Superposition refinement of parallel algorithms

R.J.R. Back, K. Sere

RUU-CS-91-34
September 1991

Utrecht University

Department of Computer Science

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

Superposition refinement of parallel algorithms

R.J.R. Back, K. Sere

Technical Report RUU-CS-91-34
September 1991

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

ISSN: 0924-3275

Superposition Refinement of Parallel Algorithms *

R.J.R. Back

Åbo Akademi University
Department of Computer Science
SF–20520 Turku, Finland
e–mail: backrj@abo.fi

K. Sere

Department of Computer Science
Utrecht University
3508 TB Utrecht, The Netherlands
e–mail: kaisa@cs.ruu.nl

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 parallel algorithms is expressed in the refinement calculus. We illustrate the power of this method by a case study, showing how a distributed broadcasting algorithm 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

*The work reported here was supported by the FINSOFT III programme sponsored by the Technology Development Centre of Finland. The stay of Kaisa Sere at the Utrecht University was supported by the Dutch organisation for scientific research under project nr. NF 62–518 (Specification and Transformation Of Programs, STOP). A version of this article will appear in the Proceedings of the 4th International Conference on Formal Description Techniques (FORTE '91), Sydney, Australia, November 1991 (invited talk).

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 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. [12], Back and Kurki–Suonio et al. [5, 15], Chandy and Misra [10], Francez et al. [9, 13], Katz [14] and others.

In this paper we will study superposition of parallel programs, within the *action system* framework for parallel and distributed computations. This was introduced by Back and Kurki–Suonio in [5], together with one form of superpositioning. Action systems have similarities with other event–based formalisms like UNITY of Chandy and Misra [10], interacting processes of Francez [13] and shared actions of Ramesh and Mehndiratta [18] 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 [5, 19]. 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 [5, 6] 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

(a) distribute some shared variables among the processes in a distributed system,

(b) add some information gathering mechanism to the system which replaces direct access to a shared variable,

(c) detect some stable property (such as termination) of an action system or

(d) 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 [11]. It was first described by Back [1, 2, 3] and has later been further elaborated by several researchers [8, 16, 17].

The refinement calculus is based on the assumption that the notion of correctness we want to preserve is *total correctness*. Total correctness is an appropriate correctness notion for *parallel algorithms*, programs that differ from sequential algorithms only in that they are executed in parallel, by co–operation of many processes. They are intended to terminate, and only the final results are of interest. The refinement calculus and the action system formalism together provide an uniform foundation for the derivation of parallel algorithms by stepwise refinement [7, 19].

The refinement calculus can also be extended to stepwise refinement of reactive programs, as shown in [4]. The methods for handling superposition described here carry over to this more general framework without much changes. For brevity, we will restrict ourselves here to the original version of the refinement calculus, where total correctness is required to be preserved.

We proceed as follows. In section 2, we give a very 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 distributed 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 use a tabular description of a superposition derivation, showing the initial specification of the program and the successive changes carried out on the program components.

2 Refinement calculus

We consider the language of guarded commands of Dijkstra [11], with some extensions. We have two syntactic categories, statements and actions. *Statement* S are defined by

$$\begin{array}{lll} S ::= & x := e & (\textit{assignment statement}) \\ | & \{Q\} & (\textit{assert statement}) \\ | & S_1; \ldots; S_n & (\textit{sequential composition}) \\ | & \textbf{if } A_1 \,[]\, \ldots \,[]\, A_m \textbf{ fi} & (\textit{conditional composition}) \\ | & \textbf{do } A_1 \,[]\, \ldots \,[]\, A_m \textbf{ od} & (\textit{iterative composition}) \\ | & \textbf{begin var } x; S \textbf{ end} & (\textit{block with local variables}). \end{array}$$

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). We will say that an action is *enabled* in a certain state, if its guard is true in that state.

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 in [11]. The weakest precondition of the assert statement is $\mathrm{wp}(\{Q\}, R) = Q \wedge R$. The weakest precondition for the block statement is

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

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. Hence, if we can prove that

$$S_0 \leq S_1 \leq \ldots \leq S_{n-1} \leq S_n,$$

then

$$S_0 \leq S_n.$$

This models the successive refinement steps in a program development: S_0 is the initial high level specification statement and S_n is the final executable and efficient program that we have derived through the intermediate program versions $S_1, \ldots, S_{n-1}$. Each refinement step preserves the correctness of the previous step, so the final program must preserve the correctness of the original specification statement.

The refinement relation is monotonic w.r.t. the statement constructors. For any statement $S(T)$ that contains T as a substatement, we have that

$$T \leq T' \;\Rightarrow\; S(T) \leq S(T').$$

Notation for replication In our examples we will need to use a lot of 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

$$\begin{aligned} x_i : \tau_i \textbf{ for } i \in < 1,2,\ldots,m > &= x_1 : \tau_1; x_2 : \tau_2; \ldots; x_m : \tau_m \\ S_i \textbf{ for } i \in < 1,2,\ldots,m > &= S_1; S_2; \ldots; S_m \\ [\!] \; S_i \textbf{ for } i \in < 1,2,\ldots,m > &= S_1 \;[\!]\; S_2 \;[\!]\; \ldots \;[\!]\; S_m. \end{aligned}$$

In stead of lists, the index may range over sets when the ordering of the elements does not matter.

Example Let $V = \{0, 1, 2, \ldots, m\}$ be a set of indices, and let $v.0, v.1, v.2, \ldots, v.m$ be a set of variables indexed by V. Then the two programs 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.

$S:\ v.1, \ldots, v.m := v.0, \ldots, v.0 \ \equiv$

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

Figure 1: Two refinement equivalent programs, S and S'.

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

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

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

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

We have the following result.

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

.

3 Action systems formalism

An *action system* $\mathcal{A}$ is a (sequential) statement of the form

$$\mathcal{A} = \textbf{begin var } x;\ S_0;\ \textbf{do } A_1\ [\!]\ \ldots\ [\!]\ A_m\ \textbf{od end} : v$$

on *state variables* $y = x \cup v$. The variables v are the *global variables* and the variables x the *local variables* of $\mathcal{A}$. 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_0 assigns initial values to the state variables.

The behavior of an action system is that of Dijkstra's guarded iteration statement [11] 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.

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, their parallel execution is equivalent to executing the actions one after the other, in either order.

A hierarchy of partitioned action systems was defined in [6]. 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 in [7, 19]. 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. A set of transformation rules and methods was developed to assist in the derivation procedure. In this paper we will develop yet another transformation rule, superposition refinement, which was not considered explicitly in [7, 19].

3.1 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 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. We are looking for a *wave* algorithm, where upon receiving a value each node broadcasts it to its neighbors.

Program C in Figure 2 is an initial specification of the required effect. Program C' in Figure 2 shows a first refinement of this specification. We have simply taken the refinement of Figure 1 and, using monotonicity, replaced the assignment statement in C by its refinement S'. A second refinement C'' moves the statement R inside the

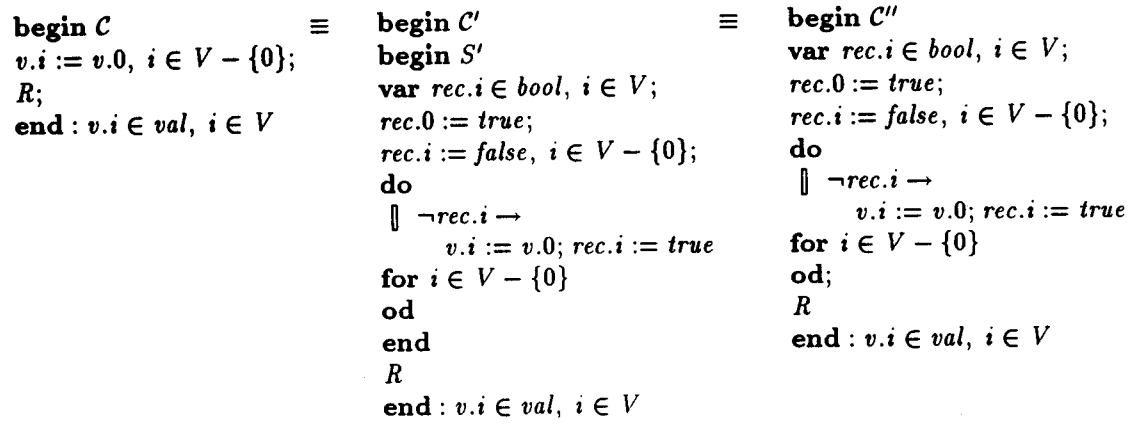

Figure 2: Initial specification C, first refinement C' and second refinement C''.

block. This refinement is correct under the assumption that the variables $rec.i$ do not appear in R, an assumption that we will make here.

Neither C nor its two refinements C' and C'' are, however, in the form of action systems, because of the trailing continuation R. An action system refinement of C'' is C_1, shown in Figure 3. Here we have made R into a single action. The auxiliary variable *rest* has been introduced in order to guarantee that R is executed only when all the assignments have been carried out, and that it is then executed only once. We have named the individual actions in this system, for ease of reference.

Using the methods described in [7, 19] the correctness of $C \leq C' \leq C'' \leq C_1$ is easily established (in fact, these are all equivalences). Hence, by transitivity, C_1 is a correct refinement of our initial specification C.

We want to place the auxiliary variables $rec.i$ in the same process as the variables $v.i, i \in V$. The auxiliary variable *rest* will be placed in the same process as the variable $v.0$. This would give us the partitioning $\mathcal{P} = \{p_i | i \in V\}$ of the action system C_1, where

$$\begin{aligned} p_0 &= \{v.0, rec.0, rest\}, \\ p_i &= \{v.i, rec.i\}, \quad i \in V - \{0\}. \end{aligned}$$

Figure 4 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 as it implicitely assumes that node 0 is directly connected to every other node. Thus, this action system is not an acceptable solution to the problem we posed above. In Section 5 we will show how superposition is used to construct a refinement of action system C_1 that does satisfy our communication constraints.

begin C_1
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** $i \in V$

Figure 3: An action system refinement C_1 of specification C

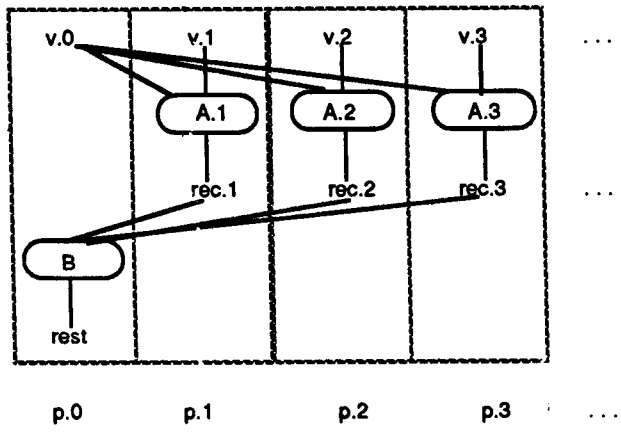

Figure 4: A process graph.

4 Superposition refinement of action systems

Let

$$\begin{array}{rcl} \mathcal{A} & = & \textbf{begin var } x;\ S_0;\ \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \ \textbf{od end} : v \text{ and} \\ \mathcal{A}' & = & \textbf{begin var } x, z;\ S_0';\ \textbf{do } A_1' \ [\!]\ \ldots\ [\!]\ A_m' \ [\!]\ B_1 \ [\!]\ \ldots\ [\!]\ B_n \ \textbf{od end} : v \end{array}$$

be two action systems with the same global variables v. The action system $\mathcal{A}'$ has some new local variables z, 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 $\mathcal{A}$.

The action system $\mathcal{A}$ is correctly refined by $\mathcal{A}'$, $\mathcal{A} \leq \mathcal{A}'$, if the following conditions are satisfied, for some assertion $I(v, x, z)$ on the state variables:

(1) *Initialization:*

  (a) The new initialization S_0' has the same effect on the old variables v, x as S_0, and

  (b) it will establish $I(v, x, z)$.

(2) *Old actions:*

  (a) The body of each new action A_i' has the same effect on the old variables v, x as the corresponding old action A_i when $I(v, x, z)$ holds,

  (b) each new action A_i' will preserve $I(v, x, z)$, and

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

(3) *Auxiliary actions:*

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

  (b) each auxiliary action B_j will preserve $I(v, x, z)$.

(4) *Termination of auxiliary actions:* Executing only auxiliary actions in an initial state where $I(v, x, z)$ holds will necessarily terminate.

(5) *Exit condition:* The exit condition of the new action system implies the exit condition of the old action system when $I(v, x, z)$ 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 z, 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. Similarly for the initialization statement: if $S_0' = S_0;\ T_0$, where T_0 only assigns values to the new variables z, then we only need to check case (b) of condition (1). The other conditions, (3) – (5), have to be checked as before. This special case corresponds to the usual notion of *syntactic superposition*, which thus simplifies the proof obligations.

4.1 Formalization of the rule

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

THEOREM 1 *(Superposition for action systems) Let*

$$\begin{aligned} \mathcal{A} &= \textbf{begin var } x;\ S_0;\ \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_m \ \textbf{od end} : v \text{ and} \\ \mathcal{A}' &= \textbf{begin var } x, z;\ S_0';\ \textbf{do } A_1' \ [\!]\ \ldots\ [\!]\ A_m' \ [\!]\ B_1 \ [\!]\ \ldots\ [\!]\ B_n \ \textbf{od end} : v. \end{aligned}$$

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. Then $\mathcal{A} \leq \mathcal{A}'$ if the following conditions hold, for some invariant $I(v, x, z)$:

(1) $S_0 \leq \textbf{begin var } z;\ S_0';\ \{I\}\ \textbf{end}.$

(2) $A_i \leq \textbf{begin var } z;\ I \rightarrow A_i';\ \{I\}\ \textbf{end}$, *for* $i = 1, \ldots, m$.

(3) *skip* $\leq \textbf{begin var } z;\ I \rightarrow B_j;\ \{I\}\ \textbf{end}$, *for* $j = 1, \ldots, n$.

(4) $I\ [\textbf{do } B_1 \ [\!]\ \ldots\ [\!]\ B_n\ \textbf{od}]\mathit{true}.$

(5) $I \wedge g\mathcal{A} \Rightarrow (g\mathcal{A}' \vee g\mathcal{B})$.

The action $I \rightarrow A_i';\ \{I\}$ in condition (2) can equivalently be written as a single action $I \wedge gA_i' \rightarrow sA_i';\ \{I\}$. A similar rewriting can be done for condition (3). In condition (3), *skip* denotes a *skip* action, i.e. an action of the form *true* $\rightarrow$ *skip*.

We will not use the superposition proof 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 actual refinements we will consider are actually rather simple, with all statements being deterministic and always terminating.

4.2 Describing superpositions

We will describe a superposition as shown in Figure 5. The symbol $\bullet$ is either $+$ or empty. We use the following conventions in describing a superposition:

1. If an action on the right hand is preceded by a $+$, then only the additions to the corresponding left hand action is shown. We define

$$\begin{aligned} \textbf{var } x + \textbf{var } z &= \textbf{var } x;\ z \\ S + S' &= S;\ S' \\ A + A' &= gA \wedge gA' \rightarrow sA;\ sA' \\ \{I\} + \{I'\} &= \{I \wedge I'\} \end{aligned}$$

2. We leave the right hand side position empty if the action is unchanged in the refinement.

3. If there is no left hand action, then the right hand action is a new auxiliary action.

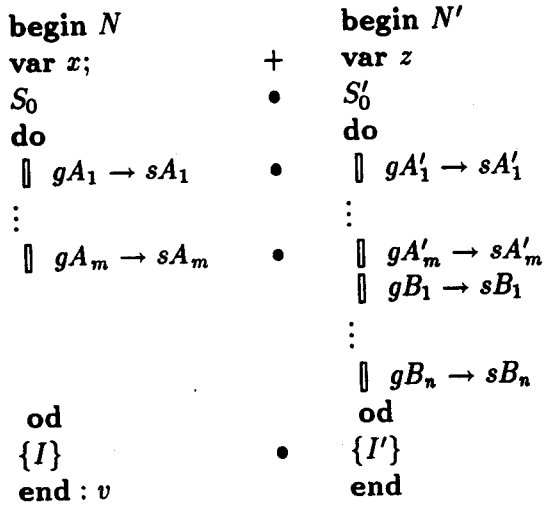

Figure 5: Describing a superposition

4. In all other cases, the right hand side action is to replace the corresponding left hand side action in the action system.

5. The invariant 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: C_2

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 C_1. In C_1 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.

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.

begin C_1	+	**begin** C_2
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**
$\{Inv.1\}$	+	$\{Inv.2\}$
end : $v \in val$		**end**

Figure 6: Superposition of forwarding mechanism

It is easy to check that the following is an invariant of action system C_1:

$$\begin{aligned} Inv.1: \quad & (rest \Rightarrow Inv.11 : (\forall i \in V : rec.i \Rightarrow v.i = v.0)) \\ \wedge \quad & (\neg rest \Rightarrow (\forall i \in V : rec.i)). \end{aligned}$$

The following additions to the loop invariant $Inv.1$ describes the way in which the new variables are to be used:

$$\begin{aligned} Inv.2: \quad rest \Rightarrow \quad & Inv.21 : (\forall i \in V : q.i \text{ contains only values } v.0) \\ \wedge \quad & Inv.22 : (\forall i \in V . H.i \subseteq E(i)) \\ \wedge \quad & Inv.23 : (\forall (k, i) \in E : i \notin H.k \Rightarrow q.i \neq <> \vee rec.i) \end{aligned}$$

We will use the multiple assignment statement as a convenient notation for working with lists (as queues will be represented). If x is a variable of type *value* and q is a variable of type *valuelist*, 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.

This superposition turns the action system into a wave algorithm. The refinement is shown in Figure 6. The new action system that we get is called C_2.

5.1 Proof of correctness of superposition

Let us now show that $C_1 \leq C_2$ holds, using the superposition rule. The invariant of the rule will be $Inv.1 \wedge Inv.2$.

(1) *Initialization:*

(a) The new initialization has the same effect on the old variables as the old initialization, because only assignments to the new variables $q.i$ and $H.i$ were added.

(b) The new initialization will establish $Inv.1 \wedge Inv.2$. The fact that $Inv.1$ is established follows already from the fact that $Inv.1$ is an invariant of the old action system and from (a). The fact that also $Inv.2$ is established is easily seen: $Inv.21$ holds because all queues $q.i$ are initialized to empty, $Inv.22$ holds because each $H.i$ is initialized to $E(i)$ and $Inv.23$ 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 $Inv.1 \wedge Inv.2$ holds. This follows from $Inv.21$, 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 $Inv.1 \wedge Inv.2$. The fact that $Inv.1$ is preserved again follows from (a) above. That $Inv.2$ is also preserved is easily seen: $Inv.21$ is clearly preserved, as values are only removed from $q.i$, $Inv.22$ is preserved because $H.i$ is unchanged and $Inv.23$ 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 new invariant, so the condition holds trivially in this case.

(c) The guard of each new action A_i' implies the guard of the corresponding old action A_i, when $Inv.1 \wedge Inv.2$ 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 $Inv.1 \wedge Inv.2$ holds, because these actions do not assign to any of the old variables.

(b) Each auxiliary action $C.k.i$ will preserve $Inv.1 \wedge Inv.2$. The fact that $Inv.1$ is preserved follows again from (a). $Inv.21$ is preserved, because by $Inv.11$, $v.k = v.0$, so the new value added to $q.i$ is $v.0$. $Inv.22$ is preserved, because we are only removing elements from $H.k$ in this action. Finally, $Inv.23$ 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 $Inv.1 \wedge Inv.2$ holds will necessarily 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 $Inv.1 \wedge Inv.2$ 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

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

If $(\forall i \in V : rec.i) \wedge rest$ holds, then this implication hold trivially. Assume therefore that it does not hold, i.e. that $\neg rec.i$ holds for some $i \in V$. Assume that $\forall i \in V.rec.i \vee q.i = <>$. 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 invariant $Inv.23$, 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 Additional superposition steps

We continue here the derivation of our broadcasting algorithm. Figure 7 shows the whole derivation in tabular form, as a sequence of successive superpositions. The initial version C_1 has already been derived above, as well as the first superposition C_2. Here we will describe two more successive superpositions of this action system, C_3 and C_4.

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 dynamic spanning tree rooted at node 0 at the same time as the values are being forwarded, giving C_3. Then, we add a mechanism that sends acknowledgments back to the root whenever a value has been received by a node, resulting in C_4. We do not show the correctness proofs of these steps here, for brevity.

6.1 Adding a dynamic spanning tree construction: C_3

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 addition to the previous invariants, we also maintain the invariance of

$$\begin{array}{lll} Inv.3: & rest \Rightarrow & Inv.31 : (\forall i \in V - \{0\}.rec.i \Rightarrow (f.i,i) \in E) \\ & \wedge & Inv.32 : (\forall k \in V.\forall i \in V - \{0\}.\forall j.fq.i.j = k \Rightarrow rec.k \wedge (k,i) \in E) \\ & \wedge & Inv.33 : (\forall k \in V - \{0\}.rec.k \Rightarrow fpath(0,k)) \end{array}$$

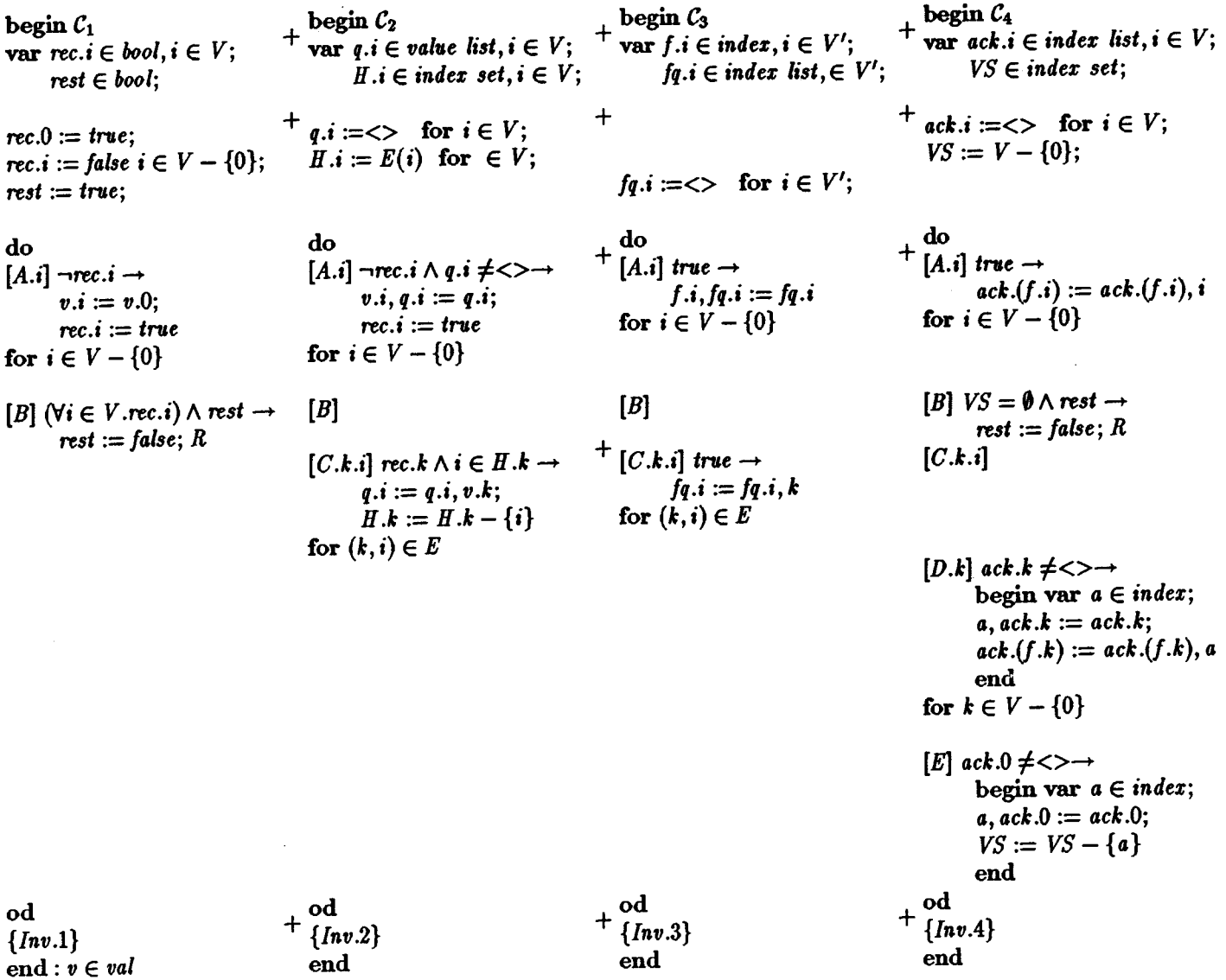

Figure 7: Entire derivation.

where *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.

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 new conjuncts of the invariant are preserved is relatively straightforward to check.

6.2 Adding backward acknowledgements: $\mathcal{C}_4$

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.

The superposition will preserve the invariance of

$$\begin{array}{lll} Inv.4: \;\; rest \Rightarrow & Inv.41: & (\forall i \in V . \forall j \in V - \{0\}. \\ & & \forall k : ack.i.k = j \Rightarrow rec.j \wedge j \in V - \{0\}) \\ \wedge & Inv.42: & VS = \{j \mid \neg rec.j\} \cup \{j \mid (\exists i, k : ack.i.k = j\}. \end{array}$$

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 necessarily terminate, if executed alone. This will follow from the fact that the father links established in the previous step form a tree that is rooted at 0.

6.3 Final program

Let us finally put all the superposition steps together. In Figure 8 we show the complete action system $\mathcal{C}_4$ that results from the refinement steps we have described. It is a wave algorithm, as required. The final process network is generated with the variable partitioning $\mathcal{P} = \{p_i \mid i \in V\}$, where

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

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.

By transitivity of refinement, the final action system $\mathcal{C}_4$ will be a correct refinement of the initial specification $\mathcal{C}$.

University, Hursley Park, England, January 1990. C. Morgan and J. C. P. Woodcock, editors, *Workshops in Computing*, pages 3–30, Springer–Verlag, 1991.

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

[8] 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.

[9] L. Bougé 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.

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

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

[12] 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.

[13] 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.

[14] S. M. Katz. A superimposition control construct for distributed systems. Technical Report STP–286–87, MCC, August 1987.

[15] R. Kurki-Suonio and H.-M. Järvinen. 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.

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

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

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

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

```
begin C_4
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
{Inv.1 ∧ Inv.2 ∧ Inv.3 ∧ Inv.4}
end : v.i ∈ value for i ∈ V
```

Figure 8: Resulting action system.