Refinement Calculus, Part II: Parallel and Reactive Programs

R.J.R. Back

Ser. A, No 93, 1989

Inst. för Informationbehandling
Lemminkäinenkatu 14
SF-20520 Åbo, Finland

November 1989

ISBN 951-649-652-0
ISSN 0358-0563
Refinement Calculus, Part II: Parallel and Reactive Programs

P.J.R. Back
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 4, SF-20520 Turku, Finland
November 2, 1989

Abstract

It is shown how to apply the refinement calculus to stepwise refinement of both parallel programs and reactive programs. The approach is based on using the action systems model to describe parallel and reactive systems. Action systems are sequential programs which can be implemented in a parallel fashion. Hence the refinement calculus for sequential programs carries over to the parallel programs expressed in this framework. Refinement of reactive programs can be expressed and proved in the refinement calculus by using the methods of data refinement from the sequential refinement calculus.
## Contents

1 Introduction ............................................. 3

2 Parallel programs as action systems ................. 3
   2.1 Action system ....................................... 4
   2.2 Parallel execution of action systems .......... 5
   2.3 Fairness in action system executions ......... 7
   2.4 Stepwise refinement of parallel programs ... 8

3 Refinement of action systems ......................... 9
   3.1 Iteration statements ................................ 9
   3.2 Data refinement .................................... 10
   3.3 Action systems .................................... 11
   3.4 Refinement with fairness constraints .......... 12

4 Reactive programs as action systems ............... 12
   4.1 Parallel composition and hiding ............... 13
   4.2 Refinement of reactive components .......... 14
   4.3 Permitting stuttering ............................ 15
   4.4 Example: Correctness of parallel execution model ....... 16
   4.5 Refinement in context ........................... 18

5 Refinement of reactive systems ................. 19
   5.1 Simulation refinement ............................ 19
   5.2 Preserving temporal properties ............... 20

6 Example derivation .................................. 20
   6.1 Refinement by transformation .................. 21
   6.2 Refining a reactive component ............. 22

7 Concluding remarks ................................ 25
1 Introduction

In part I of this overview we have presented a lattice-theoretic framework for the refinement calculus. In this second part we apply this to the stepwise refinement of parallel and reactive programs. We will base our approach on the action system model for parallel programs.

The paradigm of parallelism for parallel and distributed computations was introduced by Back and Kurki-Suonio in [5] and further developed in [8,7]. The behavior of parallel and distributed programs is described in terms of the actions that processes in the system carry out in co-operating with each other. Several 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 non-deterministic execution. We can therefore describe a parallel action system as a sequential statement in the language of commands. This allows us to use the sequential refinement calculus for stepwise refinement of action systems. We can start our derivation from a more or less sequential algorithm and successively increase the degree of parallelism in it, while preserving the correctness of the algorithm.

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 merging action systems and refining the atomicity of actions. This approach to stepwise refinement of parallel algorithms has been put forward by Back and Sere [8,12].

In this paper we will concentrate on showing how the stepwise refinement method for action systems can be extended to also cover the stepwise refinement of reactive systems. Our starting point is the approach to refining reactive programs by refinement mappings put forward by Lamport in [25] and further developed by Abadi and Lampert [1], Stark [33,34], Jonsson [21], Lynch and Tuttle [27] and Lam and Shankar [24]. We will show that refinement of reactive systems can be seen as a special case of the general method for data refinement described in part I of this paper.

The action system approach is describe in more detail in Section 2. Action systems will be just a special kind of block statements, consisting of an initialization and a loop. In Section 3 we show how to apply the general refinement theory to the specific case of refining action systems, with and without data refinement. In Section 4 we show how to describe reactive systems in this framework, and introduce operators for reactive composition. The notion of simulation refinement between reactive components is introduced based on data refinement, and we give a general method for refining reactive components of action systems.

In Section 5 we study the notion of simulation refinement more closely and show that it can be used as such for refining reactive systems. Finally, in Section 6 we give an example derivation of a parallel program, showing how to derive parallel algorithms and how to refine reactive components of parallel algorithms.

2 Parallel programs as action systems

We will below show that a subset of the command language of part I of this paper can be used for describing both parallel and reactive systems. As a consequence, the theory of program refinement can be carried over directly to this class of programs.
2.1 Action system

An action system is a statement in the command language of the form

\[ A = [[ \text{var} \ z; S_0; \text{do} \ A_1; \ldots; A_m; \text{end} ]] : z. \]

Here \( z \) are the local variables of \( A \). \( z \) are variables of the system, \( S_0 \) is the initialization statement and \( A_1, \ldots, A_m \) are the actions (or guarded commands) of \( A \). 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 \( A_i \) by \( g(A_i) \) and the statement of it by \( s(A_i) \), so \( A_i = g(A_i) \rightarrow s(A_i) \). We use the shorter forms \( A_i \) and \( s_i \) for those whenever it is unambiguous.

The local and global variables are assumed to be distinct, i.e., \( z \cap z' = \emptyset \) (no redeclaration of variables is thus permitted). Each variable may be associated with an explicit type. The state variables \( y \) consists of the local and global variables, \( y = z \cup \bar{z} \). The set of (free) state variables in action \( A_i \) is denoted \( s(A_i) \) (an action can also have variables declared in local blocks, which are then bound).

An action system provides a global description of the system behavior. The state variables determine the state space of the system. The initialization statement establishes an initial state of the system by assigning suitable values to these variables. The actions determine what can happen in the system during an execution. The execution terminates when no action is enabled anymore.

An action system can be seen as a syntactic description of a transition system. However, we are here proposing action systems as a specification and programming language for parallel systems rather than as a way of modelling the semantics of parallel programs, for which purpose transition systems are usually employed. The action system formalism is quite general. The body of an action may be an arbitrary, possibly nondeterministic statement, it need not terminate and the nondeterminism may be unbounded.

We will in the sequel assume that each action \( A \) in an action system satisfies the following two conditions:

(a) The body of \( A \) is strict, i.e., \( s(A(\text{false})) = \text{false} \).

(b) The body of \( A \) is conjunctive, i.e., \( s(A(A_i, Q_i)) = \bigwedge_i s(A_i, Q_i) \) for any set \( \{ Q_i \} \) of predicates.

This rules out angelic and miraculous statements for action bodies. The first assumption can be done without loss of generality. If \( A \) is not strict, then we can write \( A \) in the equivalent form \( A' = A \land \neg s(A(\text{false})) \) and \( A'' = \neg s(A(\text{false})) \), where \( A'' \) is strict. The second assumption is, however, fundamental and is a real restriction on the language of action systems.

Example

Figure 1 shows an example of a simple sorting program (exchange sort) described as an action system. This program will sort \( n \) integers \( X_1, \ldots, X_n \) in ascending order. The initialization statement assigns the initial values \( X_1, \ldots, X_n \) to the local variables \( x_1, \ldots, x_n \), while the \( n \) sorting actions exchange neighboring values if they are out of order. The program terminates in a state where the array \( x \) is a permutation of the original array \( X \) and \( x_{i+1} \leq x_i \) for \( i = 1, \ldots, n-1 \). 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.
\[ z.1, \ldots, z.n := X.1, \ldots, X.n; \]
\[
\text{do } z.1 > z.2 \rightarrow z.1, z.2 := z.2, z.1
\]
\[ \ldots \]
\[
\text{od}\]
\[
\text{end}\]
\[
\text{Ex}(n-1)\]
\[
\Rightarrow x.1, \ldots, x.n \in \text{integer}\]

**Figure 1: Exchange sorting**

\[
\text{Ex.1 } \quad \text{Ex.2 } \quad \ldots \quad \text{Ex.} \quad \ldots \quad \text{Ex}(n-1)
\]

\[
x_1 \quad x_2 \quad . \quad \ldots \quad x_i \quad x_{i+1} \quad \ldots \quad x_{(n-1)} \quad x_{(n-1)} \quad x_n
\]

**Figure 2: Sorting program actions and variables**

**Notation** The symbol \( \downarrow \) is the same as meet on commands. Hence \( A = A_1 \downarrow \ldots \downarrow A_m = A_1 \wedge \ldots \wedge A_m \) is a command in itself. It can be written as a single action,

\[
A_1 \downarrow \ldots \downarrow A_m = \bigvee_{i=1}^m gA_i \rightarrow i f A_i \downarrow \ldots \downarrow A_m \downarrow i f
\]

We write \( gA = \bigvee_i gA_i \) and \( xA = \quad i f A \downarrow \ldots \downarrow A_m \downarrow i f \). This permits us to consider the whole action system \( A \) as consisting of a single action \( A_i \), i.e., \( A = [\text{var } x; S_0; \text{do } A \text{ od}] : x \).

Note also that \( xA = \bigcup_i xA_i \).

When \( S_0 \) is \( x := x_0 \), where \( x_0 \) is a list of initial values for the variables \( x \), we write \( \text{var } x := x_0 \) for \( \text{var } x; x := x_0 \).

2.2 Parallel execution of action systems

The definition of action systems above presumes a sequential nondeterministic execution model. Action systems may, however, also be executed in parallel. Consider again the action system \( A \).

\[
A = [\text{var } x; S_0; \text{do } A_1 \downarrow \ldots \downarrow A_m \text{ od}] : x
\]

Let

\[
\mathcal{P} = \{ p_1, \ldots, p_r \}
\]

be a partitioning of the state variables \( y = x \cup z \), i.e.,

(i) \( \emptyset \neq p_i \subseteq y \) for each \( i \),

(ii) \( p_i \cap p_j = \emptyset \) whenever \( i \neq j \) and

(iii) \( y = \cup \mathcal{P} \).
We refer to each $p_i$ as a process. Intuitively, we identify a process with the set of state variables located at the process. We will refer to the pair $(A, P)$ as a partitioned action system.

The action $A$ is said to involve process $p$ if $A \cap p \neq \emptyset$, i.e., if $A$ accesses some variable in $p$. Let us denote by $P_A$ the set of processes involved in $A$ for a given partitioning $P$, so $P_A = \{ p \in P : A \cap p \neq \emptyset \}$.

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 $P_{A_i} \cap P_{A_j} = \emptyset$. The actions are competing if they are not independent.

A parallel execution of an action system is any execution where only independent actions are executed in parallel. As independent actions do not have any processes in common, they can also have no state variables in common. Hence, the effect of executing two independent actions in parallel is the same as executing them sequentially, in either order. The enabledness of two independent actions is also not affected by the order in which they are executed. This means that a parallel execution cannot produce any results that cannot be produced by a sequential execution (we will show this formally in Section 4.4). Different partitionings of the state variables will induce different parallel executions for the action system.

Distributed action systems We can view a partitioned action system as a distributed system, with each variable being local to some process. 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 the conventional synchronous message passing models for distributed systems such as CSP[16].

As an example, consider the sorting program where the variables are 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.

Shared variable model We can also view a partitioned action system as a concurrent system with shared variables. Some processes are then considered as (passive) protected regions of shared variables. The variables in such a region may only be accessed under mutual exclusion. The other processes are active, communicating with each other by executing actions that access the shared regions.

Figure 3: Distributed sorting
As an example, we might consider a partitioning where each variable \( x \) forms its own partition. We could take every even numbered variable to be a shared variable and every odd numbered variable to be an active process (see Figure 4, where the active processes are boxed in). Then process \( \{ x \times i \}, i \) an odd number, uses the two actions \( Ex_i \{ i \times (i - 1) \} \) and \( Ex_i \{ i \times (i + 1) \} \) to communicate with its neighboring processes, via the shared variables \( x \{ i \times (i - 1) \} \) and \( x \{ i \times (i + 1) \} \). The variable \( x \times i \) is internal to this active process. As long as the shared variables are accessed under mutual exclusion, any parallel execution will sort the array of \( x \) values.

No explicit synchronization is needed between the actions here, mutual exclusion of the shared variable regions is sufficient.

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 the synchronizing handshake is implemented, while a shared variable implementation requires some locking mechanism. However, from a logical point of view, these models are isomorphic.

We will not go deeper into the subject of how to implement the action systems in a parallel fashion. Distributed implementations of action systems are described by Back and Kurki-Suonio [5] for two-process actions in CSP with output guards. Efficient implementations of so-called decentralized action systems on broadcasting networks are presented in [4,7]. Implementations of action systems on point-to-point networks are described by Eklund [15, 13] and Bagrodia [19]. In a forthcoming paper written jointly with Sere [11] we show how to implement action systems in Occam [20] (which does not permit output guards).

2.3 Fairness in action system executions

Action systems in the form presented above do not have any kind of fairness conditions or liveness assumptions built into them. However, we can encode a fairness constraint into an action system by explicitly scheduling actions with the unbounded nondeterministic assignment statement. This can e.g. be used to enforce action fairness in executions [6,16,7], i.e. that the individual actions in the action system are selected in a fair manner. The method of encoding fairness was proposed by Apl and Alderhag [2].

We will not go very deeply into the issue of how to encode fairness into action system here, as this has been treated elsewhere. However, we show by an example how to ensure fairness for a specific action in an action system. Let \( A \) be the action system

\[
A = \{ \text{var} \; x; \; S_0; \; \text{do} \; B \; | \; A_1 \; | \; \ldots \; | \; A_m \; \text{od} \; \} : x.
\]
We want the execution to treat action \( B \) in a fair manner: if \( B \) is enabled continuously from some point on (or infinitely often), then \( B \) must be executed infinitely often (weak fairness respectively strong fairness). The action system \( A^F \) encodes this fairness constraint, 

\[
A^F = \{ \text{[var } z, \text{eB}; S_1; \text{eB} : \equiv e(z > 0); \text{do } B \rightarrow A^F \uplus \ldots \uplus A^n \text{ od]} ; z \}
\]

Here \( eB \) is an integer variable used to eventually force the execution of action \( B \), and 

\[
B^F = \text{gB} \rightarrow \text{eB}; \text{eB} : \equiv e(z > 0) \\
A^F = \text{gA}, \text{e} \rightarrow (\text{gB} \wedge \text{eB} \leq 0) \rightarrow \text{update } B_1; sA_1.
\]

The definition of \text{update eB} depends on the kind of fairness one requires. For weak fairness we define it by 

\[
\text{update eB} = \text{if } \neg \text{gB} \rightarrow \text{eB} : \equiv e(z < \text{eB}) \uplus \neg \text{gB} \rightarrow \text{eB} : \equiv e(z > 0) \text{ fi}
\]

and for strong fairness by 

\[
\text{update eB} = \text{if } \neg \text{gB} \rightarrow \text{eB} : \equiv e(z < \text{eB}) \uplus \neg \text{gB} \rightarrow \text{skip fi}
\]

The final termination of an action system \( A \) is also expressible within this framework: It is sufficient to prove termination for the corresponding action system where the appropriate fairness requirements have been encoded.

2.4 Stepwise refinement of parallel programs

The action system approach makes stepwise refinement of parallel programs simple and convenient. Parallel programs are just special kinds of sequential statements, so one can apply the refinement calculus as such to the stepwise refinement of parallel programs. The basic framework need not be extended with new program constructs in order to handle parallel execution and communication. Stepwise refinement of both sequential and parallel programs can thus be carried out within a single unifying framework.

The action system approach permits the design of the logical behavior of a system to be separated from the issue of how the system is to be implemented. The latter is seen as a design decision that does affect the way in which the action system is built, but is not reflected in the logical behavior of the system. The decision whether the action system is to be executed in a sequential, shared variable or distributed fashion can be postponed to a later stage, when the logical behavior of the action system has been designed. Also the specific way in which the system is partitioned into processes can be determined at a later stage, and possibly only for the last program version. We can experiment with different ways of partitioning variables into processes to see which gives the best performance or can be implemented in the most efficient way on the available hardware.

Stepwise refinement would typically start with a specification of the intended behavior of the system, possibly given in the form of a sequential statement. The goal is to construct a partitioned action system that satisfies some efficiency or implementation criteria. Often one has a specific target architecture in mind, so that the partitioning of some of the variables of actions is already determined (the partitioning of the global variables into processes could e.g. already be determined by the problem formulation). The required parallel action system is constructed by small refinement steps leading up to an action system satisfying the stated requirements.

A method for stepwise refinement of action systems in a temporal logic framework, based on the idea of supersposition, was first put forward in Back and Kurki-Suonio [5] (now also
published in [9]). Refining action systems in the refinement calculus was first described
Back and Sere in [10, 5, 12, 32]. The emphasis in these works is on the refinement of parallel
programs preserving their input-output behavior.
We will in the sequel consider in more detail the specific properties that action systems
have, in order to build a theory of stepwise refinement of parallel programs. We will specifi-
cally concentrate on the refinement of reactive action systems, referring to the above sources
for the work on refining parallel algorithms preserving input-output behavior.

3 Refinement of action systems
We will in this section apply the general theory of program refinement that was presented in
part I to action system refinement. We will first consider refinement of actions and iteration
statements as such. We then generalize this to data refinement of actions and iteration
statements. Finally, we give general rules for proving refinement between action systems.

3.1 Iteration statements
Actions Let us first consider under what conditions an action is refined by another action. We
have the following result.

**Lemma 1** Let A and A' be two actions. Then A ≤ A' if and only if
(i) sA ≤ sA'
(ii) ¬sA' ≤ ¬sA.
Note that we use ≤ for the implication ordering, so condition (ii) is equivalent to ¬sA ⇒ ¬sA'. An equivalent formulation for these conditions is
(i) {sA'}; A ≤ sA' and
(ii) sA' ≤ sA.
In other words, action A is refined by action A' if and only if the following two conditions
hold: (i) whenever A' is enabled, the body of A is enabled by the body of A' and (ii) A is
enabled whenever A' is enabled.

**Iteration statements** Let us next consider refinement between iteration statements. Let

$$DO = \text{do } A_1 \ldots \text{ do } A_n \text{ od } y \text{ and}$$

$$DO' = \text{do } A'_1 \ldots \text{ do } A'_n \text{ od } y.$$

Let A = A_1 \ldots A_n and A' = A'_1 \ldots A'_n. We have the following result.

**Lemma 2** DO ≤ DO' if
(i) A ≤ A'
(ii) sA ≤ sA'
An equivalent formulation is

**Lemma 3** DO ≤ DO' if
(i) sA ≤ A'_i, for i = 1, \ldots, k, and

9
Thus the exit conditions must be exactly the same in the refining and the refined loop.

Condition (i) can be difficult to prove directly in terms of the definition of refinement. The characterization theorem (of part I) permits us, however, to express it as a total correctness assertion.

**Lemma 4** $DO \leq DO'$ if the following two conditions hold:

(i) Let $\delta A_i^+$ be some generalized inverse of $\delta A_i$, for $i = 1, \ldots, m$. Then

$$\delta A(\text{true}) \land \delta A' \land y = y_0 \exists x \in \{\delta A_i^+ \mid \delta A \land y = y_0\}$$

must hold for $j = 1, \ldots, k$.

(ii) $\delta A \equiv \delta A'$

We have here used the fact that $\bigvee \delta A_i^+$ is a generalized inverse of $\delta A$. The generalized inversion can be rather simply calculated for action bodies without loops or recursion, so this is quite a useful method for establishing refinement between loops.

### 3.2 Data refinement

When making a refinement of an action system we usually also need to change local variables. This means that we have to make a data refinement. We show here how to apply the method for data refinement described in part I to data refinement of actions and loops.

**Actions** Let us first consider data refinement of actions. We have the following result.

**Lemma 5** Let $A : y$ and $A' : y'$ be actions and let $E : y \rightarrow y'$ be an encoding statement. Then $A \leq E A'$ if and only if

(i) $\delta A \leq E A'$

(ii) $\neg \delta A \leq E (\neg \delta A')$

Let us consider the special case when the encoding is given in terms of an abstraction relation $R(u, u', v)$, where $y = u, v$ and $y' = u', v$. In this case we have

$$E = (\lambda + u' - u. R(u, u', v)).$$

The characterization theorem gives us the following result.

**Lemma 6** Let $A : y$ and $A' : y'$ be actions and let $E = (\lambda + u' - u. R(u, y', v))$ be an encoding. Then $A \leq E A'$ if and only if

(i) $\delta A(\text{true}) \land R \land \delta A' \land y = y_0 \exists u. (R \land \delta A''(\delta A \land y = y_0))$

(ii) $R \land \delta A' \Rightarrow \delta A$

The conjunct $\delta A(\text{true})$ is not needed in the assumption if $A$ is total, i.e., if $A(\text{true})$ holds (which is equivalent to $\delta A \Rightarrow \delta A(\text{true})$).

10
Iteration statements. We now generalize the result for refinement of iteration statements to data refinement of these. Let $E : y \to y'$ be an encoding and let

$$
DO = \text{do } A_1 \mid \ldots \mid A_m \text{ od } : y \text{ and }
$$

$$
DO' = \text{do } A'_1 \mid \ldots \mid A'_m \text{ od } : y'.
$$

As before, let $A = A_1 \mid \ldots \mid A_m$ and $A' = A'_1 \mid \ldots \mid A'_m$. We have the following result.

**Lemma 7.** $DO \leq_E DO'$ if

(i) $A \leq A'$ and

(ii) $pA \leq E(sA')$

An equivalent formulation is

**Lemma 8.** $DO \leq_E DO'$ if

(i) $sA \leq A'_i$ for $i = 1, \ldots, k$ and

(ii) $pA \leq E(sA')$ and $pA \subseteq E(sA')$

Using the characterization theorem and assuming that the encoding is given by a relation, we then get the following proof rule for data refinement of iteration statements.

**Lemma 9.** Let $E = \{ (x + y') = x \cdot R(n, x', y) \}$, where $y = x, y$ and $y' = y, x$. Then $DO \leq_E DO'$ if the following two conditions hold:

(i) Let $sA_i$ be some generalized inverse of $sA_i$ for $i = 1, \ldots, m$. Then

$$
\exists A \text{ such that } (R \land sA_i) \land y = y_i \text{ for } i = 1, \ldots, m.
$$

(ii) $R = (sA_i \equiv gA')$

3.3 Action systems

We now put all this together. Let us consider the action systems

$$
A = \parallel \text{ var } z ; S_0 \text{ do } A_1 \mid \ldots \mid A_m \text{ od } : z
$$

$$
A' = \parallel \text{ var } z' ; S'_0 \text{ do } A'_1 \mid \ldots \mid A'_m \text{ od } : z.
$$

We want to prove that $A \leq A'$. It is sufficient to show that there is an encoding $E : x \to x'$ such that the initialization and iteration of the first action system is data refined by respectively the initialization and the iteration in the second action system. Hence, we have the following result.

**Theorem 1.** $A \leq A'$ if there exists an encoding $E : x \to x'$ such that

(i) $(\langle x + z \rangle ; S_0) \leq (\langle x + z' \rangle ; S'_0 ; E^{-1})$;

(ii) $A \leq E A'$ and

(iii) $pA \leq E(gA')$.
Using the characterization theorem and assuming that the encoding is given by an abstraction relation, this gives us the following proof rule.

**Corollary 1** Let \( x = u, v \) and \( x' = u', v' \). Assume that the initialization statements are of the form \( S_0 = (u, v = u_0, v_0) \) and \( S_0 = (u', v = u'_0, v_0) \). Then \( A \leq A' \) if there exists a relation \( R(u, u', v) \) such that the following conditions hold:

(i) \( R(u_0, u'_0, v_0) \).

(ii) Let \( sA_i \) be some generalized inverse of \( sA_i \) for \( i = 1, \ldots, m \). Then

\[
sA(\text{true}) \land R \land \forall x \forall A \forall u, v, z = u_0, v_0, z_0 \exists \forall (R \land \bigwedge_{i=1}^m sA_i'(gA_i \land u, v, z = u_0, v_0, z_0))
\]

must hold for \( j = 1, \ldots, k \).

(iii) \( R \Rightarrow (gA_i \equiv gA'_i) \).

### 3.4 Refinement with fairness constraints

As indicated in the previous section, the requirement that some actions must be treated fairly in every execution can be coded into an action system. Thus, given a specification statement \( S \), we can give a fair action system \( A' \) as an implementation of it. The refinement \( S \leq A' \) is established in the usual way. A later refinement \( A' \) might remove these fairness constraints, by introducing explicit scheduling among the actions, without using the unbounded nondeterministic assignment statement. The additional variables introduced to enforce fairness can be removed by data refinement. Alternatively, the action system may be refined into a form where the fairness requirements are guaranteed by the intended parallel implementation of the system. See [7] for a detailed discussion of these issues.

Proving properties of an action system \( A \) without fairness assumptions is conservative, in the sense that any total correctness property that holds for this action system must also hold for any action system derived from this by imposing fairness constraints. Fairness constraints can only narrow down the set of possible final states, but cannot introduce any new finite or infinite computations.

On the other hand, there might be properties that hold for every fair execution of an action system but which do not hold for unfair executions. To prove these will require that the fairness constraints are explicitly coded into the action system.

### 4 Reactive programs as action systems

A refinement \( A \leq A' \) will guarantee that the total correctness of action system \( A \) is preserved by \( A' \). This means that the latter will satisfy any total correctness specification that the former satisfies. However, it will not guarantee that the behavior of \( A' \) during execution will be the same as the behavior of \( A \). Hence, the input-output correctness of parallel programs is preserved, but not necessary their reactive behavior, i.e. the way in which they react with their environment during the execution. We will here show how to extend the refinement calculus so that also this aspect of program correctness can be preserved by refinement steps.

For simplicity, we assume in the sequel that all initializations are just assignments of initial values.
4.1 Parallel composition and hiding

Parallel composition. Given two action systems \( A \) and \( B \),
\[
A = \{ \text{var } x := z; \text{do } A_1 \{ \ldots \{ A_m \text{ od } \}\} ; z \}
\]
\[
B = \{ \text{var } y := y; \text{do } B_1 \{ \ldots \{ B_k \text{ od } \}\} ; u \}
\]
we define their parallel composition \( A \parallel B : z \cup u \) to be
\[
A \parallel B = \{ \text{var } x, y := z, y; \text{do } A_1 \{ \ldots \{ A_m \{ B_1 \{ \ldots \{ B_k \text{ od } \}\} \text{ od } \}\} ; z \cup u \}
\]
This is the same as the union operator in UNITY [14], except that we also keep track of which variables are local and which are global (UNITY only has global variables. For simplicity, we assume that \( x \cap y = \emptyset \). If this is not the case, we first need to rename local variables in the action systems so that the condition becomes fulfilled, before the parallel composition is formed.

Note that while the local variables are distinct, the global variables are shared, i.e., a variable that occurs in both \( z \) and \( u \) is considered to be the same. This explains the use of union to determine the global variables of the parallel composition.

Renaming. We can always use substitution to rename the global variables of action systems. Then if \( A \) is an action system on the global variables \( z \), then \( A[z/x] \) is an action system on the global variables \( z \) (a list of distinct variables), which we get by replacing each occurrence of a variable \( x \) by the corresponding variable in \( z \), changing local (bound) variables if necessary to avoid capture of global variables.

Hiding. Given an action system \( A : z \) of the form above, where \( z = z_1, z_2 \), we can hide some of its variables by making them local. This is achieved by the block construct,
\[
A' = \{ \text{var } z_1 := z_1; 0; \text{do } A \} ; z_2
\]
Hiding the variables \( z_1 \) thus makes them inaccessible from other actions outside \( A' \) in a parallel composition.

Given an action system
\[
C = \{ \text{var } v := v; \text{do } C_1 \{ \ldots \{ C_m \text{ od } \}\} ; z \}
\]
we can decompose it into smaller action systems by parallel composition and hiding. Let \( A = \{ A_1, \ldots, A_n \} \) and \( B = \{ B_1, \ldots, B_k \} \) be a partitioning of the actions in \( C \) (we deliberately use the symbol \( A \) for both the set of actions \( \{ A_1, \ldots, A_n \} \) and the action \( A_1 \{ \ldots \{ A_m \text{ od } \}\} \); which one is intended should be clear from the context). Let
\[
x = \tau A - \tau B - z
\]
\[
y = \tau B - \tau y - z
\]
\[
w = \tau A \cap \tau B - z
\]
We can then write \( C \) as
\[
C = \{ \text{var } w := w; 0; A \parallel B \} ; z
\]
13
\textbf{where}

\begin{align*}
A & = [[ \text{var } z := x_0; \text{do } A \text{ od }] ] : z, w \\
B & = [[ \text{var } y := y_0; \text{do } B \text{ od }] ] : z, v
\end{align*}

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 (\cite{20}) 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.

\subsection*{4.2 Refinement of reactive components}

Consider the action system

\[ C = [[ \text{var } e := e_0; \text{do } C_1 \mid \ldots \mid C_n \text{ od }] ] : z. \]

Assume that we want to refine \( C \) to another action system \( C' \) by replacing some actions \( A_1, \ldots, A_m \) in \( C \) by some other actions \( A_1', \ldots, A_m' \), possibly also changing some local variables.

Our first step is then to decompose \( C \) into

\[ C = [[ \text{var } w := w_0; A \parallel B ]], \]

as described above. Here \( A \) collects the actions \( A_1, \ldots, A_m \) to be replaced and hides the variables that are only accessed by these actions. The component \( A \) is again an action system. It is reactive, as the total system behavior (and final result) is determined by how \( A \) reacts with the other component \( B \) during execution.

We want to find conditions under which the replacement of the old actions by new actions in \( C \) is permitted also in a reactive context. In other words, we want to find conditions under which

\[ C = [[ \text{var } w := w_0; A \parallel B ] ] \leq \llbracket \text{var } w := w_0; A' \parallel B \rrbracket ] = C', \]

for an action system \( A' \).

It turns out that the method of data refinement is in fact sufficient for this purpose. We will say that action system \( A : z \) is \textit{strongly simulation refined} by action system \( A' : z \), denoted \( A \preceq A' \), if the conditions of Theorem 1 are satisfied, \textit{i.e.}, if there exists an encoding statement \( E : z \rightarrow z' \) from the local variables \( z \) of \( A \) to the local variables \( z' \) of \( A' \) such that the following three conditions are satisfied:

\begin{enumerate}
  \item \( (\lambda + z); S_0 \leq (\lambda + z'); S_0; E^{-1} \)
  \item \( A \preceq A' \) and
  \item \( qA \preceq E(qA') \)
\end{enumerate}

\textbf{Theorem 2 (Refinement of reactive components)} Let \( A \) and \( A' \) be two action systems. Then \( A \preceq A' \) implies that

\[ [[ \text{var } w := w_0; A \parallel B ] ] \leq \llbracket \text{var } w := w_0; A' \parallel B \rrbracket \]

for any choice of \( w, w_0 \) and \( B \).

This result follows directly from the fact that the data refinement \( \preceq \) achieves by an encoding statement on the local variables of \( A \) and \( A' \) only. Hence, for each action \( B_i \in B \) we have \( B_i \preceq B_i \), \textit{i.e.}, \( B_i \) is invariant under the encoding \( E \), as \( E \) and \( B_i \) do not have any variables in common.
4.3 Permitting stuttering

The above result shows that data refinement is sufficient to permit replacement of a reactive component of an action system by another, so that the total correctness of the whole action system is preserved. However, this relation is more restrictive than what is often needed. The problem is that it requires a one-to-one correspondence between the actions executed by A and by A'. Even though an action execution in A' need not always correspond to an execution of the same action in A, it must always correspond to some action execution of A. In practice, however, executing a simple action in A will often correspond to executing a sequence of two or more actions in A', so that the one to one correspondence is not maintained.

Following Lamport [26,4], this problem can be overcome by permitting stuttering actions in A', i.e., actions which do not correspond to any state change in A. The way we will handle this problem is actually quite simple. We note that for any execution of the action system A,

\[ A = \llbracket \text{var } x :: x0 ; \text{do } A \text{ od} \rrbracket ; z, \]

the meaning of A is unchanged (i.e., the weakest precondition transformer is the same) if we permit a finite number of skip actions (stutterings) to be inserted into the execution. Moreover, the behavior of A is in any reactive context is also unchanged if we add stutterings. The only restriction is that we may not add an infinite sequence of successive stutterings.

The action system A* is derived from A by permitting any finite amount of consequent stuttering (but it does not permit infinite stuttering):

\[ A^* = \llbracket \text{var } x, h :: x0, ? \text{ od } A \text{ od } h :: ? \text{ od } h \cdot h' (A' < h) \text{ od} \rrbracket ; z. \]

We assume that h ranges over a well-founded set with least element 0. The statement \( \text{var } x, h :: x0, ? \) is an abbreviation for the statement \( \text{var } x, h :: x0, h :: ? \) when \( h = ? \) in turn is an abbreviation for the statement \( h :: 0, \text{true} \).

**Lemma 10** Let A and A* be defined as above. Then

\[ \llbracket \text{var } w :: w0; A ; B \rrbracket = \llbracket \text{var } w :: w0; A^* ; B \rrbracket \]

for any choice of w, w0 and B.

We will say that the action system A is (weakly) simulation refined by action system A', denoted \( A \leq A' \), if \( A^* \leq A' \).

Simulation refinement is stronger than ordinary (total correctness) refinement, i.e., we always have that \( A \leq A \Rightarrow A \leq A' \).

The following result now follows directly from Theorem 2, in combination with the lemma above (we also need the transitivity of simulation relation proved in Theorem 6).

**Theorem 3** (Refinement of Reactive Components with Stuttering) Let A and A' be two action systems. Then \( A \leq A' \) implies that

\[ \llbracket \text{var } w :: w0; A ; B \rrbracket \leq \llbracket \text{var } w :: w0; A' ; B \rrbracket, \]

for any choice of w, w0 and B.
Proof rule for refinement with stuttering Let us check more carefully what the conditions for refinement are in the presence of stuttering. Let

\[ A^* = (\sigma A \rightarrow s A; \mathcal{h} = \mathcal{h}' (k' < k)). \]

Let the local variables of \( A \) be \( x = u, v \) and the local variables of \( A' \) be \( x' = u', v' \). We assume that the encoding is given in terms of an abstraction relation \( R(u, k, u', v) \), which now also has \( \mathcal{h} \) as an (abstract variable) argument. We then have that \( A \leq A' \) if and only if

1. \( \exists \mathcal{h}, R(u, k, u', v) \)
2. \( s A^+ \leq s A'_j \) for each \( j = 1, \ldots, k \), and
3. \( R(u, k, u', v) \Rightarrow (s A'^+ = (s A \lor \mathcal{h} > 0)). \)

In condition (ii), we have

\[ s A^+ = \text{if } \sigma A \rightarrow s A; \mathcal{h} := ? \mid \mathcal{h} > 0 \rightarrow \mathcal{h} := \mathcal{h}' (k' < k) \]

Computing the second condition using the characterization theorem then gives us the following proof rule.

Theorem 4 Let \( A \) and \( A' \) be two action systems with local variables \( x = u, v \) and \( x' = u', v \) respectively. Then \( A \leq A' \) if there exists a relation \( R(u, k, u', v) \) such that the following conditions hold:

1. \( \exists \mathcal{h}, R(u, k, u', v) \)
2. Let \( s A^+_j \) be some generalized inverse of \( s A_i \), for \( i = 1, \ldots, m \). Then
   \[ x, k, z = z_0, k_0, z_0 \land (\sigma A \Rightarrow s A_1 (\text{true}) \land R(u, k, u', v) \land s A'_j) \]
   must hold for every \( j = 1, \ldots, k \),
3. \( R(u, k, u', v) \Rightarrow (s A'^+ = (s A \lor \mathcal{h} > 0)). \)

This provides us with a general method for proving the correctness of refinement in reactive contexts.

The way that we have presented the proof rule here only shows that \( C(A) \leq C(A') \) holds if \( A \leq A' \) holds, i.e., we get a total correctness refinement of the whole program when making a refinement of its component. We show in Section 5.2 that in fact \( C(A) \not\leq C(A') \) will also hold, so that our approach gives a general method for refinement of reactive programs.

4.4 Example: Correctness of parallel execution model

We will exemplify simulation refinement of action systems by showing that parallel execution of action systems is correct for any partitioning of the state variables.

Let \( A \) be the action system

\[ A = [\text{var } x; S_0, \text{do } A_1; \ldots; \text{do } A_n \text{ od }] : x \]

16
and let $P = \{ p_1, \ldots, p_k \}$ be a partitioning of the state variables $y = z \cup \pi$. Let $A^P$ be the action system

$$A^P = [\forall \text{ var } x, \text{ res } ; S_0 ; \text{ res } := \emptyset ; \text{ do } A^P \ldots \mid A^P \mid \text{ res } \mid \pi \mid p \text{ od }] : z.$$ 

Here $\text{ res }$ is a variable that records the processes that are presently involved in the execution of some action, and

$$A^P = \begin{align*}
p A_{\pi} & = g A_{\pi} \land (p A_{\pi} \cap \text{ res } = \emptyset) \rightarrow \text{ res } := \text{ res } \cup p A_{\pi}; s A_{\pi} \\
p E & = p \in \text{ res } \rightarrow \text{ res } := \{ \text{ res } \}\end{align*}$$

The action system $A^P$ models the parallel execution of actions in $A$ for the partitioning $P$. It guarantees that two actions that involve the same process cannot be executing at the same time, one must release the process before the other can begin. The handshake actions $A^P$ synchronize the processes for execution of an action, while the release actions $E$ release the processes one-by-one from the actions (there is no end-synchronization). This model for parallel execution is studied in a temporal logic framework in Back and Kurki-Stenio [6,7].

The following result now states that executing an action system in parallel is a correct refinement of a sequential execution of the same action system, for any partitioning of the state variables.

**Theorem 5.** $A^P \subseteq A^+$ for any partitioning $P$ of $A$.

**Proof.** We need to show that $A^+$ is data refined by $A^P$ for a suitable relation $R$. We note that data refinement here only consists in adding the single variable res. We choose

$$R(k, \text{ res }) \iff (k = \# \text{ res }),$$

where $\# \text{ res }$ is the size of the set res.

We need to prove that conditions (i) – (iii) of Theorem 1 hold. The first condition requires that $\mathbb{E}(k = \emptyset)$. This is evidently true.

We have to show condition (ii) for each $A^P_\pi$ and for each $E_p$. In the first case, the condition follows from the fact that

$$x, z, x = x_0, x_0 \land g A_{\pi}$$

$$\exists \pi (k = \# \text{ res } \land z A_{\pi} \land (x_0, z, x = x_0, z_0))$$

for every $i = 1, \ldots, m$, where $z A_{\pi}$ is some generalized inverse of $s A_{\pi}$. In the second case, the condition follows from the fact that

$$x, z, x = x_0, x_0 \land (k = \# \text{ res } \land p \in \text{ res }$$

$$\text{ res } := \text{ res } \setminus p$$

$$0 \leq \# \text{ res } < \# 0 \land z, x = x_0, z_0$$

for every $j = 1, \ldots, k$.

The last condition requires us to show that

$$\forall_{i=1}^m (g A_{\pi} \land (p A_{\pi} \cap \text{ res } = \emptyset)) \lor \forall_{j=1}^k y_j \in \text{ res } \Rightarrow 0 < \# \lor \forall_{i=1}^m g A_{\pi},$$

This is easily proved by case analysis. □

17
4.5 Refinement in context

Consider again the refinement

\[ C = \semantics{\text{var } v := w; A \parallel B} \leq \semantics{\text{var } v := w; A' \parallel B} = C'. \]  

(1)

This is established if we can prove that \( A \preceq A' \). This in turn implies that \( A \) can be replaced by \( A' \) in any context, which is a very strong requirement. The fact that we are only interested in replacing \( A \) with \( A' \) in the specific context of \( B \) is not taken into account. Hence it is quite possible that \( 1 \) holds even if \( A \preceq A' \) does not hold. Thus, we need a method for taking the context into account when establishing program refinements.

The problem here is that in making the refinement \( A : w \leq A' : w \) we may not make any assumptions about how the values of \( v \) and \( v' \) may be changed by the environment. The abstraction relation may only relate the local states of \( A \) and \( A' \) to each other, i.e., it must be of the form \( R(x, x') \).

The simple solution to this is to make a data refinement of the whole \( C \), i.e., to establish directly \( C \leq C' \), which then implies \( C \subseteq C' \). This permits us to include the variables \( w \) also in the abstraction relation. This approach is, however, in most cases too cumbersome: it does not take into account that only a part of the actions in \( C \) actually influence \( A \). However, we can base a more refined method for replacement in context on this basic idea.

The idea is to add to \( A \) those actions from the environment \( B' \) that are important to determine the context of \( A \), but no others. As the same time, we add the environment variables through which the behavior of \( A \) is influenced. We do this by partitioning the actions in \( B \) into two sets, the \textit{interface actions} \( B^* \) and the \textit{environment actions} \( B'' \). The interface actions capture those aspects of the environment that are of importance for the refinement of the actions that we are interested in. We then decompose \( C \) as

\[ C = \semantics{\text{var } v := s; C_1 \parallel C_2} \]

where

\[ C_1 = \semantics{\text{var } u := w; \text{do } A \parallel B' \text{ od}} \]
\[ C_2 = \semantics{\text{var } y := y; \text{do } B'' \text{ od}}. \]

Here \( s \) are the variables shared by \( A \) and \( B' \) but not by others, \( y \) are the variables of \( B'' \) that are not accessed by other actions and \( v \) are the variables that are shared by \( C_1 \) and \( C_2 \). Notice that \( x \leq s \), but \( x \) may also contain variables from \( w \) or variables local to \( B' \). Hence, more variables may be local in \( C_1 \) than in \( A \).

We then establish the refinement

\[ C_1 \leq C'_1. \]

This is less restrictive than proving \( A \preceq A' \), but more restrictive than proving \( C \leq C' \). This refinement gives us

\[ C = \semantics{\text{var } w := w; A \parallel B} \]
\[ \leq \semantics{\text{var } v := v; C_1 \parallel C_2} \]
\[ \leq \semantics{\text{var } w := w; C_1 \parallel C_2} \]
\[ = \semantics{\text{var } w' := w; A' \parallel B' \parallel B''} \]
\[ = C'. \]

Here

\[ B' = \semantics{\text{var } y' := y'; \text{do } B' \parallel B'' \text{ od}}. \]

18
Thus the interface actions and interface variables may also change in the refinement, to adjust for changes in the target action system \( A \).

This method essentially constructs an envelope or interface \( C_1 \) for \( A \), and refines \( A \) together with this envelope. For an arbitrary action system, one may need to first transform it into a form where the target actions (the actions to be replaced), the environment action and the interface actions are clearly separated before doing the actual refinement in context.

5 Refinement of reactive systems

We have shown above that simulation refinement \( \preceq \) can be used as such for refinement of reactive systems. We will here take a closer look at the properties of this relation.

5.1 Simulation refinement

The simulation refinement relation has the basic properties required of a refinement relation, i.e., reflexivity, transitivity and monotonicity.

**Theorem 6** Let \( A, A', A'', B, B' \) be action systems, \( w \) and \( w' \) lists of distinct variables and \( w0 \) a list of initial values. Then the following properties hold:

(i) \( A \preceq A \).

(ii) \( A \preceq A' \preceq A'' \Rightarrow A \preceq A'' \).

(iii) If \( A \preceq A' \) and \( B \preceq B' \) then

   (a) \( A \| B \preceq A' \| B' \),

   (b) \( A[w/f] \preceq A'[w'/f] \) and

   (c) \( \| \text{var } v \leftarrow w0; A \| \preceq \| \text{var } v \leftarrow w0; A' \| \).

This means that we can do stepwise refinement directly in terms of simulation refinement. Starting from some action system \( A_0 \) that serves as the initial specification, we can construct a sequence of action system refinements

\[ A_0 \preceq A_1 \preceq \cdots \preceq A_n, \]

until we reach a reactive system \( A_n \) that is considered adequate. The correctness of each step can be established in the refinement calculus, and thus ultimately in the weakest precondition calculus. Thus, the theory of total correctness, originally only intended for establishing input-output correctness, turns out to be adequate also for the refinement of reactive systems.

Simulation refinement is monotonic with respect to parallel composition, renaming and hiding. This implies that we may replace any reactive component \( C[A] \) of a reactive system \( C[A] \) with its simulation refinement \( C[A] \). In other words, we always have that

\[ A \preceq A' \Rightarrow C[A] \preceq C[A']. \]

A reactive component is here a component built out of action systems using parallel composition, renaming and hiding. For non-reactive components, we can use the ordinary refinement relation and data refinement of statements. In this way the refinement calculus provides a uniform framework in which to establish correctness of refinement for both reactive and non-reactive (input-output) systems.
5.2 Preserving temporal properties

Let us also look a little closer on the semantic interpretation of simulation refinement for action systems. Let $A : z$ and $A' : z$ be two action systems. We write $A(\sigma)$ for the set of all possible complete execution sequences of $A$ that start from initial state $\sigma$. A complete execution sequence is here defined as an execution sequence of $A$ that does not lead to abstraction. A complete execution sequence may thus be either finite or infinite. We include $\bot$ in $A(\sigma)$ if there is an incomplete execution sequence from $\sigma$.

Let us further denote by $A_!(\sigma)$ the set of execution sequences that result from $A(\sigma)$ by only showing the values of the program variables $z$ (the $z$-projection of $A(\sigma)$) and removing any finite stuttering from the resulting execution sequences (we call these the $z$-visible execution sequences). We have the following result.

**Lemma 11** If $A \preceq A'$, then for each $\sigma$,

either $A_!(\sigma) \subseteq A_!(\sigma)'$ or $\bot \in A(\sigma)$.

In other words, if $A(\sigma)$ cannot lead to abortion, then any $z$-visible execution sequence of $A(\sigma)$ is a possible $z$-visible execution sequence of $A(\sigma)$ (and hence also complete).

Let $\phi : z$ be a temporal logic formula (on the variables $z$). Let us write $s \models \phi$ if $\phi$ holds for execution sequence $s$ and $A(\sigma) \models \phi$ if $s \models \phi$ for every execution sequence $s$ in $A(\sigma)$. We postulate that $\bot \not\models \phi$, for any $\phi$. The formula $\phi$ is said to be insensitive to stuttering if $s \models \phi \implies s' \models \phi$ holds whenever $s$ and $s'$ are equivalent up to stuttering.

The previous lemma now implies that simulation refinement preserves all (linear) temporal logic properties, in the following sense.

**Lemma 12** If $A : z \preceq A' : z$, then

$$A(\sigma) \models \phi \iff A'(\sigma) \models \phi$$

for any $\phi : z$ that is insensitive to stuttering and for any initial state $\sigma$.

Combining this with the monotonicity property of simulation refinement then gives us the following general result.

**Theorem 7** Let $C[A] : v$ be any action system where $A : z$ is a reactive component. If $A \preceq A'$, then

$$C[A](\sigma) \models \phi \iff C[A'](\sigma) \models \phi$$

for any $\phi : v$ that is insensitive to stuttering and for any initial state $\sigma$.

Thus, we may replace $A$ by $A'$ in any context $C[v]$ without losing any temporal properties.

6 Example derivation

We will exemplify the method for stepwise refinement of action systems by the following simple problem. Let $X_i, i = 1, \ldots, n$, be a sequence of (real) numbers, $n \geq 0$. We want to compute $\sum_{i=1}^{n} X_i^2$ in a parallel fashion.

20
6.1 Refinement by transformations

We will first derive a solution in which the squaring of the vector elements is done in parallel with summing of the squares. The two tasks will be performed in a pipelined fashion, using an intermediate buffer for communication. The refinement steps can be justified using the transformation rules in [12]. We omit these justifications for lack of space. In the second subsection we then show how to implement the buffer in a way which permits parallel insertion and removal from the buffer.

The global variables are assumed to be $X \in \text{sequence of real}$, $n \in \text{integer}$ and $s \in \text{real}$. The initial specification is as follows

$$S_0: s := \sum_{i=1}^{n} X_i^2$$

We split the computation into two parts, squaring and summing.

$$S_1: \begin{cases} \text{var } M \in \text{sequence of real}; \\ M := (X_i^2; i = 1, \ldots, n); \\ s := \sum M; \end{cases}$$

We then implement squaring and summing by loops. We have indicated concatenation of sequences by $H \cdot H$, and $(y) \cdot M := M$ stands for taking the head and the tail of $M$.

$$S_2: \begin{cases} \text{var } M \in \text{sequence of real}; i \in \text{integer}; y \in \text{real}; \\ M := (); i := 1; s := 0; \\ \text{do } i \leq n \rightarrow M := M \cdot (X_i^2); i := i + 1 \text{ od; } \\ \text{do } M \neq () \rightarrow (y) \cdot M := M; s := s + y \text{ od } \end{cases}$$

The two loops are then merged. The justification for this refinement step is the rule for loop merge described in [12]. Basically, the two loops can be merged because the second loop does not interfere with the computation of the first loop, even if their execution is interleaved.

$$S_3: \begin{cases} \text{var } M \in \text{sequence of real}; i \in \text{integer}; y \in \text{real}; \\ M := (); i := 1; s := 0; \\ \text{do } i \leq n \rightarrow M := M \cdot (X_i^2); i := i + 1 \\
\text{(square)} \\
\text{do } M \neq () \rightarrow (y) \cdot M := M; s := s + y \\
\text{(sum)} \\
\text{od } \end{cases}$$

The squaring and summing actions can now be done interleaved, but they cannot be done in parallel, as they access a common variable ($M$). We therefore split the squaring action into a proper squaring action and an action for inserting the next square into the buffer $M$. Similarly we split the summation action into an action for removing the next square from the buffer and a proper summation action. The required scheduling is achieved by the boolean variables $pred$ and $con$. 

21
Figure 5: Sum of squares, shared queue

S4: $\| \text{var } M \in \text{sequence of real}; y, x \in \text{real}; i \in \text{integer};$
$\text{prod, cons } \in \text{booleans};$
$M := () ; i := 1 ; s := 0 ; \text{prod} := \text{false} ; \text{cons} := \text{true} ;$
do $i \leq n \land \neg \text{prod} \rightarrow x := x_i^2 ; s := s + i ; \text{prod} := \text{true}$
\hspace{1cm} (square)
$\| M \neq () \land \text{cons} := \text{'x'} ; M := M ; \text{cons} := \text{false}$
\hspace{1cm} (remove)
$\| \neg \text{cons} \rightarrow s := s + i ; \text{cons} := \text{true}$
\hspace{1cm} (sum)
od

The access relation for S4 is shown in Figure 5. We see that the summing and the squaring action can now proceed in parallel, as they do not have any common variables. The insertion and removal from the buffer do, however, still exclude each other, as both access $M$.

Finally, we will change the unbounded buffer into a bounded one, by only allowing insertions when the length of $M$ is less than 3. This is permitted by the proof rule for iteration, because the exit conditions for the loops in S4 and in S5 are equivalent.

S5: $\| \text{var } M \in \text{sequence of real}; y, x \in \text{real}; i \in \text{integer};$
$\text{prod, cons } \in \text{booleans};$
$M := () ; i := 1 ; s := 0 ; \text{prod} := \text{false} ; \text{cons} := \text{true} ;$
do $i \leq n \land \neg \text{prod} \rightarrow x := x_i^2 ; i := i + 1 ; \text{prod} := \text{true}$
\hspace{1cm} (square)
$\| M \land |M| < 3 \rightarrow M := M \cdot \{x\} ; \text{prod} := \text{false}$
\hspace{1cm} (insert)
$\| \neg \text{cons} \rightarrow s := s + i ; \text{cons} := \text{true}$
\hspace{1cm} (sum)
od

6.2 Refining a reactive component

We will exemplify the method for refinement of reactive systems by implementing the buffer in such a way that elements can be inserted into it and removed from it in a parallel fashion. The example was inspired by a similar example treated in UNITY [30].

22
For this purpose, we will decompose the action system into three parallel components, the PRODUCER, the BUFFER and the CONSUMER. This gives us the following system.

S5: \[ \begin{array}{l}
\text{let} \; \prod, \, \text{prod}, \, \text{con} \in \text{boolean}; \\
\text{prod} := \text{false}; \, \text{con} := \text{true}; \\
\text{PRODUCER} \parallel \text{BUFFER} \parallel \text{CONSUMER}
\end{array} \]

Here we have

PRODUCER:
\[ \begin{array}{l}
\text{let} \; i \in \text{integer}; \\
i := 1; \\
do \; i \leq n \land \neg \text{prod} \rightarrow x := X_i^2; \; i := i + 1; \; \text{prod} := \text{true} \text{ (square')}
\end{array} \]

BUFFER:
\[ \begin{array}{l}
\text{let} \; M \in \text{sequence of real}; \\
M := \emptyset; \\
do \; \text{prod} \land |M| < 3 \rightarrow M := M \cup \{y\}; \; \text{prod} := \text{false} \text{ (insert')}
| \; M \neq \emptyset \land \text{cons} \rightarrow \{y\} \cdot M := M; \; \text{cons} := \text{false} \text{ (remove')}
\end{array} \]

CONSUMER:
\[ \begin{array}{l}
s := 0; \\
do \; \neg \text{cons} \rightarrow s := s + y; \; \text{cons} := \text{true} \text{ (sum')}
\end{array} \]

We will replace the variable \( M \) by three integer variables \( q.1 \), \( q.2 \) and \( q.3 \), to hold the elements of the queue. We assume that there is a distinguished value \( \emptyset \) to indicate that no integer is presently stored in the variable. We add internal actions to the buffer that transport the values in the buffer forward from \( q.1 \) to \( q.3 \). These will correspond to stuttering actions on the abstract level. This gives us the following refinement of the buffer:

BUFFER:
\[ \begin{array}{l}
\text{let} \; q.1, q.2, q.3 \in \text{real}; \\
q.1, q.2, q.3 := \emptyset, \emptyset, \emptyset; \\
do \; \text{prod} \land q.1 \neq \emptyset \rightarrow q.1 := x; \; \text{prod} := \text{false} \text{ (insert')}
| \; q.1 \neq \emptyset \land q.2 = \emptyset \rightarrow q.1, q.2 := q.1, q.1 \text{ (forward12)}
| \; q.2 \neq \emptyset \land q.3 = \emptyset \rightarrow q.2, q.3 := q.2 \text{ (forward23)}
| \; cons \land q.3 \neq \emptyset \rightarrow q.3 := q.3, q.3; \; \text{cons} := \text{false} \text{ (remove')} \\
\end{array} \]

23
Simultaneous insertion and removal is now possible if the variables \(q.1\) and \(q.3\) are put in different partitions. The access relation is shown in Figure 6, together with a possible partitioning into processes (private processes are shown inside the partitionings, shared outside).

### Correctness of refinement step

We want to prove that this is a correct refinement, i.e., that \(BUFFER \preceq BUFFER'\). To this end, we choose the abstraction relation

\[
R(M, k, q.1, q.2, q.3) \xrightarrow{M} M = (q.3, q.2, q.1) \land k = R(q.3, q.2, q.1).
\]

Here \(R(q.3, q.2, q.1)\) is defined to be the number of forward moves that can be made for the values \(q.3, q.2, q.1\), and \((q.3, q.2, q.1)\) denotes the sequence of nonempty values. E.g. for \(q.1 = 12, q.2 = 8, q.3 = 2\) we have \((q.3, q.2, q.1) = 21, 12\) and \(R(q.3, q.2, q.1) = 1\).

The correctness of the refinement is established by showing that all three conditions in Theorem 3 hold. The first condition requires us to prove that

\[
\exists M, R(M, k, q.1, q.2, q.3) \land k = R(q.3, q.2, q.1).
\]

This is established by choosing \(k = 0\).

Let \(z = x, y, prod, cons, M\). The second condition requires us to prove that

\[
z, k = z0, k0 \land R(M, k, q.1, q.2, q.3) \land gB' \Rightarrow [sB']
\]

\[
\exists M, k, (R(M, k, q.1, q.2, q.3) \land (sA\langle gA \land z = z0 \rangle) \lor (k < M \land z = z0))
\]

for \(B' = insert', remove', forwardto2, forwardto3\) and \(A = insert \lor remove\). Here we compute

\[
sA\langle gA \land z = z0 \rangle
\]

to be

\[
(M = M0 \cdot \{z0\} \land prod = false \land prod0 = true \land M0 < 3 \land z = z0 \land y = y0 \land cons = cons0) \lor (M0 = \{y\} \cdot M \land cons = false \land cons0 = true \land M0 > 0 \land prod = prod0 \land z = z0)
\]

\[
\forall
\]

24
Inspection shows that this is indeed the case.

Finally, we need to show that the third condition is also satisfied, i.e., that
\[ R(\mathcal{M}, a, q_1, q_2, q_3) \Rightarrow (gA' = (gA \land A > 0)), \]
where
\[ gA = (\text{pred} \land \#M < 0) \lor (\text{cons} \land \#M > 0) \]
and
\[ gA' = (\text{pred} \land q.1 = 0) \lor (q.1 \neq 0 \land q.2 = 0) \lor (q.2 \neq 0 \land q.3 = 0) \lor (\text{cons} \land q.3 \neq 0). \]
This is also easily seen by checking.

This will then establish that the implementation is indeed a correct reactive refinement of the original buffer system. Thus, we can replace BUFFER with BUFFER in any reactive context, and the resulting system permits parallel accessing where the original one required mutual exclusion.

7 Concluding remarks

We have shown above how to apply the refinement calculus to the stepwise refinement of parallel and reactive systems. The basis for this approach was the action system approach, which permits us to model parallel systems as sequential nondeterministic programs, thus making them suitable targets for the refinement calculus. We have here concentrated in particular on the refinement of reactive systems. In essence, we have shown that the methods for data refinement developed in part I of this paper are sufficient for expressing the requirements for refinement of reactive systems. The approach was applied to the derivation of a small example system, to illustrate how it works in practice.

The action system approach bears a number of similarities to the UNITY approach by Chandy and Misra [14]. UNITY programs are similar to action systems, although their actions are restricted to conditional, deterministic assignment statements. Also, the fairness assumptions are hardcoded into the system. The approach to refinement of UNITY programs is different from ours: the specification of the program is refined instead of refining the program text as we do here. The logic used is a variant of temporal logic, whereas we base our reasoning in this paper on the weakest precondition calculus.

The framework built by Mann and Pnueli [28,31], using transition systems for modelling the execution of parallel and reactive programs, is quite close to the action system approach. In fact, one of the original inspirations to using action systems was to be able to use temporal logic directly to reason about execution of parallel programs, rather than to introduce a confusing transformation from some ordinary programming language to a transition system representation. The action systems are more general than transition systems are usually taken to be, as the bodies can be arbitrary, possibly nonterminating statements. However, the main difference is that we propose the action system formalism as a programming language. This makes it possible to reason directly about action systems either in temporal logic or in the weakest precondition calculus, making program derivations and proofs more syntactic in nature.

A difference in our approach, as compared to the approach by Abadi and Lamport [1] is that we do not have explicit livelock conditions for action systems. We also do not make any implicit fairness assumptions for the action systems. Different notions of fairness in action systems are studied in detail by Back and Kurki-Suonio [7], in a temporal logic framework. Here we have been content with the fact that fairness constraints can be encoded in the
action system if so desired. The encoung can, however, he quite cumbersome, so more specialized rules for refinement of action systems with implicit fairness constraints would be needed. A suitable framework for this could be the way in which fairness is modelled in Parnas [16], as a means of choice in guarded iteration statements. We ravo this as an issue for further research.

One of the main result of the study above is that refinement of reactive systems can be proved in the refinement calculus directly (and thus in the weakest precondition calculus). It seems that the more powerful temporal logic framework is not needed for proving correctness of reactive refinements. On the other hand, one could argue that temporal logic is necessary (or at least convenient) for specifying the behavior of the required reactive system. The starting point of a reactive refinement would have to be a specification in e.g., temporal logic or in UNITY, even if the successive refinement steps only work on action systems and rely solely on simulation refinement and the refinement calculus. In particular, the liveness constraints are often convenient to express in temporal logic.

Although the above position is reasonable, one can also argue for a more operational specification as the starting point of a derivation. The behavior of a reactive system is an operational notion, and automata theoretic approaches have been found quite useful in giving detailed specifications for such systems, e.g., Milner's CCS[29] and Hoare's CSP [19]. A disadvantage of such specifications is that they do not handle variables and more complicated states too well. Taking a high-level action system directly as the specification of a reactive system combines the advantages of the automata theoretic approach and the temporal logic approach: it has a simple operational interpretation but one can still reason about the properties of states variables. Liveness conditions can be expressed by action fairness assumptions. Work on applying this approach to the specification of large reactive systems has been carried out by Kurki-Suonio [23,22], combining the action system formalism, Harel's state charts [17] and object oriented programming concepts.

We have not touched upon the completeness of the simulation refinement method here. A completeness result is proved by Abadi and Lampport [2], under the condition that one may add history and prophecy variables to the specification of a reactive system. The additional assumptions that are made for this completeness result rule out the use of unboundedly non-deterministic assignment statements, which we again need to model stuttering and fairness. Hence, the completeness result by Abadi and Lampport do not carry over directly to our approach. Their framework also differs from ours in that they postulate an explicit liveness requirement of the transition system. We leave this issue also as further study.

Acknowledgements

The work reported here was supported by the FINSOFTE II program sponsored by the Technology Development Centre of Finland. I would like to thank Bengt Jonsson, Reino Kurki-Suonio, Leslie Lampport, Carroll Morgan and Amir Pnueli for helpful discussions on the topics treated here and Jaakim von Wright for useful comments on the preliminary version of this paper.

References


27


28