $pc'.i = 2, 3, 4$. If $pc'.i = 2$, then the first action is enabled. Otherwise, one of the other is enabled.

(iv) [*Auxiliary actions: skip* $\leq_R H'$] The actions $CAS.i$, $CBS.i$ refine skip actions. This follows from the fact that they do not change any of the global variables of CS_2. The fact that they preserve the invariant was already shown.

(v) [*Internal convergence:* $R[\mathtt{do}\ H'\ \mathtt{od}]true$] Executing auxiliary actions alone, for $i = 0, 1$, will obviously terminate.

(vi) [*Non-interference:* $R \wedge B\ true \Rightarrow B\ R$] The invariant and the relation between program counters only refers to local variables, except for w, which, by assumption, $NS.i$ does not access.

Restructuring the action systems We will now recombine the critical section with the noncritical section, giving a refinement of the original closed system MS_0. The previous steps show that

MS_1 : **begin** (**var** $cr.i \in Bool$ **for** $i = 0, 1$);
$(cr.i := false$ **for** $i = 0, 1)$;
$CS_2 || NS_0$
end : z, w.

is a correct refinement of the system MS_0.

Expanding this gives us the action system

MS_2 : **begin** (**var** $cr.i \in Bool$ **for** $i = 0, 1$);
$(b.i \in Bool, pc.i, y.i \in Int$ **for** $i = 0, 1)$; $t : 0 \ldots 1$;
$(cr.i := false$ **for** $i = 0, 1)$;
$(b.i, pc.i := false, 0$ **for** $i = 0, 1)$;
$(t := 0 \wedge t := 1)$;
do
$(\ [\!]\ cr.i \wedge pc.i = 0 \rightarrow b.i := true;\ pc.i := 1$ **for** $i = 0, 1)$ $[BS.i]$
$(\ [\!]\ pc.i = 1 \rightarrow t := i;\ pc.i := 2$ **for** $i = 0, 1)$ $[TS.i]$
$(\ [\!]\ pc.i = 2 \wedge (\neg b.(1 - i) \vee t = 1 - i) \rightarrow pc.i := 3$ **for** $i = 0, 1)$ $[CAS.i]$
$(\ [\!]\ pc.i = 3 \rightarrow y.i := w + i + 1;\ pc.i := 4$ **for** $i = 0, 1)$ $[CBS.i]$
$(\ [\!]\ pc.i = 4 \rightarrow w := y.i;\ cr.i := false;\ pc.i := 5$ **for** $i = 0, 1)$ $[CCS.i]$
$(\ [\!]\ pc.i = 5 \rightarrow pc.i := 0;\ b.i := false$ **for** $i = 0, 1)$ $[BR.i]$
$(\ [\!]\ \neg cr.i \rightarrow N.i$ **for** $i = 0, 1)$ $[NS.i]$
od
end : z, w.

Regrouping We partition the action system anew, but now into processes $P.i$ for $i = 0, 1$:

MS_3 : **begin**(**var** $b.i \in Bool$ **for** $i = 0, 1$); $t : 0 \ldots 1$;
$(b.i := false$ **for** $i = 0, 1)$;
$(t := 0 \wedge t := 1)$;
$PS.0 || PS.1$
end : z, w.

Here the processes $PS.i$, $i = 0, 1$, are defined as follows:

PS_i : **begin var** $cr.i \in Bool, pc.i, y.i \in Int$;
$cr.i := false$; $pc.i := 0$;
do
$\quad [] \; cr.i \wedge pc.i = 0 \rightarrow b.i := true$; $pc.i := 1$ $\quad [BS.i]$
$\quad [] \; pc.i = 1 \rightarrow t := i$; $pc.i := 2$ $\quad [TS.i]$
$\quad [] \; pc.i = 2 \wedge (\neg b.(1-i) \vee t = 1-i) \rightarrow pc.i := 3$ $\quad [CAS.i]$
$\quad [] \; pc.i = 3 \rightarrow y.i := w + i + 1$; $pc.i := 4$ $\quad [CBS.i]$
$\quad [] \; pc.i = 4 \rightarrow w := y.i$; $cr.i := false$; $pc.i := 5$ $\quad [CCS.i]$
$\quad [] \; pc.i = 5 \rightarrow pc.i := 0$; $b.i := false$ $\quad [BR.i]$
$\quad [] \; \neg cr.i \rightarrow N.i$ $\quad [NS.i]$
od
end : $w, b.0, b.1, t, z$.

Simplifying processes The processes $PS.i$, $i = 0, 1$, may be simplified, by having just one program counter, removing $cr.i$. This is a pure data refinement, involving no globally visible variables:

PS'_i : **begin var** $pc.i, y.i \in Int$;
$pc.i := 5$;
do
$\quad [] \; pc.i = 0 \rightarrow b.i := true$; $pc.i := 1$ $\quad [BS.i]$
$\quad [] \; pc.i = 1 \rightarrow t := i$; $pc.i := 2$ $\quad [TS.i]$
$\quad [] \; pc.i = 2 \wedge (\neg b.(1-i) \vee t = 1-i) \rightarrow pc.i := 3$ $\quad [CAS.i]$
$\quad [] \; pc.i = 3 \rightarrow y.i := w + i + 1$; $pc.i := 4$ $\quad [CBS.i]$
$\quad [] \; pc.i = 4 \rightarrow w := y.i$; $pc.i := 5$ $\quad [CCS.i]$
$\quad [] \; pc.i = 5 \rightarrow pc.i := 6$; $b.i := false$ $\quad [BR.i]$
$\quad [] \; pc.i = 6 \rightarrow N'.i$ $\quad [NS.i]$
od
end : $w, b.0, b.1, t, z$.

Here $N'.i$ is statement $N.i$, but with each assignment $cr.i := false$ replaced with $pc.i := 0$. The data abstraction relation is the following:

$pc.i$	$cr.i$	$pc'.i$
0	F	6
0	T	0
1	T, F	1
2	T, F	2
3	T, F	3
4	T, F	4
5	T, F	5

This implementation will shrink the number of execution paths, by preventing the non-critical section to start before the critical section is completely finished, i.e. , also $b.i$ has been set to false. This avoids that the competing process has to wait for the other process to finish its non-critical section, thus speeding up things.

The result is the refined mutual exclusion algorithm

MS_4 : **begin**(**var** $b.i \in Bool$ **for** $i = 0, 1)$; $t : 0 \ldots 1$;
$b.i := false, i = 0, 1$;
$(t := 0 \wedge t := 1)$;
$PS'.0 || PS'.1$
end : z, w.

Sequential notation Finally, we may omit the program counters alltogether, keeping them implicit, and instead explicitly indicate the atomicity of statements. This gives us the final form of our program.

MS_5 : **begin** (**var** $b.i \in Bool$ **for** $i = 0, 1)$; $t : 0 \ldots 1$;
$b.i := false$ **for** $i = 0, 1)$;
$(t := 0 \wedge t := 1)$;
$PS''.0 || PS''.1$
end : z, w,

where processes $PS''.i$ are defined by

PS_i'' : **begin**
do
$\langle N'.i \rangle$; $\langle b.i := true \rangle$; $\langle t := i \rangle$; $\langle \neg b.(1 - i) \vee t = 1 - i \rightarrow skip \rangle$;
$\langle y.i := w + i + 1 \rangle$; $\langle w := y.i \rangle$; $\langle b.i := false \rangle$
od
end : $w, b.0, b.1, t, z$.

Final comments The final program satisfies the original requirements, in that we have $MS_0 \sqsubseteq MS_5$, by transitivity. The difference is that whereas the original program executed the updating of the w variable in a single atomic step, the final algorithm does the same update non-atomically. In fact, all accesses to shared variables between the processes $PS''.0$ and $PS''.1$ are done with a single reference to the shared variable. (The guard in the statement $\langle \neg b.(1-i) \vee t = 1-i \rightarrow skip \rangle$ does have two reference to shared variables, but in the action system framework, this action can be replaced by two actions with the same body but with each of the disjunct as guard.) This means that assuming mutual exclusion for a single read or write access to a variable is sufficient to guarantee atomicity of the actions in the action system.

The final solution implements Peterson's mutual exclusion algorithm, in order to permit the atomicity refinement. To prove the correctness of the refinement step, it was necessary to show mutual exclusion of the critical sections. Also, the refinement steps required us to show that no deadlock could occur (this is a consequence of the exit condition).

8 Concluding remarks

Refinement of reactive systems seems to be just a special case of data refinement, with two additional ingredients. First, it is not possible to refine the atomicity of a statement in such a way that two global variables previously updated in a single action will be updated in two separate actions. This would make visible a state that was previously hidden (the state where one of the globals has been updated and the other has not). Hence, the initial specification of an action system should not contain such a joint update if one wants to separate these updates in an implementation of the system.

The second difference has to do with fairness requirements. It seems sufficient to assume weak action fairness, because when modelling the action system as processes, this will correspond to a minimal progress property. This is different from purely sequential programs, where one does not require weak fairness of processes. Note that this only holds for reactive systems where processes are actually interpreted as parallel processes, or where the underlying scheduling mechanism, although sequential, is still assumed to be fair. We have ignored the issues of fairness in this context, but these are treated in, e.g., [1] and, in the context of refining action systems, in [5, 9].

Appart from these differences, the refinement of reactive systems does not introduce anything which is not already needed when arguing about refinement of ordinary sequential programs. The correctness notion to be preserved, trace inclusion, is stronger than what one preserves in usual total correctness refinement. However, by only using data refinement in derivations, this correctness notion will be preserved.

The correctness of the initial action system, i.e., that it satisfies some given trace specification, may be proved in almost any suitable logic, such as temporal logic[30], UNITY [16], TLA [28], as well as other possible logics where one can express properties of sets of (possibly infinite) state sequences. An other alternative is to consider the initial action system itself to be the specification, as is done in, e.g., process algebra frameworks [31, 22], and as we did in the case study.

Acknowledgements

The work reported here was supported by the FINSOFT III program sponsored by the Technology Development Centre of Finland. I would like to thank Robert Barta, Ulla Binau, Bengt Jonsson, Marcel van de Groot, Peter Hofstee, Reino Kurki-Suonio, Rustan Leino, Leslie Lamport, Alan Martin, Carroll Morgan, Amir Pnueli, Kaisa Sere, Jan van de Snepscheut and Joakim von Wright for very helpful discussions on the topics treated here. Frank Stomp deserves a special thank, for his insistence on including environment invariants in the model.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. 3rd IEEE Symp. on LICS*, Edinburgh, 1988.

[2] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[3] R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE, January 1989.

[4] R. J. R. Back. Refining atomicity in parallel algorithms. In *PARLE 89 Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, Eindhoven, the Netherlands, June 1989. Springer Verlag.

[5] R. J. R. Back. Refinement calculus II: Parallel and reactive programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer–Verlag, 1990.

[6] R. J. R. Back. Refinement calculus, lattices and higher order logic. Technical report, Marktoberdorf Summer School on Programming Logics, 1992.

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

[8] R. J. R. Back and R. Kurki-Suonio. Distributed co–operation with action systems. *ACM Transactions on Programming Languages and Systems*, 10:513–554, October 1988.

[9] R. J. R. Back and R. Kurki-Suonio. Superposition and fairness in reactive system refinement. In *Jerusalem conference on Information Technology*, Jerusalem, Israel, October 1990.

[10] R. J. R. Back and K. Sere. Refinement of action systems. In *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, Groningen, The Netherlands, June 1989. Springer–Verlag.

[11] R. J. R. Back and K. Sere. Deriving an occam implementation of action systems. In *Third BCS Refinement Workshop*, Lecture Notes in Computer Science. Springer–Verlag, January 1990.

[12] R. J. R. Back and K. Sere. Superposition refinement of parallel algorithms. In K. R. Parker and G. A. Rose, editors, *Formal Description Techniques IV*, IFIP Transaction C–2. North–Holland, 1992.

[13] R. J. R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science, pages 42–66. Springer–Verlag, 1990.

[14] R. Bagrodia. *An environment for the design and performance analysis of distributed systems.* PhD thesis, The University of Texas at Austin, Austin, Texas, 1987.

[15] J. H. C.A.R. Hoare and J. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.

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

[17] W. Chen and J. T. Udding. Towards a calculus of data refinement. In *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, Groningen, The Netherlands, June 1989. Springer–Verlag.

[18] P. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Comput. Sci.*, 87(1):143–162, 1991.

[19] D. Gries and J. Prins. A new notion of encapsulation. In *Proc. SIGPLAN Symp. Language Issues in Programming Environments*, June 1985.

[20] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[21] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[23] C. Jones. *Software Development: A Rigorous Approach.* Prentice–Hall International, 1980.

[24] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Dept. of Computer Systems, Uppsala University, Uppsala, 1987. Available as report DoCS 87/09.

[25] B. Jonsson. On decomposing and refining specifications of distributed systems. In *REX Workshop for Refinement of Distributed Systems,* volume 430 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, 1989. Springer–Verlag.

[26] S. S. Lam and A. U. Shankar. A relational notation for state transition systems. Technical Report TR-88-21, Dept. of Computer Sciences, University of Texas at Austin, 1988.

[27] L. Lamport. Reasoning about nonatomic operations. In *Proc. 10th ACM Conference on Principles of Programming Languages*, pages 28–37, 1983.

[28] L. Lamport. A Temporal Logic of Actions. Src report 57, Digital SRC, 1990.

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

[30] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proc.* 10^{th} *ACM Symp. on Principles of Programming Languages*, pages 141–154, 1983.

[31] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes of Computer Science.* Springer Verlag, 1980.

[32] C. Morgan and J. Woodcock. Of wp and CSP. In *Proceedings of VDM-91*, 1991.

[33] C. C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, January 1988.

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

[35] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[36] E. W. Stark. Proving entailment between conceptual state specifications. *Theoretical Comput. Sci.*, 56:135–154, 1988.

[37] J. von Wright. Data refinement and the simulation method. Reports on computer science and mathematics 138, Åbo Akademi, 1992.

[38] J. von Wright. Data refinement with stuttering. Reports on computer science and mathematics 137, Åbo Akademi, 1992.