Stepwise refinement of reactive processor farms

K. Sere

RUU-CS-91-17
June 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Stepwise refinement of reactive processor farms

K. Sere

Technical Report RUU-CS-91-17
June 1991

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Stepwise Refinement of Reactive Processor Farms *

Kaisa Sere

Department of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
The Netherlands
e-mail: kaisa@cs.ruu.nl

Abstract

A processor farm is a distributed system that consists of a unique master processor together with a number of identical slave processors. The master processor interacts with some environment that generates tasks to be solved by the slave processors. The processors are connected via some communication network that takes care of the communication of the tasks between the master and the slaves. We show how a combination of action systems and the refinement calculus can be used to formally derive, in a stepwise manner, a communication network and protocol between the processors forming an arbitrary processor farm.

1 Introduction

A *processor farm* [9,12] is a distributed system that consists of a unique master processor together with an arbitrary number of identical slave processors connected via a communication network. An environment generates tasks that the farm is required to solve. The tasks are given to the master that distributes them to the slaves for solving. A slave returns some answer to the master for each task it solves. The network between the master and the slaves takes care of the delivery of the tasks and corresponding answers.

The processor farm paradigm has turned out to be a very useful and practical methodology for parallellizing algorithms in scientific computing [9]. We study the formal derivation of a distributed system that models the communication in an arbitrary processor farm. The derivation is carried out within the action systems framework.

*The work reported here was supported by the Finsoft III program sponsored by the Technology Development Centre of Finland. The author is on leave from Åbo Akademi University, Department of Computer Science, SF-20520 Turku, Finland. The stay of the author at Utrecht University was supported by the Dutch organisation for scientific research under project nr. NF 62-518 (Specification and Transformation Of Programs, STOP). A short version of this article will appear in the Proceedings of IFIP TC6/WG6.1 11th International Symposium on Protocol Specification, Testing and Verification, Stockholm, Sweden, 18-20 June 1991, North-Holland.

The *action system* formalism for parallel and distributed computations was introduced by Back and Kurki–Suonio in [4]. The behaviour of parallel and distributed programs is in this framework described in terms of the actions which 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 common variables. 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.

The use of action systems permits the design of the logical behaviour of a system to be separated from the issue of how the system is to be implemented. The decision whether the action system is to be executed in a sequential or parallel fashion can be postponed to a later stage, when the logical behaviour of the action system has been designed. The construction of the program is thus done within a single unifying framework. Action systems have many similarities with other event–based formalisms like UNITY [7].

The action system approach makes *stepwise refinement* of parallel programs simple and convenient. Parallel programs are just special kinds of sequential statements, so stepwise refinement of both sequential and parallel programs can be carried out within the same framework.

The *refinement calculus*, which is relies on the weakest precondition calculus of Dijkstra [8], is a formalization of the stepwise refinement approach to program construction. It was first described by Back in [2] and has been further elaborated in [6,13,14].

A method for stepwise refinement of action systems in a temporal logic framework was put forward by Back and Kurki–Suonio [4]. Refinement of action systems within the refinement calculus has been described by Back and Sere in [5,15]. The total correctness of an action system $\mathcal{A}$ is in these frameworks preserved by its refinement $\mathcal{A}'$. However, the behavior of $\mathcal{A}'$ during execution may not be the same as the behavior of $\mathcal{A}$. Hence, the input–output correctness of parallel programs is preserved, but not necessarily their *reactive behavior*, i.e., the way in which they interact with their environment during the execution. Recently these frameworks were extended by Back [3] to cover even this aspect.

We show through a case study how a proof rule for correctness of refinement in a reactive context [3] can be used to *derive* reactive systems. The application area of our case study is the communication network in a processor farm. The network together with a protocol for *communication over unreliable channels* is derived. We show how asynchronous communication over channels that store and lose messages can be modeled within our framework.

We proceed as follows. In section 2, we present the action systems formalism. The proof rule that we base our derivation on is presented in section 3. The case study is documented in sections 4–7. In section 4, an initial specification is given. In section 5, we introduce a proper form for a processor farm with one master processor and a number of slave processors. We assume a fully connected network. In section 6, we develop a system that works on a partialy connected communication network. In

section 7, a communication protocol is imposed between the master and the slaves. We end in section 8 with some remarks on the methodology.

2 Reactive systems as action systems

Action systems An *action system* $\mathcal{A}$ is an initialized iteration statement

$$\mathcal{A} = |[\ \mathbf{var}\ x; S; \mathbf{do}\ A_1 \ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od}\ ]| : z$$

on *state variables* $y = x \cup z$. The variables z are the global variables and the variables x are local to $\mathcal{A}$. Each variable is associated with some domain of values. The set of possible assignments of values to the state variables constitutes the *state space*. The initialization statement S assigns initial values to the state variables.

Each action A_i is of the form $g_i \rightarrow S_i$, where the *guard* g_i is a boolean expression and the *body* S_i a sequential statement on the state variables. The guard of action A will be denoted gA and the body sA. The state variables referenced in action A will be denoted by vA. The behaviour of an action system is that of Dijkstra's guarded iteration statement [8].

Parallel action systems Let $\mathcal{P} = \{p_1, \ldots, p_k\}$ be a partitioning of y, i.e.,

(i) $p_i \subseteq y$ and $p_i \neq \emptyset$, for $i = 1, \ldots, k$,

(ii) $\bigcup_{i=1}^{k} p_i = y$ and

(iii) $p_i \cap p_j = \emptyset$ when $i \neq j$, for $i, j = 1, \ldots, k$.

We identify each partition p_i with a *process*, with the variables in p_i as local variables. We say that action A *involves* process p, if $vA \cap p \neq \emptyset$.

Let $procA = \{p \in \mathcal{P} \mid A \text{ involves } p\}$. Two actions A and B are *independent in partitioning* $\mathcal{P}$ if $procA \cap procB = \emptyset$. An action A is *private* in $\mathcal{P}$, if $|procA| = 1$, otherwise it is *shared* in $\mathcal{P}$. We permit actions that are independent in some partitioning to be executed in parallel in that partitioning. As two independent actions do not have any variables in common, their parallel execution is equivalent to executing the actions one after the other, in either order.

Reactive action systems An action system can be viewed as a reactive program, where the system interacts with some environment (an other action system) through its global variables.

Let

$$\begin{aligned} \mathcal{A} &= |[\ \mathbf{var}\ x; S; \mathbf{do}\ A_1 \ [\!]\ \cdots\ [\!]\ A_m\ \mathbf{od}\ ]| : z \text{ and} \\ \mathcal{B} &= |[\ \mathbf{var}\ y; T; \mathbf{do}\ B_1 \ [\!]\ \cdots\ [\!]\ B_n\ \mathbf{od}\ ]| : u \end{aligned}$$

where S and T only initialize the local variables x and y respectively. The *parallel composition* $\mathcal{A} \parallel \mathcal{B} : z \cup u$ of $\mathcal{A}$ and $\mathcal{B}$ is defined to be

$$\mathcal{A} \parallel \mathcal{B} = |[\ \mathbf{var}\ x, y; S; T; \mathbf{do}\ A_1 \ [\!]\ \cdots\ [\!]\ A_m\ [\!]\ B_1\ [\!]\ \cdots\ [\!]\ B_n\ \mathbf{od}\ ]| : z \cup u$$

where we assume that $x \cap y = \emptyset$. This is the same as the union operator in UNITY [7], except that we keep track of which variables are local and which global (UNITY only has global variables).

Let $\mathcal{A}$ be the action system above. Let $z = z_1, z_2$. We can *hide* some of the variables in $\mathcal{A}$ by making them local as follows

$$\mathcal{A}' = |[\ \mathbf{var}\ z_1; \mathcal{A}\ ]| : z_2$$

Hiding the variables z_1 makes them inaccessible to the actions outside $\mathcal{A}'$ in a parallel composition of action systems. Using parallel composition and hiding we can structure large action systems into smaller, reactive action systems, which commuticate with each other via the shared, visible variables.

Consider an action system C

$$C = |[\ \mathbf{var}\ u; S; \mathbf{do}\ C_1 \ [\!]\ \cdots\ [\!]\ C_m\ \mathbf{od}\ ]| : z.$$

Let $A = \{A_1, \ldots, A_k\}$ and $B = \{B_1, \ldots, B_l\}$ be a partitioning of the actions in C with

$$\begin{aligned} x &= vA - vB - z \\ y &= vB - vA - z \\ w &= vA \cap vB - z. \end{aligned}$$

We can then write C as follows

$$C' = |[\ \mathbf{var}\ w; S'; \mathcal{A} \parallel \mathcal{B}\ ]| : z$$

where the *reactive components* $\mathcal{A}$ and $\mathcal{B}$ are

$$\begin{aligned} \mathcal{A} &= |[\ \mathbf{var}\ x; T_1; \mathbf{do}\ A_1 \ [\!]\ \cdots\ [\!]\ A_k\ \mathbf{od}\ ]| : w, z \\ \mathcal{B} &= |[\ \mathbf{var}\ y; T_2; \mathbf{do}\ B_1 \ [\!]\ \cdots\ [\!]\ B_l\ \mathbf{od}\ ]| : w, z. \end{aligned}$$

Here $S' : w, z$ initializes the variables w, z, $T_1 : x, w, z$ the variables x and $T_2 : y, w, z$ initializes the variables y so that $|[\ \mathbf{var}\ u; S\ ]| : z = |[\ \mathbf{var}\ u; S'; T_1; T_2\ ]| : z$. The reactive components interact via the visible variables w and z.

3 Derivation of reactive action systems

Reactive refinement Consider an action system $\mathcal{A} : x, z$ and its reactive refinement (to be defined) $\mathcal{A}' : x', z$, where the local variables x, x' of the action systems may be different, but the global variables z must be the same. We either have that there is a one–to–one correspondence between actions in $\mathcal{A}$ and $\mathcal{A}'$, or that executing a single action in $\mathcal{A}$ corresponds to executing a sequence of two or more actions in $\mathcal{A}'$. In the latter case we require that the actions of $\mathcal{A}$ are simulated by the actions of $\mathcal{A}'$ as will be explained below.

We permit so called *stuttering actions* in $\mathcal{A}'$, i.e., actions which do not correspond to any state change in $\mathcal{A}$. We have that for any execution of the action system $\mathcal{A}$, the

meaning of $\mathcal{A}$ 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 $\mathcal{A}$ 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. This framework is similar to the approaches of Abadi and Lamport [1], Jonsson [10], Lam and Shankar [11] and others.

We have the following general rule for proving refinement in reactive contexts [3].

THEOREM 1 *Let $\mathcal{A}$ and $\mathcal{A}'$ be two action systems with local variables $x = u, v$ and $x' = u', v$ respectively and with the global variables z. Then $\mathcal{A}$ is* reactively refined *by action system $\mathcal{A}'$, denoted $\mathcal{A} \preceq \mathcal{A}'$, if there exists a relation $R(u, h, u', v)$ such that the following conditions hold:*

(i) $\exists h.R(u0, h, u'0, v0)$.

(ii) *For each action A' in $\mathcal{A}'$*

$$x, h, z = x0, h0, z0 \wedge R(u, h, u', v) \wedge gA'$$
$$[sA']$$
$$\exists u, h.(R(u, h, u', v) \wedge (s\mathcal{A}^{-}(g\mathcal{A} \wedge x, z = x0, z0) \vee (h < h0 \wedge x, z = x0, z0))$$

must hold.

(iii) $R(u, h, u', v) \Rightarrow (g\mathcal{A}' \equiv (g\mathcal{A} \vee h > 0))$.

Here $P[S]Q$ denotes the total correctness of statement S w.r.t. P and Q and $S^{-}(P)$, the generalized inverse statement of S [6], corresponds to the strongest postcondition of S w.r.t. P. The 0–indexed values denote initial values for the variables. Further, let $\mathcal{A} = [\![$ **var** x; S; **do** $A_1 \,[\!]\, \ldots \,[\!]\, A_m$ **od** $]\!] : z$, which is equivalent to $[\![$ **var** x; S; **do** $\bigvee_{i=1}^{m} gA_i \rightarrow$ **if** $A_1 \,[\!]\, \ldots \,[\!]\, A_m$ **fi od** $]\!] : z$. Then $g\mathcal{A} = \bigvee_{i=1}^{m} gA_i$ and $s\mathcal{A} =$ **if** $A_1 \,[\!]\, \ldots \,[\!]\, A_m$ **fi**.

Theorem 1 (ii) states that an action in $\mathcal{A}'$ corresponds to an action in $\mathcal{A}$, if it corresponds to a change in the state in $\mathcal{A}$. It corresponds to a stuttering action, if it does not correspond to any change in the state in $\mathcal{A}$

Refinement of reactive action systems The reactive refinement relation has the basic properties required of a refinement relation, i.e., reflexivity, transitivity and monotonicity [3].

THEOREM 2 *Let $\mathcal{A}, \mathcal{A}', \mathcal{A}'', \mathcal{B}, \mathcal{B}'$ be action systems and w a list of variables. Then the following properties hold:*

(i) $\mathcal{A} \preceq \mathcal{A}$.

(ii) $\mathcal{A} \preceq \mathcal{A}' \preceq \mathcal{A}'' \Rightarrow \mathcal{A} \preceq \mathcal{A}''$.

(iii) *If $\mathcal{A} \preceq \mathcal{A}'$ and $\mathcal{B} \preceq \mathcal{B}'$ then*

(a) $\mathcal{A} \parallel \mathcal{B} \preceq \mathcal{A}' \parallel \mathcal{B}'$,

(b) $|[\ \mathbf{var}\ w; \mathcal{A}\ ]| \preceq |[\ \mathbf{var}\ w; \mathcal{A}'\ ]|$.

This means that we can do stepwise refinement in terms of reactive refinement. Starting from some action system $\mathcal{A}_0$ that serves as the initial specification, we can construct a sequence of action system refinements

$$\mathcal{A}_0 \preceq \mathcal{A}_1 \preceq \ldots \preceq \mathcal{A}_n,$$

until we reach a reactive system $\mathcal{A}_n$ that is considered adequate.

Reactive refinement relation is monotonic with respect to parallel composition and hiding. This implies that we may replace any *reactive component* $\mathcal{A}$ of a reactive system $\mathcal{C}[\mathcal{A}]$ with its reactive refinement $\mathcal{A}'$. In other words, we always have that

$$\mathcal{A} \preceq \mathcal{A}' \Rightarrow \mathcal{C}[\mathcal{A}] \preceq \mathcal{C}[\mathcal{A}'].$$

A reactive component is here a component built out of action systems using parallel composition and hiding.

Preserving temporal properties The reactive refinement relation preserves temporal logic properties: if $\mathcal{A}$ satisfies a temporal logic formula ϕ and $\mathcal{A} \preceq \mathcal{A}'$, then $\mathcal{A}'$ will also satisfy ϕ, provided that ϕ is *insensitive to stuttering.* A temporal logic formula ϕ is said to be insensitive to stuttering, if $\phi(s) \Leftrightarrow \phi(s')$ holds for any two execution sequencies s and s' that are the same if any finite sequence of stuttering transitions are removed from s and s'.

4 Case study: A processor farm in its environment

As a case study in stepwise refinement of reactive systems we derive an action system that models the communication structure in a processor farm [9,12]. The farm consists of a unique master processor and an arbitrary number of identical slave processors. The purpose of the slaves is to solve in parallel the tasks which originally reside in the master. After solving a task, each slave returns an answer to the master and receives a new task. Tasks are assumed to be unique and they can be solved in any order. We assume for simplicity that the tasks are arbitrary integers. We are not interested in what it means to actually solve a task.

The tasks are generated by some environment system (not specified here) that executes in parallel with the farm. They are given to the processor farm for solving through a queue q. The answers are delivered to the environment through a queue s. The queues belong to the partition corresponding to the process executing on the master processor.

Our initial specification $\mathcal{F}_0$ for the processor farm is as follows:

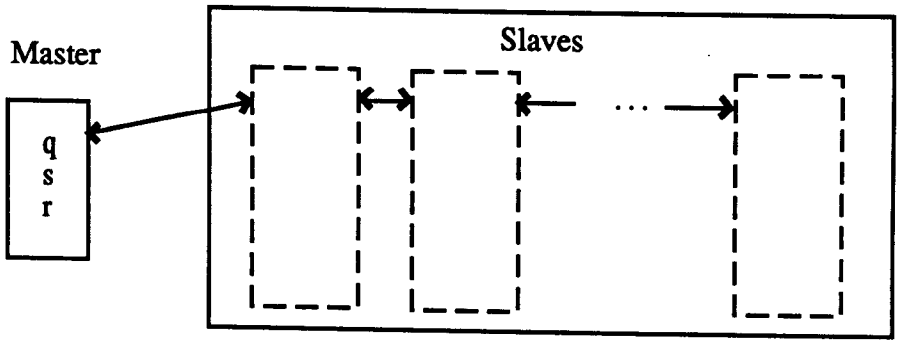

Figure 1: A processor farm.

$$
\begin{array}{lll}
\mathcal{F}_0 = & |[\ \textbf{var}\ r \in \textit{bag of integer}; & \\
& r := \emptyset; & \\
& \textbf{do} & \\
& \quad [\!] \ |q| > 0 \rightarrow x, q := hd(q), tl(q); r := r \cup f(x) & (\textit{ins}) \\
& \quad [\!] \ |r| > 0 \rightarrow x := x'.\{x' \in r\}; r := r - x; s := s \cdot x & (\textit{rem}) \\
& \textbf{od} & \\
&]|: q, s \in \textit{queue of integer} &
\end{array}
$$

The corresponding partitioning is $\mathcal{P} = \{\{q, s, r\}\}$. The *nondeterministic assignment statement* $x := x'.Q$ [2] assigns to the variable x an value x' so that assertion Q holds for x and x'. The queues and bags are unbounded and have the capacity of storing at least 1 task. A centered dot $\cdot$ denotes concatenation of an element with a queue. The integer variable x is assumed to be local to each action body.

We have described the behaviour of the farm with two separate actions, *ins* and *rem*. If only one action is used, an outside observer can observe a temporal property, where each task that is removed from q, is immediatly solved and the answer is inserted to s without delay. Our solution models the fact that this takes time. The tasks can be solved in arbitrary order. We model this property by the bag r: should r be a queue, we would observe a temporal property where tasks leave q and corresponding answers enter s in the same order. We assume that the elements are chosen from r in a fair manner. (For details in fairness issues in this context, the reader is refered to [3].)

In subsequent sections we show how the action system $\mathcal{F}_0$, which interacts with its environment through the visible variables q and s, is stepwise turned into a processor farm with a master processor together with n, $n \geq 1$, slave processors. The processors are connected via a point–to–point network. There is no global memory among them. Our target architecture is presented in Figure 1.

5 Master and slaves

Refinement step: Split an action We start by creating a solving action. In the final version we want the actual task solving to be a private action of each slave. We split the action *ins* into two separate actions: one action for removing tasks from q and an other for solving a task. We replace the bag r with a bag $o1$ that holds the tasks and a queue $o2$ that holds the answers: if $o1$ would be a queue, we could observe tasks floating through the farm in a FIFO order. This replacement can be carried out as r is local to $\mathcal{F}_0$.

The refining reactive action system $\mathcal{F}_1$ is as follows:

$\mathcal{F}_1 =$ ||[**var** o1 ∈ **bag of integer**; o2 ∈ **queue of integer**;
o1, o2 := ∅, ∅;
do
[] $|q| > 0 \rightarrow x, q := hd(q), tl(q)$; o1 := o1 ∪ x (*ins'*)
[] $|o1| > 0 \rightarrow$ x := x'.{x' ∈ o1}; o1 := o1 − x; o2 := o2 · f(x) (*sol*)
[] $|o2| > 0 \rightarrow$ x, o2 := hd(o2), tl(o2); $s := s \cdot x$ (*rem'*)
od
]|: $q, s \in$ *queue of integer*

(We highlight the refined statements with a different font.) The underlying variable partitioning is $\mathcal{P} = \{\{q, s, o1, o2\}\}$.

The action *ins'* in $\mathcal{F}_1$ corresponds to action *ins* in $\mathcal{F}_0$ and *rem'* in $\mathcal{F}_1$ corresponds to action *rem* in $\mathcal{F}_0$. The new action *sol* does not correspond to any state change in $\mathcal{F}_0$. Hence, it corresponds to a stuttering action. The correctness of this reactive refinement is shown by the following lemma.

LEMMA 1 $\mathcal{F}_0 \preceq \mathcal{F}_1$.

PROOF Let $R(r, h, o1, o2) =_{df} h = |o1| \wedge r = (f(o1) \cup o2)$. Here h equals the number of tasks in bag $o1$, i.e., the maximum possible number of consequtive *sol* actions (stutterings).

We have to show that the conditions (i)–(iii) in Theorem 1 are satisfied with this relation:

(i) $\exists h.R(r0, h, o10, o20)$ holds with $h = 0$.

(ii) It is easily checked that for each action A in $\mathcal{F}_1$

$$r, h, q, s = r0, h0, q0, s0 \wedge R(r, h, o1, o2) \wedge gA$$
$$[sA]$$
$$\exists r, h.(R(r, h, o1, o2) \wedge (s\mathcal{F}_0^-(g\mathcal{F}_0 \wedge r, q, s = r0, q0, s0) \vee h < h0 \wedge r, q, s = r0, q0, s0))$$

holds. We have that $s\mathcal{F}_0^-(g\mathcal{F}_0 \wedge r, q, s = r0, q0, s0)$ is

$$q = tl(q0) \wedge r = r0 \cup f(hd(q0)) \wedge |q0| > 0$$
$$\vee$$
$$\exists x \in r0\,.\, r = r0 - x \wedge s = s0 \cdot x \wedge |r0| > 0.$$

(iii) $R(r,h,o1,o2) \Rightarrow (g\mathcal{F}_1 \equiv (g\mathcal{F}_0 \vee h > 0))$ where $g\mathcal{F}_0 = |q| > 0 \vee |r| > 0$, $g\mathcal{F}_1 = |q| > 0 \vee |o1| > 0 \vee |o2| > 0$ and $h = |o1|$. We have that

($\Leftarrow$) Assume $g\mathcal{F}_0 \vee h > 0$. Then either $|q| > 0$, which implies $|q| > 0$, or $|r| > 0$, which implies $|f(o1) \cup o2| > 0$. In the latter case either $|f(o1)| > 0$, which implies $|o1] > 0$, or $|o2| > 0$. Finally, if $h > 0$, then $|o1] > 0$.

($\Rightarrow$) Assume $g\mathcal{F}_1$. Then either $|q| > 0$, which implies $|q| > 0$, or $|o1| > 0$, which implies $h > 0$, or $|o2| > 0$, which implies $|r| > 0$.

□

Refinement step: Introduce the slaves We now bring about the n slaves in the form of the variables $y.i$, $i = 1,\ldots,n$. The bag $o1$ is replaced with a queue $p1$. The queue $o2$ is replaced by the variables y and a queue $p2$ that holds the answers. The booleans $idle.i$, $i = 1,\ldots,n$, denote whether a slave holds a solved task or not. The action *sol* is splitted into $2n$ separate actions and the associated variable partitioning is $\mathcal{P} = \{\{q,s,\mathtt{p1},\mathtt{p2}\},\{\mathtt{y.i},\mathtt{idle.i}\}$ **for** $i = 1,\ldots,n\}$. (We indicate replication of declarations, statements and actions by a **for**–clause after the construct.)

The new version of the farm is as follows:

$\mathcal{F}_2 =$ |[**var** p1,p2 ∈ **queue of integer**;
idle.i ∈ **boolean**, **y.i** ∈ **integer** **for** $i = 1,\ldots,n$;
p1,p2 := ∅,∅; **idle.i** = **true** **for** $i = 1,\ldots,n$;
do
[] $|q| > 0 \rightarrow x, q := hd(q), tl(q)$; p1 := p1 · x — (ins'')
[] |p1| > 0 ∧ **idle.i** → — $(s1.i)$
x,p1 := hd(p1), tl(p1); y.i, idle.i := f(x), **false**
for $i = 1,\ldots,n$
[] ¬**idle.i** → p2 := p2 · y.i; **idle.i** := **true** — $(s2.i)$
for $i = 1,\ldots,n$
[] |p2| > 0 → x,p2 := hd(p2), tl(p2); $s := s \cdot x$ — (rem'')
od
]|: $q, s \in$ *queue of integer*

Each action in $\mathcal{F}_2$ is shared by a master–slave pair in partitioning $\mathcal{P}$.

The action ins'' in $\mathcal{F}_2$ corresponds to action ins' in $\mathcal{F}_1$ and rem'' in $\mathcal{F}_2$ corresponds to rem' action in $\mathcal{F}_1$. The new actions $s1.i, s2.i$, $i = 1,\ldots,n$, in $\mathcal{F}_2$ correspond to stuttering actions. The correctness of the refinement is shown by the following lemma.

LEMMA 2 $\mathcal{F}_1 \preceq \mathcal{F}_2$.

PROOF The lemma follows from Theorem 1 with $R(o1,o2,h,idle,p1,y,p2) =_{df} h = H \wedge o1 = p1 \wedge o2 = O \cup p2$ where H is the number of slaves currently busy (i.e., for which $\neg idle.i$ holds) and O is the set of answers $y.i$ held by the currently busy slaves.
□

Refinement step: Separate task solving from communication As a final refinement in creating the slaves, we isolate the task solving into a private action for each slave. Let us first decompose the system:

$$
\begin{array}{ll}
\mathcal{F}_3 = & [\![\ \textbf{var}\ p1, p2 \in \textit{queue of integer}; \\
& \quad p1, p2 := \emptyset, \emptyset;\ \mathcal{M} \parallel \mathcal{S}_0 \\
&]\!]: q, s \in \textit{queue of integer}
\end{array}
$$

where the master $\mathcal{M}$ is

$$
\begin{array}{ll}
\mathcal{M} = & [\![\ \textbf{do} \\
& \quad [\!]\ |q| > 0 \rightarrow x, q := hd(q), tl(q);\ p1 := p1 \cdot x & (ins'') \\
& \quad [\!]\ |p2| > 0 \rightarrow x, p2 := hd(p2), tl(p2);\ s := s \cdot x & (rem'') \\
& \quad \textbf{od} \\
&]\!]: q, s, p1, p2 \in \textit{queue of integer}
\end{array}
$$

and the slaves $\mathcal{S}_0$ are

$$
\begin{array}{ll}
\mathcal{S}_0 = & [\![\ \textbf{var}\ [idle.i \in boolean, y.i \in integer]\ \textbf{for}\ i = 1, \ldots, n; \\
& \quad idle.i = true\ \ \textbf{for}\ i = 1, \ldots, n; \\
& \quad \textbf{do} \\
& \quad [\!]\ |p1| > 0 \wedge idle.i \rightarrow & (s1.i) \\
& \qquad x, p1 := hd(p1), tl(p1);\ y.i, idle.i := f(x), false \\
& \quad \textbf{for}\ i = 1, \ldots, n \\
& \quad [\!]\ \neg idle.i \rightarrow p2 := p2 \cdot y.i;\ idle.i := true & (s2.i) \\
& \quad \textbf{for}\ i = 1, \ldots, n \\
& \quad \textbf{od} \\
&]\!]: p1, p2 \in \textit{queue of integer}
\end{array}
$$

We have that $\mathcal{F}_2 = \mathcal{F}_3$. Each action is still shared by a unique master–slave pair.

We now refine the reactive slave component $\mathcal{S}_0$ separately from the master. We replace the variables y by the new variables z, and add a boolean array $rec.i$, $i = 1, \ldots, n$. Each boolean $rec.i$ denotes whether the corresponding slave has received a task for solving or not. The partitioning is $\mathcal{P} = \{\{p1, p2\}, \{idle.i, \texttt{z.i}, \texttt{rec.i}\}\ \textbf{for}\ i = 1, \ldots, n\}$.

The refined system is as follows:

$$
\begin{array}{ll}
\mathcal{S}_1 = & [\![\ \textbf{var}\ idle.i, \texttt{rec.i} \in boolean, \texttt{z.i} \in \texttt{integer}\ \ \textbf{for}\ i = 1, \ldots, n; \\
& \quad idle.i, \texttt{rec.i} = true, \texttt{false}\ \ \textbf{for}\ i = 1, \ldots, n; \\
& \quad \textbf{do} \\
& \quad [\!]\ |p1| > 0 \wedge idle.i \rightarrow & (s1'.i) \\
& \qquad x, p1 := hd(p1), tl(p1);\ \texttt{z.i}, idle.i, \texttt{rec.i} := \texttt{x}, false, \texttt{true} \\
& \quad \textbf{for}\ i = 1, \ldots, n \\
& \quad [\!]\ \texttt{rec.i} \rightarrow \texttt{z.i} := \texttt{f(z.i)};\ \texttt{rec.i} := \texttt{false} & (ss.i) \\
& \quad \textbf{for}\ i = 1, \ldots, n \\
& \quad [\!]\ \neg idle.i \wedge \neg \texttt{rec.i} \rightarrow \texttt{p2} := \texttt{p2} \cdot \texttt{z.i};\ idle.i := true & (s2'.i) \\
& \quad \textbf{for}\ i = 1, \ldots, n \\
& \quad \textbf{od} \\
&]\!]: p1, p2 \in \textit{queue of integer}
\end{array}
$$

We now have that the solving actions ss are private in partitioning $\mathcal{P}$.

The actions $s1'.i$ in $\mathcal{S}_1$ correspond to actions $s1.i$ in $\mathcal{S}_0$ and $s2'.i$ in $\mathcal{S}_1$ correspond to actions $s2.i$ in $\mathcal{S}_0$. The new actions $ss.i$, $i = 1, \ldots, n$, in $\mathcal{S}_1$ correspond to stuttering actions. The correctness of this reactive refinement is shown as follows.

LEMMA 3 $\mathcal{S}_0 \preceq \mathcal{S}_1$.

PROOF The correctness of the refinement follows from Theorem 1 using $R(y, h, z, rec, idle)$ $=_{df}$ $h = H \wedge \forall i.1..n.\ P.i \wedge Q.i$ where H is the number of slaves currently holding an unsolved task (i.e., for which $rec.i$ holds), $P.i =_{df} \neg idle.i \wedge rec.i \Rightarrow y.i = f(z.i)$ and $Q.i =_{df} \neg idle.i \wedge \neg rec.i \Rightarrow y.i = z.i$. □

Summing up Let $\mathcal{F}_4$ be as follows

$$
\begin{array}{l}
\mathcal{F}_4 = \ |[\ \textbf{var}\ p1, p2 \in \textit{queue of integer}; \\
\qquad p1, p2 := \emptyset, \emptyset;\ \mathcal{M} \parallel \mathcal{S}_1 \\
\qquad]|:\ q, s \in \textit{queue of integer}
\end{array}
$$

We then have the following theorem:

THEOREM 3 $\mathcal{F}_3 \preceq \mathcal{F}_4$.

PROOF We have that $\mathcal{S}_0 \preceq \mathcal{S}_1$ by Lemma 3. Then $\mathcal{F}_3 \preceq \mathcal{F}_4$ follows by Theorem 2 using monotonicity of reactive refinement. □

6 A slave network

Let us turn our attention to the communication actions $s1'$ and $s2'$. We want to decentralize the queues $p1$ and $p2$ so that each slave would have a task queue and an answer queue of its own. However, we first have decompose the system $\mathcal{F}_4$ anew, as the target queues are visible to $\mathcal{M}$ in $\mathcal{S}_1$. Let therefore

$$
\begin{array}{l}
\mathcal{F}_5 = \ |[\ \textbf{var}\ idle.i, rec.i \in \textit{boolean}, z.i \in \textit{integer}\ \ \textbf{for}\ i = 1, \ldots, n; \\
\qquad idle.i, rec.i = true, false\ \ \textbf{for}\ i = 1, \ldots, n;\ \mathcal{T}_0 \parallel \mathcal{S}_2 \parallel \mathcal{A}_0 \\
\qquad]|:\ q, s \in \textit{queue of integer}
\end{array}
$$

where the task system $\mathcal{T}_0$ is

$$
\begin{array}{lll}
\mathcal{T}_0 = \ |[\ \textbf{var}\ p1 \in \textit{queue of integer}; & & \\
\qquad p1 := \emptyset; & & \\
\qquad \textbf{do} & & \\
\qquad\quad [\!] \ |q| > 0 \rightarrow x, q := hd(q), tl(q);\ p1 := p1 \cdot x & & (ins'') \\
\qquad\quad [\!] \ |p1| > 0 \wedge idle.i \rightarrow & & (s1'.i) \\
\qquad\qquad x, p1 := hd(p1), tl(p1);\ z.i, idle.i, rec.i := x, false, true & & \\
\qquad \textbf{for}\ i = 1, \ldots, n & & \\
\qquad \textbf{od} & & \\
\qquad]|:\ q \in \textit{queue of integer}; idle.i, rec.i \in \textit{boolean}, z.i \in \textit{integer}\ \ \textbf{for}\ i = 1, \ldots, n & &
\end{array}
$$

the solving actions are

$$
\begin{array}{ll}
\mathcal{S}_2 = & [\![\ \textbf{do} \\
& \quad [\!]\ rec.i \rightarrow z.i := f(z.i);\ rec.i := false \qquad (ss.i) \\
& \quad \textbf{for } i = 1, \ldots, n \\
& \quad \textbf{od} \\
&]\!]: rec.i \in boolean, z.i \in integer \ \textbf{ for } i = 1, \ldots, n
\end{array}
$$

and the answer system $\mathcal{A}_0$ is

$$
\begin{array}{ll}
\mathcal{A}_0 = & [\![\ \textbf{var } p2 \in queue\ of\ integer; \\
& \quad p2 := \emptyset; \\
& \quad \textbf{do} \\
& \quad [\!]\ \neg idle.i \wedge \neg rec.i \rightarrow p2 := p2 \cdot z.i;\ idle.i := true \qquad (s2'.i) \\
& \quad \textbf{for } i = 1, \ldots, n \\
& \quad [\!]\ |p2| > 0 \rightarrow x, p2 := hd(p2), tl(p2);\ s := s \cdot x \qquad (rem'') \\
& \quad \textbf{od} \\
&]\!]: s \in queue\ of\ integer;\ idle.i, rec.i \in boolean, z.i \in integer \ \textbf{ for } i = 1, \ldots, n
\end{array}
$$

We have that $\mathcal{F}_4 = \mathcal{F}_5$. The reactive components $\mathcal{T}_0$ and $\mathcal{A}_0$ are now refined separately. We only show how to further refine $\mathcal{T}_0$, $\mathcal{A}_0$ is treated in a similar manner.

Refinement step: A task array Let us decentralize the local queue $p1$ of $\mathcal{T}_0$ by replacing it with the queues $u.1, \ldots, u.n$ with the variable partitioning $\mathcal{P} = \{\{q\}, \{\mathtt{u.i}\}, \{idle.i,\ z.i, rec.i\} \ \textbf{for } i = 1, \ldots, n\}$. This models a processor farm where each slave has a separate communication process to receive and forward tasks.

The reactive system $\mathcal{T}_0$ is refined to $\mathcal{T}_1$ below:

$$
\begin{array}{ll}
\mathcal{T}_1 = & [\![\ \textbf{var } \mathtt{u.i} \in \textbf{queue of integer } \textbf{for } i = 1, \ldots, n; \\
& \quad \mathtt{u.i} = \emptyset \ \textbf{ for } i = 1, \ldots, n; \\
& \quad \textbf{do} \\
& \quad [\!]\ |q| > 0 \rightarrow x, q := hd(q), tl(q);\ \mathtt{u.1} := \mathtt{u.1} \cdot \mathtt{x} \qquad (ins''') \\
& \quad [\!]\ |\mathtt{u.i}| > 0 \rightarrow \qquad (ft.i) \\
& \qquad\quad \mathtt{x, u.i} := \mathtt{hd(u.i), tl(u.i)};\ \mathtt{u.(i+1)} := \mathtt{u.(i+1)} \cdot \mathtt{x} \\
& \quad \textbf{for } i = 1, \ldots, n-1 \\
& \quad [\!]\ |\mathtt{u.i}| > 0 \wedge idle.i \rightarrow \qquad (s1''.i) \\
& \qquad\quad \mathtt{x, u.i} := \mathtt{hd(u.i), tl(u.i)};\ z.i, idle.i, rec.i := x, false, true \\
& \quad \textbf{for } i = 1, \ldots, n \\
& \quad \textbf{od} \\
&]\!]: q \in queue\ of\ integer;\ idle.i, rec.i \in boolean, z.i \in integer \ \textbf{ for } i = 1, \ldots, n
\end{array}
$$

The action ins''' in $\mathcal{T}_1$ corresponds to the action ins'' in $\mathcal{T}_0$ and the actions $s1''.i$ in $\mathcal{T}_1$ correspond to actions $s1'.i$ in $\mathcal{T}_0$. The new actions $ft.i$, $i = 1, \ldots, n$, in $\mathcal{T}_1$ correspond to stuttering actions as they do not correspond to any state change in $\mathcal{T}_0$. The correctness of this reactive refinement is shown by the following lemma.

LEMMA 4 $\mathcal{T}_0 \preceq \mathcal{T}_1$.

PROOF Let $R(p1, h, u) =_{df} h = |\cup_{i=1}^{n} u.i| \wedge p1 = \cup_{i=1}^{n} u.i$. The lemma follows from Theorem 1. □

Summing up Let $\mathcal{F}_6$ be as follows

$\mathcal{F}_6 =$ |[**var** $idle.i, rec.i \in$ *boolean*, $z.i \in$ *integer* **for** $i = 1, \ldots, n$;
$idle.i, rec.i =$ *true*, *false* **for** $i = 1, \ldots, n$; $\mathcal{T}_1 \parallel \mathcal{S}_2 \parallel \mathcal{A}_0$
]|: $q, s \in$ *queue of integer*

with variable partitioning $\mathcal{P} = \{\{q, s, p2\}, \{u.i\}, \{idle.i, rec.i, z.i\}$ **for** $i = 1, \ldots, n\}$. In this partitioning, the task solving can go on simultaneously with the communication of the tasks and answers due to the independence of the actions.

We have the following theorem:

THEOREM 4 $\mathcal{F}_5 \preceq \mathcal{F}_6$.

PROOF We have that $\mathcal{T}_0 \preceq \mathcal{T}_1$ by Lemma 4. The theorem now follows by Theorem 2 using the monotonicity of reactive refinement. □

7 A communication protocol

Finally, we refine the network further by introducing a more elaborate communication protocol between some of the processors: we require that each task that is sent by the master is acknowledged by the first slave. If a task is not acknowledged, it will be resent.

Let $\mathcal{F}_7$ be as follows

$\mathcal{F}_7 =$ |[**var** $u.2 \in$ *queue of integer* **for** $n > 1$; $idle.1, rec.1 \in$ *boolean*, $z.1 \in$ *integer*;
$idle.1, rec.1 :=$ *true*, *false*; $u.2 = \emptyset$ **for** $n > 1$; $\mathcal{C}_0 \parallel \mathcal{R}$
]|: $q, s \in$ *queue of integer*

where

$\mathcal{C}_0 =$ |[**var** $u.1 \in$ *queue of integer*;
$u.1 = \emptyset$;
do
[] $|q| > 0 \rightarrow x, q := hd(q), tl(q)$; $u.1 := u.1 \cdot x$ (ins''')
[] $|u.1| > 0 \rightarrow x, u.1 := hd(u.1), tl(u.1)$; $u.2 := u.2 \cdot x$ $(ft.1)$
for $n > 1$
[] $|u.1| > 0 \wedge idle.1i \rightarrow$ $(s1''.1)$
$x, u.1 := hd(u.1), tl(u.1)$; $z.1, idle.1, rec.1 := x,$ *false*, *true*
od
]|: $q, u.2 \in$ *queue of integer*; $idle.1, rec.1 \in$ *boolean*, $z.1 \in$ *integer*

and where $\mathcal{R}$ contains the rest of the actions in $\mathcal{F}_6$ in the usual manner. We have that $\mathcal{F}_6 = \mathcal{F}_7$. We have now isolated the task communication between the master and the first slave by making $u.1$ local to $\mathcal{C}_0$.

Refinement step: Ack–messages We model the acknowledgements to be sent in actions $ft.1$ and $s1''.1$ by a new queue au that contains the ack–messages, here the tasks in the order received. This queue resigns in the partition corresponding to the first slave.

$C_1 =$ |[**var** $u.1, \mathrm{au} \in$ *queue of integer*;
$u.1, \mathrm{au} := \emptyset, \emptyset$;
do
[] $|q| > 0 \rightarrow x, q := hd(q), tl(q);\ u.1 := u.1 \cdot x$ (ins''')
[] $|u.1| > 0 \rightarrow$ $(ft'.1)$
$x, u.1 := hd(u.1), tl(u.1);\ u.2, \mathrm{au} := u.2 \cdot x, \mathrm{au} \cdot \mathbf{x}$
for $n > 1$
[] $|u.1| > 0 \wedge idle.1 \rightarrow$ $(s1'''.1)$
$x, u.1 := hd(u.1), tl(u.1);$
$z.1, idle.1, rec.1, \mathrm{au} := x, false, true, \mathrm{au} \cdot \mathbf{x}$
[] $|\mathrm{au}| > 0 \rightarrow \mathrm{au} := \mathtt{tl}(\mathrm{au})$ (ack)
od
]|: $q, u.2 \in$ *queue of integer*; $idle.1, rec.1 \in$ *boolean*, $z.1 \in$ *integer*

The new action ack in C_1 corresponds to a stuttering action as there is no correspondance to it in C_0. The correctness of this reactive refinement is shown below.

LEMMA 5 $C_0 \preceq C_1$.

PROOF Let $R(h, au) =_{df} h = |au|$. The lemma then follows from Theorem 1. □

Refinement step: A loss–only channel Until now we have associated variables with processes. But we can also associate certain variables with *channels* as in the UNITY framework [7]. In our processor farm, all the queues could in principle be implemented by channels as they are accessed in a very restricted manner: there are actions that only insert elements to the end of queues and others that only remove elements from the front of queues provided the target queue is not empty. The following action system, a reactive refinement of C_1, models communication through a channel that may lose messages.

Let us assume that $u.1$ and au in C_1 are unbounded channels between the master and the first slave. Assume further, that the master keeps a copy, su, of $u.1$. When an acknowledgement message via au in an action ack is received, the corresponding task, which is the first element of su, is removed. Tasks are retransmitted along $u.1$, if an acknowledgement for which $hd(au) \neq hd(su)$ holds is received.

$C_2 =$ $[\![$ **var** $\mathtt{t.1}, au, \mathtt{su} \in$ *queue of integer*;
$\mathtt{t.1}, au, \mathtt{su} := \emptyset, \emptyset, \emptyset$;
do
$[]$ $|q| > 0 \rightarrow x, q := hd(q), tl(q); \mathtt{t.1}, \mathtt{su} := \mathtt{t.1} \cdot \mathtt{x}, \mathtt{su} \cdot \mathtt{x}$ (ins'''')
$[]$ $|\mathtt{t.1}| > 0 \rightarrow$ $(ft'.1)$
$\mathtt{x}, \mathtt{t.1} := \mathtt{hd(t.1)}, \mathtt{tl(t.1)}; u.2, au := u.2 \cdot x, au \cdot x$
for $n > 1$
$[]$ $|\mathtt{t.1}| > 0 \wedge idle.i \rightarrow$ $(s1'''.1)$
$\mathtt{x}, \mathtt{t.1} := \mathtt{hd(t.1)}, \mathtt{tl(t.1)};$
$z.1, idle.1, rec.1, au := x, false, true, au \cdot x$
$[]$ $|au| > 0 \rightarrow$ (ack')
do $\mathtt{hd(au)} \neq \mathtt{hd(su)} \rightarrow \mathtt{t.1}, \mathtt{su} := \mathtt{t.1} \cdot \mathtt{hd(su)}, \mathtt{tl(su)} \cdot \mathtt{hd(su)}$ **od**;
$au, \mathtt{su} := tl(au), \mathtt{tl(su)}$
od
$]\!]$: $q, u.2 \in$ *queue of integer*; $idle.1, rec.1 \in$ *boolean*, $z.1 \in$ *integer*

We have replaced $u.1$ by $t.1$ as technically $u.1$ in C_1 is not the same variable as $t.1$ in C_2.

There are no stuttering actions in C_2, but some code is added to actions ins''' and ack.

LEMMA 6 $C_1 \preceq C_2$.

PROOF Let $R(u.1, t.1, su, au) =_{df} u.1 = t.1 \wedge |au| > 0 \Rightarrow hd(su) = hd(au)$. The lemma follows by Theorem 1. □

Summing up Let $\mathcal{F}_8$ be as follows

$\mathcal{F}_8 =$ $[\![$ **var** $u.2 \in$ *queue of integer* **for** $n > 1$; $idle.1, rec.1 \in$ *boolean*, $z.1 \in$ *integer*;
$idle.1, rec.1 := true, false$; $u.2 = \emptyset$ **for** $n > 1$; $C_2 \parallel \mathcal{R}$
$]\!]$: $q, s \in$ *queue of integer*

With the variable partitioning

$$\mathcal{P} = \{\{q, s, p2, su\}, \{t.i\} \text{ for } i = 2, \ldots, n, \{idle.i, rec.i, z.i\} \text{ for } i = 1, \ldots, n\}$$

and with the channels $t.1$ and au, $\mathcal{F}_8$ models a processor farm, where the tasks are communicated from the master to the slaves via an unreliable channel $t.1$, see Figure 2.

We have the following theorem:

THEOREM 5 $\mathcal{F}_7 \preceq \mathcal{F}_8$.

PROOF We have that $C_0 \preceq C_1$ by Lemma 5 and $C_1 \preceq C_2$ by Lemma 5. The theorem now follows by Theorem 2 using transitivity and monotonicity of reactive refinement. □

The resulting system is a correct reactive refinement of our initial specification:

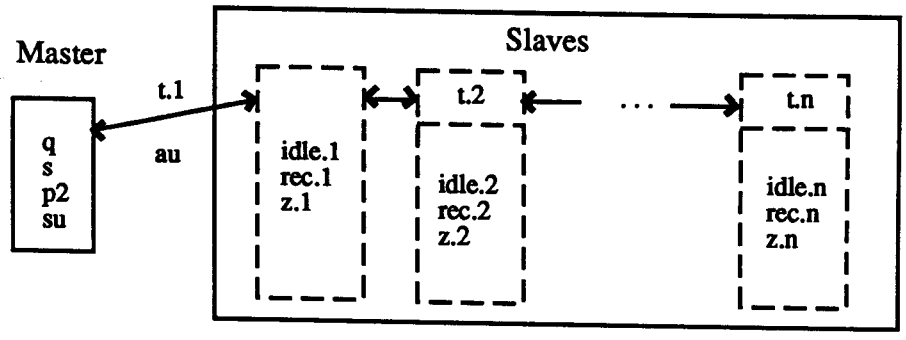

Figure 2: A processor farm with two channels.

THEOREM 6 $\mathcal{F}_0 \preceq \mathcal{F}_8$.

PROOF We first have that $\mathcal{F}_0 \preceq \mathcal{F}_1$ by Lemma 1, $\mathcal{F}_1 \preceq \mathcal{F}_2$ by Lemma 2, $\mathcal{F}_2 = \mathcal{F}_3$ as only a decomposition is carried out, $\mathcal{F}_3 \preceq \mathcal{F}_4$ by Theorem 3, $\mathcal{F}_4 = \mathcal{F}_5$ due to decomposition and $\mathcal{F}_5 \preceq \mathcal{F}_6$ by Theorem 4. Further, $\mathcal{F}_7$ is a decomposition of $\mathcal{F}_6$ and by Theorem 5 we have that $\mathcal{F}_7 \preceq \mathcal{F}_8$. Hence, $\mathcal{F}_0 \preceq \mathcal{F}_8$ by Theorem 2 using transitivity of reactive refinement. □

8 Conclusions

We have demonstrated how a communication network and a protocol for a reactive distributed system can be stepwise brought about using a combination of the action systems formalism and the refinement calculus.

We used the parallel composition operator together with hiding to restructure our action systems and the interfaces between them. Before carrying out the refinements we decomposed the system so that the target variables became local to small reactive components. In this way the refinements became manageable in size and the amount of work in the proofs was reduced.

Each step basically consisted of refining the data representation. The relation R was in each case quite easy to find: it basically counted the number of possible consecutive stuttering actions combined with an invariant that holds between the new and the old variables. A strenght of the approach is that when the relation R is given, the applied proof rule is directly mechanizable.

Acknowledgement

I am grateful to the members of the Programming methodology group at Åbo Akademi University for comments.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. of the 3rd Annual IEEE Symp. on Logic In Computer Science*, pages 165–175, Edinburgh, 1988.

[2] R. J. R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1978. Report A–1978–4.

[3] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer–Verlag, 1990.

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

[5] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 1(1):133–180, January 1990.

[6] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer–Verlag, 1990.

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

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice–Hall International, 1976.

[9] A. J. G. Hey and T. D. J. Pritchard. Parallelism in scientific programming and its efficient implementation on transputer arrays. Technical report, Concurrent Computation Group, Department of Electronics and Computer Science, University of Southampton, 1987.

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

[11] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE–10(4):325–342, July 1984.

[12] P. Møller and J. Staunstrup. Problem–heap: A paradigm for multi–processor algorithms. *Parallel Computing*, 4:63–74, 1987.

[13] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[14] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[15] K. Sere. *Stepwise Derivation of Parallel Algorithms.* PhD thesis, Department of Computer Science, Åbo Akademi University, Turku, Finland, 1990.