Formal Aspects of Computing (1995) 3: 1–000
© 1995 BCS

Superposition Refinement of Reactive Systems

R.J.R. Back[1] and K. Sere[2]

[1]Abo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland, e-mail: backrj@abo.fi;
[2]University of Kuopio, Department of Computer Science and Applied Mathematics, P.O.Box 1627, FIN-70211 Kuopio, Finland, e-mail: Kaisa.Sere@uku.fi

Keywords: Action systems; Superposition; Refinement calculus; Distributed systems

Abstract. Superposition refinement enhances an algorithm by superposing one computation mechanism onto another mechanism, in a way that preserves the behavior of the original mechanism. Superposition seems to be particularly well suited to the development of parallel and distributed programs: an originally simple sequential algorithm can be extended with mechanisms that distribute control and state information to many processes, thus permitting efficient parallel execution of the algorithm. We will in this paper show how superposition of reactive systems is expressed in the refinement calculus. We illustrate the power of this method by a case study, showing how a distributed broadcasting system is derived through a sequence of superposition refinements.

1. Introduction

A common way of constructing programs is to start from an existing program that achieves part of what is needed, and add code to this program so that additional requirements are satisfied. Often this will require that some changes are done to the original program, so that the extensions fit into it. When the changes in the original program are small, in the sense that the underlying computation is essentially unchanged, we refer to this construction method as *superpositioning.*

Superposition seems to be useful in most fields of programming, because it

Correspondence and offprint requests to: Kaisa Sere, University of Kuopio, Department of Computer Science and Applied Mathematics, P.O.Box 1627, FIN-70211 Kuopio, Finland.

permits us to construct a complicated program by a sequence of successive enhancements, each of which is reasonably small and usually encodes a single design decision. In other words, it permits us to tackle one issue at the time, rather than having to make a joint design decision and settle a number of interrelated design questions all at the same time.

Superposition as a method for program refinement has come up in a number of different contexts, e.g. in the works of Dijkstra et al. [DLM78], Back and Kurki-Suonio et al. [BaK83, KuJ89], Chandy and Misra [ChM88], Francez et al. [BoF88, FrF90], Katz [Kat93] and others.

In this paper we will study superposition of parallel and distributed programs, within the *action system* framework. This formalism was introduced by Back and Kurki-Suonio [BaK83], together with one form of superpositioning. Action systems have similarities with other event-based formalisms like UNITY of Chandy and Misra [ChM88], interacting processes of Francez [FrF90] and shared actions of Ramesh and Mehndiratta [RaM87] among others. The system behavior is in these formalisms described in terms of the events or actions which processes in the system carry out in co-operating with each other.

Action systems support the construction of parallel and distributed systems in a stepwise manner [BaK83, Ser90]. Stepwise refinement of action systems starts with a specification of the intended behavior of the system, given as a sequential statement. The goal is to construct an action system that satisfies certain criteria and fits into some predefined syntactic category [BaK83, BaS91] for which an efficient implementation on the assumed distributed architecture can be given.

Superposition refinements of action systems are done in order to increase the degree of parallelism of the program, as well as distribute control in the program. For instance, modifications can be made in order

- to distribute some shared variables among the processes in a distributed system,
- to add some information gathering mechanism to the system which replaces direct access to a shared variable,
- to detect some stable property (such as termination) of an action system, or
- to impose some communication protocol upon the processes executing an action system.

These changes are done in such a way that the original computation is not disturbed while new functionality is added to the code.

Superposing one mechanism onto another often constitutes a rather large refinement step, the correctness of which can be quite difficult to establish using only informal reasoning. Therefore, a formal treatment of the method is needed. The *refinement calculus* provides a general formal framework for carrying out program refinements and proving the correctness of each refinement step. We will here show how to describe superposition of action systems within this calculus.

Refinement calculus is a formalization of the stepwise refinement method based on the weakest precondition calculus of Dijkstra [Dij76]. It was proposed by Back [Bac80, Bac88] and has later been further elaborated by several researchers [BaW90, Mor88, Mor87].

A combination of the refinement calculus and the action system formalism have turned out to provide an uniform foundation for the derivation of *paral-*

lel algorithms by stepwise refinement [BaS90, Ser90]. In parallel algorithms the notion of correctness we want to preserve is *total correctness*.

The refinement calculus has been extended to stepwise refinement of *reactive programs* [Bac90]. In these systems the externally observable behaviour of a program must be preserved in every refinement step. Superposition refinement of reactive systems is a special case of the general method for refinement of reactive action systems of Back [Bac90], which in turn is a restricted form of *data refinement* [BaW90].

We will here give a formalization of the superposition method as a reactive refinement rule. We illustrate the power of the rule via a nontrivial program derivation: a reactive broadcasting system. The final action system is derived by three successive applications of the superposition rule. Each application superposes some important mechanism onto the action system under development.

We proceed as follows. In section 2, we give a brief overview of the basic notions of the refinement calculus, to the extent needed in this paper. In section 3, the action systems formalism is presented. In section 4, the superposition method is first described informally, and then formally within the refinement calculus. Its application to an example superposition step is described in detail in Section 5. We use the derivation of a broadcasting algorithm as a general case study for illustrating the method. Section 6 gives an overview of the whole derivation of the distributed broadcasting algorithm. We end with some concluding remarks in section 7.

Besides showing how to carry out superpositions in the refinement calculus, we will also give special attention to the way program derivations using superposition are presented. Traditionally, program derivations following the refinement paradigm show all the intermediate versions in the derivation in full. We will try to compress this information and remove redundancy, hopefully without sacrificing understandability of the derivation. We propose a tabular description of a superposition derivation which clearly identifies the program components and the successive changes carried out on these.

2. Refinement calculus

In this section we describe the foundations of the refinement calculus for reactive systems. We define the notion of correct refinement between statements and describe the method of data refinement, on which the formalization of the superposition principle depends.

2.1. Refinement relation

Specification language We use the guarded commands language of Dijkstra [Dij76], with some extensions. We have two syntactic categories, statements and actions. *Statements* S are defined by

$S ::=$	$x := e$	(*assignment statement*)
	$\mid \{Q\}$	(*assert statement*)
	$\mid S_1; \ldots; S_n$	(*sequential composition*)
	$\mid$ **if** $A_1 \; [\!] \; \ldots \; [\!] \; A_m$ **fi**	(*conditional composition*)
	$\mid$ **do** $A_1 \; [\!] \; \ldots \; [\!] \; A_m$ **od**	(*iterative composition*)
	$\mid$ **begin var** x; S **end**	(*block with local variables*)

Here $A_1, \ldots, A_m$ are actions, x is a list of variables, e is a list of expressions and Q is a predicate.

An *action* (or *guarded command*) A is of the form

$$A ::= g \rightarrow S$$

where g is a boolean expression (the *guard* of A, denoted gA) and S is a statement (the *body* of A, denoted sA).

The *assert statement* $\{Q\}$ acts as *skip* if the condition Q holds in the initial state. If the condition Q does not hold in the initial state, the effect is the same as *abort*. Thus, *skip* $= \{true\}$ and *abort* $= \{false\}$. The other statements have their usual meanings.

The *weakest preconditions* of the assignment statement, sequential, conditional and iterative composition are defined as by Dijkstra [Dij76]. The weakest precondition of the assert statement is

$$\mathrm{wp}(\{Q\}, R) = Q \wedge R.$$

The weakest precondition for the block statement is

$$\mathrm{wp}(\mathbf{begin\ var}\ x; S\ \mathbf{end}, R) = (\forall x : \mathrm{wp}(S, R)).$$

Refinement of statements A statement S is said to be *(correctly) refined* by statement S', denoted $S \leq S'$, if

$$(\forall\ Q :\ \mathrm{wp}(S, Q)\ \Rightarrow \mathrm{wp}(S', Q)).$$

This is equivalent to the condition

$$(\forall\ P, Q :\ P[S]Q\ \Rightarrow\ P[S']Q).$$

Here $P[S]Q$ stands for the total correctness of S w.r.t. precondition P and postcondition Q. In other words, refinement means that whatever total correctness criteria S satisfies, S' will also satisfy this criteria (S' can satisfy other total correctness criteria also, which S does not satisfy).

Intuitively, a statement S is refined by a statement S', if (i) whenever S is guaranteed to terminate, S' is also guaranteed to terminate, and (ii) any possible outcome of S' for some initial state is also a possible outcome of S for this same initial state. This means that a refinement may either extend the domain of termination of a statement or decrease the nondeterminism of the statement, or both.

Two statements S and S' are *refinement equivalent*, denoted $S \equiv S'$, if they refine each other. This means that they are guaranteed to terminate on the same set of initial states, and will produce the same set of possible outcomes on these initial states.

The refinement relation is reflexive and transitive. Moreover, the statement constructors considered here are monotonic w.r.t. the refinement relation.

Notation for replication In our examples we will need to use replicated structures, so we will adopt a convenient notation for these. A **for**-clause will state that the previous declaration or statement is replicated, once for each value of the index variable. If the operator between the statements is not sequential composition, then it has to be indicated explicitly. Thus, we have that

$$x_i : \tau_i\ \mathbf{for}\ i \in < 1, 2, \ldots, m > \ = \ x_1 : \tau_1; x_2 : \tau_2; \ \ldots ; x_m : \tau_m$$

$S: v_1, \ldots, v_m := v_0, \ldots, v_0 \equiv$

begin S'
var $rec.i \in bool$ **for** $i \in V$;
$rec.0 := true$
$rec.i := false$ **for** $i \in V - \{0\}$;
do
$[\!]\ \neg rec.i \rightarrow v.i := v.0;\ rec.i := true$
for $i \in V - \{0\}$
od
end

Fig. 1. Two refinement equivalent programs, S and S'.

$$\begin{array}{rcl} S_i \textbf{ for } i \in < 1, 2, \ldots, m > & = & S_1;\ S_2;\ \ldots;\ S_m \\ [\!]\ S_i \textbf{ for } i \in < 1, 2, \ldots, m > & = & S_1\ [\!]\ S_2\ [\!]\ \ldots\ [\!]\ S_m. \end{array}$$

Instead of lists, the index may range over sets when the ordering of the elements does not matter. Also, we may write just a comma for **for**, when space is scarce.

Example Let $V = \{0, 1, 2, \ldots, m\}$ be a set of indices and $v.0, v.1, v.2, \ldots, v.m$ a set of variables indexed by V. Then the two programs S and S' in Figure 1 are refinement equivalent. Both will always terminate, and will establish the same final state for a given initial state: each variable $v.i$ will have the value $v.0$. The proof that program S' has this effect is based on loop invariant

$$(\forall i : 1 \leq i \leq m : rec.i \Rightarrow v.i = v.0).$$

The local variable $rec.0$ is not really needed, but turns out to be useful in the derivations to follow.

Refinement of actions We also define weakest preconditions for actions, by

$$\text{wp}(g \rightarrow S, R) \quad = \quad g \Rightarrow \text{wp}(S, R).$$

Refinement between statements can then be extended to a notion of refinement between actions. Let A and A' be two actions. Action A is *refined* by action A', $A \leq A'$, if

$$(\forall\ Q :\ \text{wp}(A, Q) \Rightarrow \text{wp}(A', Q)).$$

LEMMA 1. Let A and A' be two actions. Then $A \leq A'$ if and only if

(i) $\{gA'\};\ sA \leq sA'$ and
(ii) $gA' \Rightarrow gA$.

In other words, action A is refined by action A' if and only if (i) whenever A' is enabled, the body of A is refined by the body of A' and (ii) A is enabled whenever A' is enabled.

2.2. Data refinement

Above we gave the general definitions for correct refinements between statements and actions. Superposition is a method where the data representation of a pro-

gram is modified. Therefore, we now present the tools for data refinement within the refinement calculus.

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

$$R \wedge \mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S', \exists x.R \wedge Q).$$

Note that $\exists x.R \wedge Q$ is a predicate on the variables x', z.

Data refinement is here given as a new relation $\leq_R$ between statements, different from the ordinary refinement relation $\leq$. However, as shown elsewhere [BaW90], this new relation can be written in terms of the refinement relation $\leq$.

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

(i) $\{gA'\}; sA \leq_R sA'$ and
(ii) $R \wedge gA' \Rightarrow gA$.

3. Action systems formalism

In this section the action systems formalism will be presented. We will describe how the formalism is used to model parallel and distributed computation and how reactive systems are represented as action systems. We will also give an initial action systems specification of our case study, the broadcasting algorithm.

3.1. Parallel programming with action systems

Action systems An *action system* $\mathcal{A}$ is a statement of the form

$$\mathcal{A} = \textbf{begin var } x := x_0; \textbf{ do } A_1 \,[\!]\, \ldots \,[\!]\, A_m \textbf{ od end} : z$$

on *state variables* $y = x \cup z$. The *global* variables z are indicated explicitly for notational convenience. 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 $x := x_0$ assigns initial values to the state variables. (The state variables referenced in an action A will be denoted by vA.)

The behavior of an action system is that of Dijkstra's guarded iteration statement [Dij76] on the state variables: the initialization statement is executed first, thereafter, as long as there are enabled actions, one action at a time is nondeterministically chosen and executed.

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

Let $\mathcal{A}$ and $\mathcal{B}$ be the two action systems

$$\begin{array}{lll} \mathcal{A} & = & \textbf{begin var } x := x_0; \textbf{ do } A_1 \,[\!]\, \cdots \,[\!]\, A_m \textbf{ od end} : z \text{ and} \\ \mathcal{B} & = & \textbf{begin var } y := y_0; \textbf{ do } B_1 \,[\!]\, \cdots \,[\!]\, B_n \textbf{ od end} : u. \end{array}$$

The *parallel composition* $\mathcal{A} \parallel \mathcal{B} : z \cup u$ of $\mathcal{A}$ and $\mathcal{B}$ is defined to be

$$\mathcal{A} \parallel \mathcal{B} = \textbf{begin var } x, y := x_0, y_0; \textbf{ do } A \;[\!]\; B \textbf{ od end} : z \cup u$$

where we assume that $x \cap y = \emptyset$. Here

$$\begin{array}{lcl} A & = & A_1 \;[\!]\; \cdots \;[\!]\; A_m \\ B & = & B_1 \;[\!]\; \cdots \;[\!]\; B_n. \end{array}$$

The parallel composition operator for action systems is the same as the union operator in UNITY [ChM88], except that we keep track of which variables are local and which global (UNITY only has global variables). Note that our actions are atomic. Hence, interleaving semantics is assumed.

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

$$\mathcal{A}' = \textbf{begin var } z_1; \mathcal{A} \textbf{ end} : z_2.$$

Hiding the variables z_1 makes them inaccessible to the actions outside $\mathcal{A}'$ in a parallel composition of action systems.

Structuring action systems Using parallel composition and hiding we can structure large action systems into smaller, reactive action systems, which communicate with each other via the shared, visible variables.

Consider an action system $\mathcal{C}$

$$\mathcal{C} = \textbf{begin var } u := u_0; \textbf{ do } C_1 \;[\!]\; \cdots \;[\!]\; C_m \textbf{ od end} : z.$$

Let $A = \{A_1, \ldots, A_k\}$ and $B = \{B_1, \ldots, B_l\}$ be a partitioning of the actions in $\mathcal{C}$ $(k + l = m)$ with

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

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

$$\mathcal{C}' = \textbf{begin var } w := w_0; \mathcal{A} \parallel \mathcal{B} \textbf{ end} : z$$

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

$$\begin{array}{lcl} \mathcal{A} & = & \textbf{begin var } x := x_0; \textbf{ do } A_1 \;[\!]\; \cdots \;[\!]\; A_k \textbf{ od end} : w, z \\ \mathcal{B} & = & \textbf{begin var } y := y_0; \textbf{ do } B_1 \;[\!]\; \cdots \;[\!]\; B_l \textbf{ od end} : w, z. \end{array}$$

The reactive components interact via the visible variables w and z.

Partitioned action systems An action system can be viewed as a parallel program, when a process network is associated with it. Basically this can be done either by (i) assigning each action to a process, or by (ii) assigning each state variable to a process. Here we concentrate on the latter method. The processes execute the actions concurrently guaranteeing that the atomicity requirement of actions is respected. (More on this topic can be found elsewhere [BaS91, Ser90].)

Let $\mathcal{P} = \{p_1, \ldots, p_k\}$ be a partitioning of the state variables y in action system $\mathcal{A}$. The tuple $(\mathcal{A}, \mathcal{P})$ is called a *partitioned action system*. We identify each partition p_i in a partitioned action system with a *process*. The variables in p_i are then the variables belonging to this process. We say that action A *involves* process p_i, if it refers to a variable in p_i.

Let pA be the set of processes involved in action A in a partitioned action system $(\mathcal{A}, \mathcal{P})$, i.e., $pA = \{p \in \mathcal{P} \mid A \text{ involves } p\}$. Two actions A and B are *independent* if $pA \cap pB = \emptyset$. An implementation may permit actions that are independent in some partitioning to be executed in parallel. As two independent actions do not have any variables in common, the result of their parallel execution is equivalent to executing the actions one after the other, in either order.

Derivation of action systems A hierarchy of partitioned action systems was defined be Back and Sere [BaS91]. For every class of action systems in this hierarchy, there is an efficient implementation of action systems onto some (centralized or distributed) machine architecture. These classes are, however, rather restricted, so the task of constructing an action system that fits some specific class can be quite hard.

A stepwise method for constructing an action system that fits some specific implementation class was put forward by Back and Sere [BaS90, Ser90]. The idea is that a more or less sequential system is transformed into an action system with the required characteristics. The action systems in the first steps do not, e.g., have to respect the process boundaries, and the network topology can be allowed to be arbitrary. In later versions, these restrictions will then be enforced, by making suitable modifications of the action system.

The derivation is done within the refinement calculus. In this paper we will develop a transformation rule, superposition refinement, that can be used within the refinement steps when reactive parallel and distributed systems are constructed in a stepwise manner.

3.2. Example: A broadcasting algorithm

Let (V, E) be a connected graph with V a finite set of nodes and E a finite set of edges on V. Let the nodes denote processes and the edges denote communication channels between the processes. Each process is assumed to know the identities of its direct neighbors. Node 0 knows additionally the identities of all the nodes in the network. Communication can only take place between nodes directly connected by an edge, but may be bidirectional.

Each node $i \in V$ has a global variable $v.i$. We are requested to design an action system that assigns (broadcasts) the value $v.0$ to each variable $v.i$, $i \in V - \{0\}$. The termination of the broadcast must be detected by node 0, after which this node initiates some other computation R. The algorithm should work for any connected graph (V, E).

Program C_0 in Figure 2 is an initial specification (an action system) of the required effect. The local variables $rec.i$ will guarantee that the value of $v.0$ is assigned to each $v.i$ only once. The local variable $rest$ guarantees that R is executed only when all the assignments have been carried out, and that it is then executed only once. We assume that the variables $rec.i$ do not appear in R. The global variables denoted by t are not used by the broadcasting algorithm, but might be used by R. We have given names to the individual actions, for ease of reference.

The broadcasting algorithm C_0 works as a reactive component in a larger system. Let therefore $\mathcal{E}$ be an action system that models the environment of C_0.

begin $\mathcal{C}_0$
var $rec.i \in bool$ **for** $i \in V$;
  $rest \in bool$;
$rec.0 := true$;
$rec.i := false$ **for** $i \in V - \{0\}$;
$rest := true$;
do
$[A.i]\ \neg rec.i \rightarrow$
  $v.i := v.0;\ rec.i := true$
for $i \in V - \{0\}$
$[B]\ (\forall i \in V.rec.i) \wedge rest \rightarrow$
  $rest := false;\ R$
od
end : $v.i \in val$ **for** $i \in V, t \in T$

Fig. 2. An action system specification $\mathcal{C}_0$, and a partitioning of this action system.

The environment $\mathcal{E}$ executes in parallel with $\mathcal{C}_0$:

$$\mathcal{C}_0 \parallel \mathcal{E}$$

and the components communicate through the visible variables $v.i$ and $.t$. The environment cannot refer to $rec.i$ and to $rest$ as they are local to $\mathcal{C}_0$. We assume that as long as the broadcasting of $v.0$ is in progress, the environment actions do not modify the visible array v. The environment can detect the termination of the broadcast via the variables t that can be used in R as well as by reading the array$v$. Therefore, the following invariant holds in $\mathcal{C}_0 \parallel \mathcal{E}$:

$$R_0 : R_{01} \wedge R_{02}$$

where

$$R_{01} : \quad (rest \Rightarrow \forall i \in V : rec.i \Rightarrow v.i = v.0)$$
$$R_{02} : \quad (\neg rest \Rightarrow (\forall i \in V : rec.i)).$$

Here R_{01} requires that as long as the broadcast goes on and a node i has received it, it must have received the value $v.0$. The invariant R_{02} states that at the end of the broadcast all the nodes have received the value $v.0$. Observe that we assumed that graph $V(V, E)$ is connected.

The reason for not choosing the statement $v.i := v.0$ **for** $i \in V;\ R$ as our initial specification is the behavior of the system as seen by the environment. In case an initial specification has certain visible behaviour, we want to preserve that behaviour throughout our refinements all the way to an implementation. In the initial specification $\mathcal{C}_0$, an environment $\mathcal{E}$ can observe the $v.i$ values getting modified one by one in some arbitary order. In an action $v.i := v.0$ **for** $i \in V;\ R$ the values change all at once in one atomic action, a property that is not realistic in a distributed environment.

We want to place the variables $rec.i$ in the same process as the variables $v.i$, $i \in V$. The variable $rest$ will be placed in the same process as the root $v.0$. This would give us the partitioning $\mathcal{P} = \{p_i | i \in V\}$ of the action system $\mathcal{C}_0$, where

$$p_0 \quad = \quad \{v.0, rec.0, rest\},$$

$$p_i = \{v.i, rec.i\}, \quad i \in V - \{0\}$$

and where the variables t are left out.

Figure 2 shows this partitioning graphically. It shows that node 0 has to communicate directly with every other node in the graph, first to communicate the value $v.0$ and then later to detect termination. This violates the requirement that communication only takes place along the edges in the graph. Thus, the partitioned action system $(\mathcal{C}_0, \mathcal{P})$ is not an acceptable solution to the problem we posed above. In Section 5 and Section 6 we will show how superposition is used to construct a refinement of action system $\mathcal{C}_0$ that does indeed satisfy our communication constraints when coupled with the partitioning $\mathcal{P}$.

4. Superposition refinement of action systems

Superposition means that new computation is added into a program while preserving the behaviour of the original computation. In terms of action systems, this means that new variables and code to modify these variables is added into an action system. Code is added by modifying existing actions and introducing new actions or both. These modifications must be done in some controlled manner, as the old computation must remain essentially unchanged.

In this section we describe one way of establishing the above requirement, the superposition principle for action systems. In this paper we consider action systems that model reactive behaviour. The principle will be first described informally and thereafter formalized as a refinement between two action systems. Superposition refinement is, as will be noticed, a form of data refinement between action systems. Therefore, our formalization will be a variant of the general method of data refinement between action systems as described by Back [Bac90].

4.1. Superposition refinement rule

Let $\mathcal{A}$ and $\mathcal{A}'$ be two action systems

$$\begin{aligned} \mathcal{A} &= \textbf{begin var } x := x_0; \textbf{ do } A_1 \,[\!]\, \ldots \,[\!]\, A_m \textbf{ od end} : z \text{ and} \\ \mathcal{A}' &= \textbf{begin var } x, x' := x_0, x_0'; \textbf{ do } A' \,[\!]\, B \textbf{ od end} : z \end{aligned}$$

where

$$\begin{aligned} A' &= A_1' \,[\!]\, \cdots \,[\!]\, A_m' \\ B &= B_1 \,[\!]\, \cdots \,[\!]\, B_n. \end{aligned}$$

The two action systems have the same global variables z. The action system $\mathcal{A}'$ has some new local variables x', in addition to the local variables x that $\mathcal{A}$ also has. For each *old action* A_i in $\mathcal{A}$ there is a corresponding *new action* A_i' in $\mathcal{A}'$. The *auxiliary actions* B_j in $\mathcal{A}'$ do not correspond to any actions in $\mathcal{A}$.

Hence, we have superposed onto $\mathcal{A}$ some computation that uses the new added components present in $\mathcal{A}'$. For this be a proper superposition, we have to check that the old computation of $\mathcal{A}$ is preserved, i.e., we have to show that the new action system $\mathcal{A}'$ is a refinement of the original action system $\mathcal{A}$. Furthermore, this refinement is inherently a data refinement, as new state variables are added into the system while the old variables have to preserve their original observable behavior.

The action system $\mathcal{A}$ is correctly data refined by $\mathcal{A}'$ using a ***data invariant*** $R(x, x', z)$ (the abstraction relation) on the state variables, denoted $\mathcal{A} \leq_R \mathcal{A}'$, if the following conditions are satisfied:

(1) *Initialization:*

(a) The initialization $x, x' := x_0, x'_0$ will establish $R(x, x', z)$.

(2) *Old actions:*

(a) Each new action A'_i has the same effect on the old variables x, z as the corresponding old action A_i when $R(x, x', z)$ holds,

(b) each new action A'_i will establish $R(x, x', z)$, and

(c) the guard of each new action A'_i implies the guard of the corresponding old action A_i, when $R(x, x', z)$ holds.

(3) *Auxiliary actions:*

(a) None of the auxiliary actions B_j has any effect on the old variables x, z when $R(x, x', z)$ holds, and

(b) each auxiliary action B_j will establish $R(x, x', z)$.

(4) *Termination of auxiliary actions:* The computation denoted by the auxiliary actions B_j terminates when $R(x, x', z)$ holds.

(5) *Exit condition:* The exit condition of the new action system

$$\neg(gA'_1 \vee \ldots \vee gA'_m \vee gB_1 \vee \ldots \vee gB_n)$$

implies the exit condition of the old action system

$$\neg(gA_1 \vee \ldots \vee gA_m)$$

when $R(x, x', z)$ holds.

Often a new action A'_i that is to replace an old action is constructed by simply strengthening the guard and adding some assignments to the new variables x', i.e., $A'_i = gA_i \wedge gC_i \rightarrow sA_i; sC_i$. In this case, we only need to check case (b) in condition (2), because the other conditions will be trivially satisfied.

Superposition in context The conditions (1) - (5) guarantee that the observable behaviour of $\mathcal{A}$ in terms of the state variables x, z is preserved in $\mathcal{A}'$. When the action system $\mathcal{A}$ occurs in a parallel composition with other action systems, however, these requirements are not sufficient. They are based on the assumption that the action system is executed in isolation, as a closed system. To take the context into account, we have to add one more condition on the superposition refinement.

Let $\mathcal{A}$ and $\mathcal{A}'$ be as above. Let $R(x, x', z)$ be some data invariant on the state variables of these action systems. Let $\mathcal{E}$ be another action system, an arbitary environment,

$$\mathcal{E} = \textbf{begin var } y := y_0; \textbf{ do } E_1 \, [\!] \ \ldots \ [\!] \, E_h \textbf{ od end} : z.$$

Then $\mathcal{A} \parallel \mathcal{E} \leq_R \mathcal{A}' \parallel \mathcal{E}$, if $\mathcal{A} \leq_R \mathcal{A}'$ and, in addition,

(6) *Non-interference:* Execution of any environment action E_k in a state where $R(x, x', z)$ holds preserves $R(x, x', z)$.

This condition guarantees that the interleaved execution of E_k actions preserves the abstraction relation R.

4.2. Formalization of the rule

The superposition method is more formally expressed in the following theorem.

THEOREM 1. (Superposition for action systems)
(a) Let

$$\begin{array}{rcl} \mathcal{A} & = & \textbf{begin var } x := x0; \textbf{ do } A_1 \; [] \; \dots \; [] \; A_m \textbf{ od end} : z \text{ and} \\ \mathcal{A}' & = & \textbf{begin var } x, x' := x_0, x'_0; \textbf{ do } A' \; [] \; B \textbf{ od end} : z \end{array}$$

be two action systems, where

$$\begin{array}{rcl} A' & = & A'_1 \; [] \; \cdots \; [] \; A'_m \\ B & = & B_1 \; [] \; \cdots \; [] \; B_n. \end{array}$$

Let $g\mathcal{A}$ be the disjunction of the guards of the A_i actions, $g\mathcal{A}'$ the disjunctions of the guards of the A'_i actions and $g\mathcal{B}$ the disjunction of the guards of the B_j actions. Let $R(x, x', z)$ be an abstraction relation on the state variables. Then $\mathcal{A} \leq_R \mathcal{A}'$, if

(1) $R(x_0, x'_0, z)$,
(2) $A_i \leq_R A'_i$, for $i = 1, \dots, m$,
(3) $true \rightarrow skip \leq_R B_j$, for $j = 1, \dots, n$,
(4) $R \Rightarrow \mathrm{wp}(\textbf{do } B_1 \; [] \; \dots \; [] \; B_n \textbf{ od}, true)$,
(5) $R \wedge g\mathcal{A} \Rightarrow (g\mathcal{A}' \vee g\mathcal{B})$.

(b) Let $\mathcal{E}$ be an action system,

$$\mathcal{E} = \textbf{begin var } y := y_0; \textbf{ do } E_1 \; [] \; \dots \; [] \; E_h \textbf{ od end} : z.$$

Then $\mathcal{A} \parallel \mathcal{E} \leq_R \mathcal{A}' \parallel \mathcal{E}$ if $\mathcal{A} \leq_R \mathcal{A}'$ and, in addition,

(6) $R \wedge \mathrm{wp}(E_k, true) \Rightarrow \mathrm{wp}(E_k, R)$, for $k = 1, \dots, h$.

The usefulness of superposition refinement for action systems comes from the following result:

THEOREM 2. Let $\mathcal{A}$ and $\mathcal{A}'$ be action systems as above. Then $\mathcal{A} \leq \mathcal{A}'$ if there exists a data invariant R such that $\mathcal{A} \leq_R \mathcal{A}'$

We will not use the superposition method on this level of formality in the sequel, but will be content with using the informal description of the method first given in the case study below. Most of the refinements we will consider are actually rather simple, with all statements being deterministic and always terminating.

4.3. Describing superpositions

We will describe a superposition as shown in Figure 3. The symbol $\bullet$ is either $+$ or empty. If an action on the right hand is preceded by a $+$, then only the additions to the corresponding left hand action are shown. We define

$$\begin{array}{rcl} \textbf{var } x + \textbf{var } z & = & \textbf{var } x; z \\ S + S' & = & S; S' \\ A + A' & = & gA \wedge gA' \rightarrow sA; sA' \\ \{R\} + \{R'\} & = & \{R \wedge R'\}. \end{array}$$

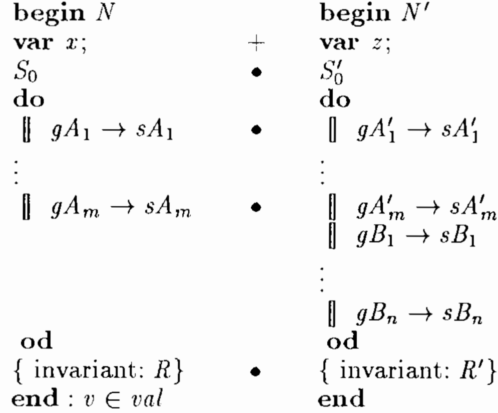


Fig. 3. Describing a superposition.

We leave the right hand side position empty if the action is unchanged in the refinement. If there is no left hand action, then the right hand action is a new auxiliary action. In all other cases, the right hand side action is to replace the corresponding left hand side action in the action system. The abstraction relation used in the superposition rule is shown after the loop.

We only permit additions to the list of local variables in superposition. We may also chain a number of successive superpositions, each new superposition providing a new column in the tabular representation of a program derivation.

5. A first superposition refinement

We will exemplify superposition refinement by showing how to change the way in which the value $v.0$ is passed among the nodes in program $\mathcal{C}_0$. In $\mathcal{C}_0$ process 0 is assumed to communicate with every other process in the network. It was, however, required that only the edges of the graph should be used for communication. We therefore add a mechanism that only uses the permitted connections to broadcast the value $v.0$. Each process, upon receiving the value $v.0$, forwards it to all other processes that it is directly connected to.

5.1. Adding a value forwarding mechanism: $\mathcal{C}_1$

The mechanism is implemented by adding a queue $q.i$ for each $i \in V$. This queue will hold the values node i has received from other nodes to which it is directly connected in the graph. The queue $q.i$ will thus contain one or more copies of the value $v.0$. When node i finds its queue non-empty, it extracts the first value from it, assigns this value to its own variable $v.i$ and then forwards it to all nodes directly connected to itself. The rest of queue $q.i$ is ignored. The set $H.i$ will hold the indices of those nodes that are directly connected to node i and that have not yet been sent the value $v.0$ from node i. Figure 4 shows the sending graphically: upon receiving the value $v.0$ each node forwards it towards the directions given by the outgoing arrows.

Fig. 4. Sending $v.0$ values.

The invariant R_0 describes the algorithm in terms of the original variables. We repeat it here for ease of reference:

$$R_0 : R_{01} \wedge R_{02}$$

where

$$\begin{array}{ll} R_{01}: & (rest \Rightarrow \forall i \in V : rec.i \Rightarrow v.i = v.0) \\ R_{02}: & (\neg rest \Rightarrow (\forall i \in V : rec.i)). \end{array}$$

The data invariant R_1 describes the way in which the new variables are to be used in the resulting action system $\mathcal{C}_1$:

$$R_1 : rest \Rightarrow R_{11} \wedge R_{12} \wedge R_{13}$$

with

$$\begin{array}{ll} R_{11}: & (\forall i \in V : (\forall j, k : q.i.j = k \Rightarrow k = v.0)) \\ R_{12}: & (\forall i \in V : H.i \subseteq E(i)) \\ R_{13}: & (\forall (k,i) \in E : i \notin H.k \Rightarrow q.i \neq <> \vee rec.i) \end{array}$$

where $E(i)$ is the set of nodes connected to node i in the graph. Here R_{11} requires that for every node i, if there is a value k in its queue $q.i$, this value must equal $v.0$. The invariant R_{12} states that for all nodes i, the set $H.i$ must contain a subset of nodes connected to node i. Finally, R_{13} requires that if node k is connected to node i, i.e., $(k,i) \in E$, and node i is not among the nodes to which k sends its $v.k$ value, i.e., in $H.k$, then either k has already sent the value and $q.i$ is nonempty or i has already received the value. The refinement is shown in Figure 5.

We use the multiple assignment statement as a convenient notation for working with lists (as queues will be represented with lists). If x is a variable of type *value* and q is a variable of type *value list*, then $x, q := q$ will assign the first element of q to x and remove it from q. Similarly, $q := q, x$ will add x as last element to q.

$\mathcal{C}_0$		$\mathcal{C}_1$
begin $\mathcal{C}_0$		**begin** $\mathcal{C}_1$
var $rec.i \in bool$ **for** $i \in V$;	+	**var** $q.i \in value\ list$ **for** $i \in V$;
$rest \in bool$;		$H.i \in index\ set$ **for** $i \in V$;
$rec.0 := true$;	+	$q.i :=<>$ **for** $i \in V$;
$rec.i := false$ **for** $i \in V - \{0\}$;		$H.i := E(i)$ **for** $i \in V$;
$rest := true$;		
do		**do**
$[A.i]\ \neg rec.i \rightarrow$		$[A.i]\ \neg rec.i \wedge q.i \neq <> \rightarrow$
$v.i := v.0;\ rec.i := true$		$v.i, q.i := q.i;\ rec.i := true$
for $i \in V - \{0\}$		**for** $i \in V - \{0\}$
$[B]\ (\forall i \in V.rec.i) \wedge rest \rightarrow$		$[B]$
$rest := false;\ R$		
		$[C.k.i]\ rec.k \wedge i \in H.k \rightarrow$
		$q.i := q.i, v.k;$
		$H.k := H.k - \{i\}$
		for $(k, i) \in E$
od		**od**
$\{R_0\}$	+	$\{R_1\}$
end : $v \in val, t \in T$		**end**

Fig. 5. Superposition of forwarding mechanism

5.2. Proof of correctness of superposition

Let us now show that $\mathcal{C}_0 \leq_{R_0 \wedge R_1} \mathcal{C}_1$ holds, using the superposition rule.

(1) *Initialization:* The new initialization will establish $R_0 \wedge R_1$: The fact that R_0 is established follows already from the fact that it is established in $\mathcal{C}_0$ and that the new initialization has the same effect on the old variables as the old initialization, as only assignments to new variables were added. R_{11} holds because all queues $q.i$ are initialized to empty, R_{12} holds because each $H.i$ is initialized to $E(i)$ and R_{13} holds initially, because $i \notin H.k$ holds for no $(i, k) \in E$.

(2) *Old actions $A.i$ and B:*

(a) The body of each new action $A.i$ has the same effect on the old variables as the corresponding old action $A.i$ when R_1 holds. This follows from R_{11}, by which the assignment $v.i := v.0$ is equivalent to assigning to $v.i$ the first element of queue $q.i$. For the B action, the condition holds trivially, because the new B action is the same as the old B action.

(b) Each new action A'_i will preserve $R_0 \wedge R_1$: The fact that R_0 is established follows from (a) above. R_{11} is clearly preserved, as values are only removed from $q.i$. R_{12} is preserved because $H.i$ is unchanged and R_{13} is preserved, because at the same time as $q.i$ may become empty, $rec.i$ is set to true. The B action does not change any variables that are constrained by the $R_0 \wedge R_1$, so the condition holds trivially in this case.

(c) The guard of each new action A'_i implies the guard of the correspon- ding old action A_i, when R_1 holds, because the new quard has an added conjunct. For the B action, the condition holds trivially.

(3) *Auxiliary actions $C.k.i$:*

(a) None of the auxiliary actions $C.k.i$ has any effect on the old variables when R_1 holds, because these actions do not assign to any of the old variables.

(b) Each auxiliary action $C.k.i$ will preserve $R_0 \wedge R_1$: The fact that R_0 is established follows from (a) above. R_{11} is preserved, because $v.k = v.0$ due to R_{01}, so the new value added to $q.i$ is $v.0$. R_{12} is preserved, because we are only removing elements from $H.k$ in this action, and R_{13} is preserved, because $q.i$ is made non-empty by the action.

(4) *Termination of auxiliary actions $C.k.i$:* Executing only auxiliary actions in an initial state where R_1 holds will necesserily terminate. This follows from the fact that there can be only finitely many elements in all sets $H.k$ altogether, and each auxiliary action will remove one element from one of these sets.

(5) *Exit condition:* The exit condition of the new action system implies the exit condition of the old action system, whenever R_1 holds. This is really the only nontrivial proof obligation in this superposition. We will prove the counterpositive of the statement, which means that we have to show that

$$
\begin{aligned}
& (\exists i \in V : \neg rec.i) \vee ((\forall i \in V : rec.i) \wedge rest) \\
\Rightarrow \quad & \\
& (\exists i \in V : \neg rec.i \wedge q.i \neq <>) \vee ((\forall i \in V : rec.i) \wedge rest) \vee \\
& (\exists (k, i) \in E : rec.k \wedge i \in H.k).
\end{aligned}
$$

If $(\forall i \in V : rec.i) \wedge rest$ holds, then this implication holds trivially. Assume therefore that it does not hold, i.e., that $\neg rec.i$ holds for some $i \in V$. Assume that $rec.i \vee q.i =<>$ holds for every $i \in V$. As the graph is connected, there must exist a path from 0 to i. We have that $rec.0$ is true and $rec.i$ is false. Hence, on this path there must exist a node k such that $rec.k$ is true, but for successor j of node k on this path, $rec.j$ is false. By assumption, this means that $q.j =<>$. By R_{13}, this again means that j must be in $E(k)$. Hence, there does exist a pair $(k, j) \in E$ such that $rec.k$ and $j \in H.k$.

(6) *Non-interference:* It was assumed that the environment holds R_0 invariant during the broadcast, so the non-interference condition holds, too.

6. Additional superposition steps

We continue here the derivation of our broadcasting algorithm. Figure 6 shows the whole derivation in tabular form, as a sequence of successive superpositions. (In the figure, V' denotes $V - \{0\}$.) The initial version $\mathcal{C}_0$ has already been shown above, as well as the first superposition $\mathcal{C}_1$. Here we will describe two more successive superpositions of this action system, $\mathcal{C}_2$ and $\mathcal{C}_3$.

A requirement we had was that the termination of the broadcast should be detected by node 0. Hence, we must make all the nodes report to node 0 when they have received their value. This will be done in two superposition steps, first

adding a mechanism that constructs a spanning tree rooted at node 0 at the same time as the values are being forwarded, giving C_2. Then, we add a mechanism that sends acknowledgments back to the root whenever a value has been received by a node, resulting in C_3. We do not show the correctness proofs of these steps here, for brevity.

6.1. Adding a spanning tree construction: C_2

We construct a spanning tree among the nodes in the graph in the following way. Each node i considers as its father the node from where it received the value $v.0$. Variable $f.i$ holds the index of the father for node i. The queue $fq.i$ holds for each node i the indices of the nodes that have sent values to node i through queue $q.i$. In Figure 7 the thicker lines denote father connections.

We use the data invariant $R_0 \wedge R_1 \wedge R_2$ where R_2 is:

$$R_2 : rest \Rightarrow R_{21} \wedge R_{22} \wedge R_{23}$$

with

$$\begin{array}{ll} R_{21}: & (\forall i \in V - \{0\}.rec.i \Rightarrow (f.i, i) \in E) \\ R_{22}: & (\forall k \in V.\forall i \in V - \{0\}.\forall j.fq.i.j = k \Rightarrow rec.k \wedge (k, i) \in E) \\ R_{23}: & (\forall k \in V - \{0\}.rec.k \Rightarrow fpath(0, k)). \end{array}$$

Here *fpath* is the least relation that satisfies the condition

$$fpath(i, j) \;=\; f.j = i \vee fpath(i, f.j)$$

for any two nodes i, j in V. The invariant R_{21} requires that if node i has received its value, i.e., $rec.i$ holds, then its father node $f.i$ must be connected to it, i.e., $(f.i, i) \in E$. The invariant R_{22} requires that if a value k is found in the queue $fq.i$, then node k must have received its value already and furthermore, nodes k and i are connected, $(k, i) \in E$. Finally, the invriant R_{23} states that only nodes to which there is a path from node 0 will participate in the broadcast.

Checking the correctness of the superposition is in this case quite simple, as no new auxiliary actions are introduced, and the guards of the old actions are unchanged. The fact that the invariant $R_0 \wedge R_1 \wedge R_2$ is established is relatively straightforward to check.

6.2. Adding backward acknowledgements: C_3

Having added a spanning tree construction, we may use the spanning tree to forward acknowledgements towards node 0.

Each node i holds a queue $ack.i$ of received acknowledgements. When node i receives value $v.0$ it acknowledges this by placing an acknowledgement message into $ack.(f.i)$, i.e., into the acknowledgement queue of its father. Whenever its ack-queue is non-empty, node i forwards the acknowledgements to its father. Node 0 keeps track of the nodes whose acknowledgements it has not yet received, in the set VS. Figure 8 illustrates the sending of the acknowledgements that take place along the father links towards the directions denoted by the arrows.

The superposition will use the data invariant $R_0 \wedge R_1 \wedge R_2 \wedge R_3$ where R_3 is as follows:

$$R_3 : rest \Rightarrow R_{31} \wedge R_{32}$$

Fig. 6. Tabular representation of whole derivation

Fig. 7. Determining father nodes.

Fig. 8. Sending acknowledgements.

where

$$\begin{array}{ll} R_{31}: & (\forall i \in V.\forall j \in V - \{0\}.\forall k : ack.i.k = j \Rightarrow rec.j \wedge j \in V - \{0\}) \\ R_{32}: & VS = \{j \mid \neg rec.j\} \cup \{j \mid (\exists i, k : ack.i.k = j\}. \end{array}$$

Hence, if a node i founds the identity of node j in its queue of acknowledgements, node j must have received the broadcast. This is more formally expressed in the invariant R_{31}. Furthermore, according to the invariant R_{32} the set VS contains the identities of all nodes j that have not received the broadcast together with the identities of nodes whose acknowledgements are still in the acknowledgement queue of some node i.

The correctness of this superposition refinement can again be checked by our rule. This superposition is less trivial than the preceding one. The main difficulty this time is to show that the auxiliary actions terminate. This will follow from the fact that the father links established in the previous step form a tree that is rooted at node 0.

6.3. Final program

Let us finally put all the superposition steps together. In Figure 9 we show the complete action system $\mathcal{C}_3$ that results from the refinement steps we have described.

The final process network is generated with the variable partitioning $\mathcal{P} = \{p_i \mid i \in V\}$, where

$$\begin{array}{lll} p_0 & = & \{v.0, rec.0, rest, q.0, H.0, ack.0, VS\}, \\ p_i & = & \{v.i, rec.i, q.i, H.i, f.i, fq.i, ack.i\}, \quad i \in V - \{0\}. \end{array}$$

In this partitioning, all the communication takes place between nodes directly connected to each other. None of the actions involve more than two adjacent processes. Furthermore, termination is detected by node 0 as was originally requested. Therefore $(\mathcal{C}_3, \mathcal{P})$ satisfies the requirements stated previously.

The final action system $\mathcal{C}_3$ will be a correct data refinement of the initial specification $\mathcal{C}_0$ using the data invariant $R_0 \wedge R_1 \wedge R_2 \wedge R_3$, i.e., we have that

$$\mathcal{C}_0 \leq_{R_0 \wedge R_1 \wedge R_2 \wedge R_3} \mathcal{C}_3.$$

Moreover,

$$\mathcal{C}_0 \parallel \mathcal{E} \leq_{R_0 \wedge R_1 \wedge R_2 \wedge R_3} \mathcal{C}_3 \parallel \mathcal{E}$$

where the environment $\mathcal{E}$ is assumed to respect the constraints in $R_0 \wedge R_1 \wedge R_2 \wedge R_3$.

7. Conclusions

We have shown how the superposition refinement rule for action systems can be formalized within the refinement calculus for reactive systems. Due to this formalization, superposition can be applied as a program transformation rule when doing program derivation in the refinement calculus framework.

The main emphasis of this paper was to show the practical usefulness of the rule. We claim that our broadcasting example, which is far from trivial, shows that the method is of practical value. Another larger example on application of the superposition rule is given by Back and Sere [BaS91] where a compiler from action systems to occam is derived.

Elsewhere [BaS92] a formalization of superposition refinement of parallel algorithms is given. There, only the preservation of the input-output relation of the algorithm is of interest. In that case the system is considered in isolation, as a black box, and there is no environment that can observe the behaviour of the algorithm. Only the initial and final states are of importance, not the path along which the computation proceeds. Therefore also the superposition refinement rule is somewhat simpler.

The method to structure superposition derivations should also be of interest. Using a tabular representation, only the new information needs to be shown at each step.

Acknowledgements

The work reported here was partly supported by the FINSOFT III programme sponsored by the Technology Development Centre of Finland. We are grateful to Jan van de Snepscheut for a critical review of a previous version of the derivation.

References

[Bac80] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[Bac88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bac90] 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.

[BaK83] 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.

[BaS91] R. J. R. Back and K. Sere. Deriving an occam implementation of action systems. In *Proc. of the 3rd Refinement Workshop BCS FACS/IBM UK Laboratories/Oxford University*, Hursley Park, England, January 1990. *Workshops in Computing*, Springer–Verlag, 1991.

[BaS90] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 1(1):133–180, January 1990.

[BaS92] R. J. R. Back and K. Sere. Superposition refinement of parallel algorithms. In K. R. Parker, and G. A. Rose, editors, *Formal Description Techniques, IV, IFIP Transactions C-2*, pages 475–494. North–Holland, 1992.

[BaW90] 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.

[BoF88] L. Bouge and N. Francez. A compositional approach to superposition. In *Proc. of the 14th ACM Conference on Principles of Programming Languages*, pages 240–249, San Diego, California, USA, January 13–15 1988.

[ChM88] K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison–Wesley, 1988.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.

[DLM78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffen. On–the–fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21:966 – 975. 1978.

[FrF90] N. Francez and I. R. Forman. Superimposition for interacting processes. In J. C. M. Baeten and J. W. Klop, editors. *CONCUR '90 Theories of Concurrency: Unification and Extension. Proceedings*, volume 458 of *Lecture Notes in Computer Science*, pages 230–245, Amsterdam, the Netherlands, August 1990. Springer–Verlag.

[Kat93] S. M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.

[KuJ89] R. Kurki-Suonio and H.-M. Jarvinen. Action system approach to the specification and design of distributed systems. In *Proc. of the 5th International Workshop on Software Specification and Design*, pages 34–40. ACM Software Engineering Notes 14(3), May 1989.

[Mor88] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[RaM87] S. Ramesh and S. L. Mehndiratta. A methodology for developing distributed programs. *IEEE Transactions on Software Engineering*, SE–13(8):967–976, 1987.

[Ser90] K. Sere. *Stepwise Derivation of Parallel Algorithms*. PhD thesis, Department of Computer Science, Abo Akademi University, Turku, Finland, 1990.

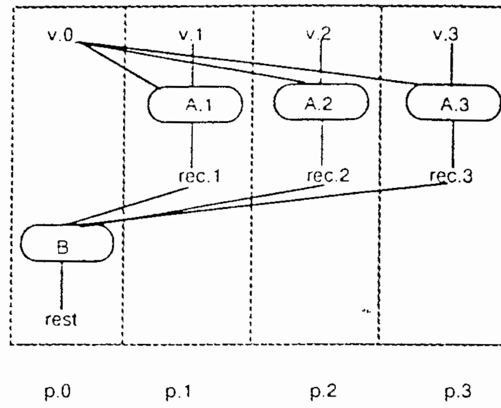


FIG. 2b

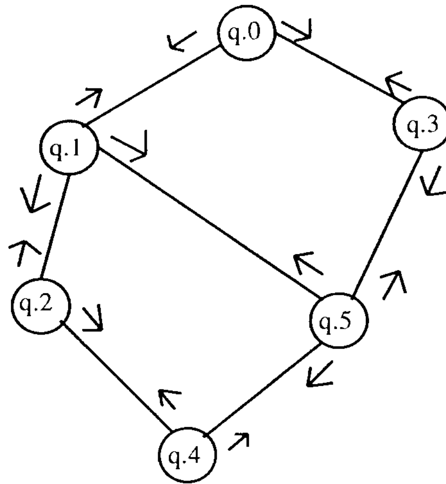


FIG 4

```
begin C1
var rec.i ∈ bool, i ∈ V;
    rest ∈ bool;

rec.0 := true;
rec.i := false i ∈ V − {0};
rest := true;

do
[A.i] ¬rec.i →
        v.i := v.0;
        rec.i := true
for i ∈ V − {0}

[B] (∀i ∈ V.rec.i) ∧ rest →
        rest := false; R

od
{Inv.1}
end : v ∈ val
```

```
  begin C2
+ var q.i ∈ value list, i ∈ V;
        H.i ∈ index set, i ∈ V;

+ q.i :=<>   for i ∈ V;
  H.i := E(i)   for ∈ V;

  do
  [A.i] ¬rec.i ∧ q.i.≠<>→
        v.i, q.i := q.i;
        rec.i := true
  for i ∈ V − {0}

  [B]

  [C.k.i] rec.k ∧ i ∈ H.k →
        q.i := q.i, v.k;
        H.k := H.k − {i}
  for (k, i) ∈ E

  od
+ {Inv.2}
  end
```

```
  begin C3
+ var f.i ∈ index, i ∈ V′;
      fq.i ∈ index list, ∈ V′;

+

  fq.i :=<>   for i ∈ V′;

  do
+ [A.i] true →
        f.i, fq.i := fq.i
  for i ∈ V − {0}

  [B]

+ [C.k.i] true →
        fq.i := fq.i, k
  for (k, i) ∈ E

  od
+ {Inv.3}
  end
```

```
  begin C4
+ var ack.i ∈ index list, i ∈ V;
      VS ∈ index set;

+ ack.i :=<>   for i ∈ V;
  VS := V − {0};

  do
+ [A.i] true →
        ack.(f.i) := ack.(f.i), i
  for i ∈ V − {0}

  [B] VS = ∅ ∧ rest →
        rest := false; R
  [C.k.i]

  [D.k] ack.k ≠<>→
        begin var a ∈ index;
        a, ack.k := ack.k;
        ack.(f.k) := ack.(f.k), a
        end
  for k ∈ V − {0}

  [E] ack.0 ≠<>→
        begin var a ∈ index;
        a, ack.0 := ack.0;
        VS := VS − {a}
        end
  od
+ {Inv.4}
  end
```

Figure 6: Tabular representation of entire derivation

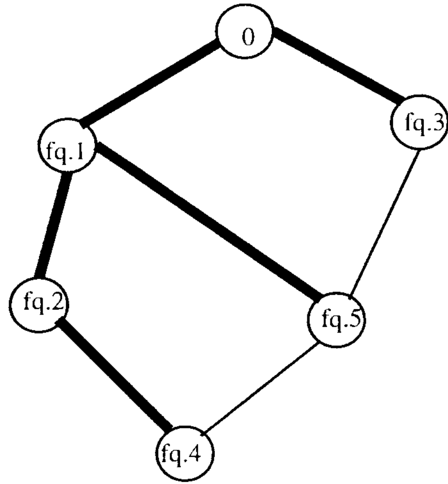
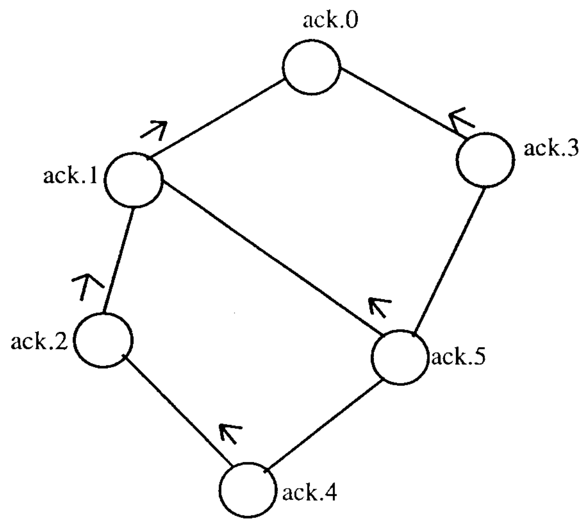


FIG 7



F16.8

```
begin C_3
var rec.i ∈ bool for i ∈ V; rest ∈ bool;
      q.i ∈ value list for i ∈ V;
      H.i ∈ index set for i ∈ V;
      f.i ∈ index for i ∈ V − {0};
      fq.i ∈ index list for i ∈ V − {0};
      ack.i ∈ index list for i ∈ V; VS ∈ index set;

rec.0 := true; rec.i := false for i ∈ V − {0}; rest := true;
q.i :=<> for i ∈ V;
H.i := E(i) for i ∈ V;
fq.i :=<> for i ∈ V − {0};
ack.i :=<> for i ∈ V; VS := V − {0};

do
[A.i] ¬rec.i ∧ q.i ≠<>→
       v.i, q.i := q.i; rec.i := true;
       f.i, fq.i := fq.i; ack.(f.i) := ack.(f.i), i
for i ∈ V − {0}

[B] VS = ∅ ∧ rest → rest := false; R

[C.k.i] rec.k ∧ i ∈ H.k →
       q.i := q.i, v.k; H.k := H.k − {i}; fq.i := fq.i, k;
for (k, i) ∈ E

[D.k] ack.k ≠<>→
       begin var a ∈ index;
       a, ack.k := ack.k; ack.(f.k) := ack.(f.k), a
       end
for k ∈ V − {0}

[E] ack.0 ≠<>→
       begin var a ∈ index;
       a, ack.0 := ack.0; VS := VS − {a}
       end
od
{R_0 ∧ R_1 ∧ R_2 ∧ R_3}
end : v.i ∈ value for i ∈ V, t ∈ T
```

Fig. 9. Resulting action system.