Refining Action Systems within B-Tool

M. Waldén[1] and K. Sere[2]

[1] Åbo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland, `Marina.Walden@abo.fi`
[2] University of Kuopio, Department of Computer Science and Applied Mathematics, P.O.Box 1627, FIN-70211 Kuopio, Finland, `Kaisa.Sere@uku.fi`

Abstract. Action systems is a formalism designed for the construction of parallel and distributed systems in a stepwise manner within the refinement calculus. In this paper we show how action systems can be derived and refined within a mechanical proof tool, the B-Tool. We describe how action systems are embedded in B-Tool. Due to this embedding we can now develop parallel and distributed systems within the B-Tool. We also show how a typical and nontrivial refinement rule, the superposition refinement rule, is formalized and applied on action systems within B-Tool. A derivation towards a distributed load balancing algorithm is given as a case study.

1 Introduction

Action systems are used to construct parallel and distributed systems in a stepwise manner as described by Back et al. [2, 4]. They are often developed using a poweful program modularization and structuring method called *superposition* [7, 9, 2]. In superposition some new functionality is added to an algorithm in the form of additional variables and assignments to these while the original computation is preserved.

Stepwise refinement of action systems is formalized within the *refinement calculus* [4] based on the weakest precondition calculus of Dijkstra [6]. A formalization of superposition as a refinement step within this calculus is put forward by Back and Sere [3]. Superposing one mechanism onto another often constitutes a large refinement step with many proof obligations. An example of verifying a complex distributed algorithm using superposition refinement is given in [13]. To get more confidence in the correctness proofs we need to use some mechanical tool. In this paper we show how superposition refinement of action systems can be performed using B-Tool.

The name B-Tool will in this paper refer to the B-Method and the B-Toolkit. The B-Toolkit [11] comprises a set of tools supporting a method of software development, the B-Method [1]. This method is succesfully used in many industrial projects applying formal methods. The B-Method is founded on the set theory and relies on an extension of the weakest precondition calculus of Dijkstra [6].

In this paper we show how action systems can be embedded in the B-Tool. We compare the refinement notions of the two systems, the action systems and the B-Method, and show how the superposition refinement rule formalized on action systems can be applied within the B-Tool. Since the superposition is a method for developing parallel and distributed systems, embedding the superposition method in the B-Tool makes it possible to develop parallel and distributed systems within B-Tool.

We will first briefly describe action systems and superposition refinement in section 2. In section 3, we give an overview of the B-Tool. In section 4, we show how action systems and superposition refinement are embedded in B-Tool. Finally, B-Tool is used for developing a load balancing algorithm of Hofstee et al. [8]. The first refinement step towards a distributed algorithm is given as an example. The complete derivation is reported in [14]. The load balancing algorithm within B-Tool is described in section 5. We conclude in section 6.

2 Action Systems and Superposition Refinement

We first consider the action systems framework together with its associated refinement calculus. We only give a very brief introduction here. More on these topics and further references can be found elsewhere [2, 3, 4].

2.1 Action Systems

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

$$\mathcal{A} \overset{\text{def}}{=} |[\ \textbf{var}\ x;\ x := x_0;\ \textbf{do}\ A_1\ \| \ \ldots\ \|\ A_m\ \textbf{od}\ ]| : z$$

on *state variables* $y = x \cup z$, where the variables z are the *global* variables and x are the *local* variables. 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 initialisation statement $x := x_0$ assigns initial values to the state variables x.

Each action A_i is of the form $g_i \rightarrow S_i$ where the *guard* g_i is a boolean expression on the state variables and the *body* S_i is a statement on the state variables. We denote the guard g_i of A_i by gA_i and the body S_i by sA_i. Furthermore, we say that an action is enabled in a state when its guard evaluates to *true* in that state.

The behavior of an action system is that of Dijkstra's guarded iteration statement [6] on the state variables: the initialisation statement is executed first, thereafter, as long as there are enabled actions, one action at a time is nondeterministically chosen and executed. When all the actions are disabled the action system terminates.

If two actions are independent, i.e. they do not have any variables in common, they can be executed in parallel. Their parallel execution is then equivalent to executing the actions one after the other, in either order.

2.2 Refinement of Action Systems

The superposition method has been formalized as a program refinement rule within the refinement calculus for action systems. Let us now briefly describe this calculus.

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

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

holds. Here wp is the *weakest precondition* predicate transformer [6]. Successive data refinements are modelled as follows: If $S_0 \leq_{R_1} S_1$ and $S_1 \leq_{R_2} S_2$ then $S_0 \leq_{R_1 \wedge R_2} S_2$.

Data refinement of actions is defined in a similar way, considering that the weakest precondition [6] for an action is defined as:

$$\mathrm{wp}(A, R) \overset{\mathrm{def}}{=} gA \Rightarrow \mathrm{wp}(sA, R).$$

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

(A1) $\{gA'\};\ sA \leq_R sA'$ and
(A2) $R \wedge gA' \Rightarrow gA$.

Intuitively, (A1) means that A' has the same effect on the program variables that A has when A' is enabled and R holds and moreover, A' establishes R. The condition (A2) requires that A is enabled whenever A' is enabled provided R holds.

The superposition refinement of action systems is a special kind of data refinement and it is formally expressed as follows. Let $\mathcal{A}$ and $\mathcal{A}'$ be the two action systems:

$$\mathcal{A} \overset{\mathrm{def}}{=} |[\ \mathbf{var}\ x;\ x := x_0;\ \mathbf{do}\ A_1\ \|\ \dots\ \|\ A_m\ \mathbf{od}\ ]| : z \text{ and}$$
$$\mathcal{A}' \overset{\mathrm{def}}{=} |[\ \mathbf{var}\ x, y;\ x, y := x_0, y_0;\ \mathbf{do}\ A'_1\ \|\ \dots\ \|\ A'_m\ \|\ B_1\ \|\ \dots\ \|\ B_n\ \mathbf{od}\ ]| : z.$$

Let $g\mathcal{A}$ be the disjunction of the guards of the actions A_i, $g\mathcal{A}'$ the disjunction of the guards of the actions A'_i and $g\mathcal{B}$ the disjunction of the guards of the actions B_j. Then $\mathcal{A} \leq_R \mathcal{A}'$ using $R(x, y, z)$, if

(S1) $R(x_0, y_0, z)$,
(S2) $A_i \leq_R A'_i$, for $i = 1, \dots, m$,
(S3) $\mathit{skip} \leq_R B_j$, for $j = 1, \dots, n$,
(S4) $R \wedge \neg(g\mathcal{A}' \vee g\mathcal{B}) \Rightarrow \neg g\mathcal{A}$,
(S5) $R \Rightarrow \mathrm{wp}(\mathbf{do}\ B_1\ \|\ \dots\ \|\ B_n\ \mathbf{od}, \mathit{true})$.

Informally, an action system $\mathcal{A}$ is correctly data refined by another action system $\mathcal{A}'$ using the data invariant R when

(S1) the initialisation in $\mathcal{A}'$ establishes R,
(S2) every action A_i is data refined by the corresponding A'_i using R,
(S3) every action B_j is a data refinement of the empty statement *skip* using R,
(S4) all actions in $\mathcal{A}$ are disabled whenever all actions in $\mathcal{A}'$ are disabled when R holds, i.e. the exit condition $\neg(g\mathcal{A}' \vee g\mathcal{B})$ of $\mathcal{A}'$ implies the exit condition $\neg g\mathcal{A}$ of $\mathcal{A}$ when R holds, and
(S5) the computation denoted by the actions $B_1, \ldots, B_n$ terminates provided R holds.

3 Overview of the B-Tool

A superposition refinement step is often large. In order to give more confidence in the correctness proof of a superposition step we want to use a mechanical tool. We will study B-Tool to see how it applies to action systems and how its proof rules agree with the superposition rule. The B-Toolkit [11] comprises a set of tools, which support a method of software development called the B-Method.

The B-Method is a mathematical method which gives a model oriented approach to software construction. The method is founded on set theory and relies on an extension of Dijkstra's weakest precondition calculus [6].

A program derivation in B-Method consists of a specification, possibly a number of refinements and an implementation. The *specification* is a high level description of a program under development and it usually involves a lot of nondeterminism. By the *refinements* the specification can stepwise be transformed into an implementation. This *implementation* represents the last refinement. It can directly be translated to executable code and it may not contain any nondeterminism.

Within B-Method the specifications, the refinements and the implementations are represented as Abstract Machines consisting of a context of global constraints and of operations on state variables. For specifying the operations B-Method uses the Abstract Machine Notation (AMN), which is a generalisation of Dijkstras guarded command notation. Every statement in AMN is a form of substitution. Each generated substitution S is defined as a predicate transformer which transforms a postcondition R into the weakest precondition for S to establish R, $\text{wp}(S, R)$.

The processing of Abstract Machines begins with syntax- and type-checking. Verification conditions needed for proving the specification consistency and the correctness of refinement steps can be automatically generated within the B-Toolkit. Furthermore, these verification conditions can be automatically or interactively proved using the so called autoprover or interprover, respectively. The provers are built on a mathematical library containing a collection of mathematical laws for the underlying set-theoretic notation. The autoprover first tries to discharge the proof obligations using the mathematical library. If the standard library is not enough, the user may supplement it with new necessary rules and then with the help of the interprover discharge the rest of the proof obligations.

```
MACHINE
  $Machine\_name(p)$
CONSTRAINTS
  $P$
CONSTANTS
  $c$
PROPERTIES
  $Q$
VARIABLES
  $x$
INVARIANT
  $R$
INITIALISATION
  $T$
OPERATIONS
  $Operation\_name$ = PRE $L$ THEN $S$ END ;
  ⋮
END
```

Fig. 1. The syntactic structure of an abstract machine.

In addition to the above mentioned functions there are also facilities within B-Toolkit for generating code and documentation.

4 Embedding Superposition Refinement within B-Tool

We now show how action systems, their refinement, and the superposition refinement rule can be embedded in B-Tool.

4.1 Action Systems within B-Tool

Abstract Machine Specification We first look closer at how specifications are constructed within B-Tool. The syntactic structure of an abstract machine is given in Figure 1. An abstract specification, or abstract machine, in B-Tool is identified by a unique machine name. It can be supplied with parameters p for giving dimensional characteristics of the specification. The properties P of these parameters are given in the *constraints* clause. Furthermore, within the abstract machine we can introduce constants c which are defined in terms of the parameters and some given sets. The *properties* clause gives the definition Q of these constants.

The variables x in an abstract machine are defined in the *invariant* clause and initialised in the *initialisation* clause. The invariant R consist of a set of predicates including set-theoretical typing of each variable. The initialisation T is a substitution statement. A machine can also include variables of other machines to different extents.

An abstract machine has a number of operations. These operations are named procedures which might have parameters and/or be of resulting type. The operations are given in the form of substitutions using the AMN language. They are the interface of the machine. In Figure 1 the operation consists of a substitution S with a precondition L.

The internal consistency of the abstract specification in Figure 1 can now be proved in B-Tool by showing that the following five requirements are fulfilled:

$$\begin{array}{ll}
\text{(C1)} & \exists p.P \\
\text{(C2)} & P \Rightarrow \exists c.Q \\
\text{(C3)} & (P \wedge Q) \Rightarrow \exists x.R \\
\text{(C4)} & (P \wedge Q) \Rightarrow \mathrm{wp}(T, R) \\
\text{(C5)} & (P \wedge Q \wedge R \wedge L) \Rightarrow \mathrm{wp}(S, R).
\end{array}$$

The first three obligations are concerned with the consistency of the contextual information, i.e., the formal parameters, the constants and the variables. The fourth checks that the invariant is established initially and the fifth that each operation maintains the invariant.

Action Systems Let us now study how an action system can be embedded into this Abstract Machine specification. The AMN substitution $P \Rightarrow S$ is called the guarded substitution and is interpreted as guarding of the substitution S by the predicate P. The weakest precondition for the guarded substitution is defined as

$$\mathrm{wp}(P \Rightarrow S, R) \quad \overset{\text{def}}{=} \quad P \Rightarrow \mathrm{wp}(S, R).$$

This is, however, the same as the weakest precondition for the action $P \rightarrow S$. Hence, an action can be interpreted as a guarded substitution.

The syntax for a guarded substitution interpreted as an operation in B-Tool is:

Operation_name = PRE *true* THEN (SELECT P THEN S END) END ;

where the precondition has the value *true* and can be left out. The guarded substitution as an operation is then

SELECT P THEN S END .

Hence, we choose to represent each action in B-Tool by such an operation.

Let us now consider the following action system:

$$\mathcal{A} \overset{\text{def}}{=} |[\ \mathbf{var}\ x;\ x := x_0;\ \mathbf{do}\ A_1 \ [\!] \ A_2 \ [\!] \ A_3\ \mathbf{od} \ ]| : z$$

as well as its invariant $R(x, z)$ and its constraint $P(z)$. The invariant $R(x, z)$ describes the behaviour of the local variables x possibly in terms of the global variables z, while $P(z)$ gives the constraints of the variables z. We can now write the action system as the Abstract Machine specification *ActionSystem* as given in Figure 2. In the specification the global variables of an action system

```
$\mathcal{A} \overset{\text{def}}{=}$ |[ var $x$;
         $x := x_0$;
         do
               ▯ $A_1$
               ▯ $A_2$
               ▯ $A_3$
         od
      ]|: $z$
```

```
MACHINE
  $ActionSystem(z)$
CONSTRAINTS
  $P(z)$
VARIABLES
  $x$
INVARIANT
  $R(x, z)$
INITIALISATION
  $x := x_0$
OPERATIONS
  $A_1 =$ SELECT $gA_1$ THEN $sA_1$ END ;
  $A_2 =$ SELECT $gA_2$ THEN $sA_2$ END ;
  $A_3 =$ SELECT $gA_3$ THEN $sA_3$ END
END
```

Fig. 2. An action system and its embedding in an abstract machine.

are considered as parameters and, hence their constraints in $P(z)$ are given in the *constraint* clause. For the rest, the translation of action systems into AMN is straightforward.

A consistency proof of the specification is given using the autoprover in B-Toolkit verifying the proof obligations $(C1) - (C5)$.

4.2 Superposition Refinement within B-Tool

We will now study how the superposition rule can be interpreted within B-Tool. We begin by describing the Abstract Machine refinement in B-Tool.

Abstract Machine Refinement A refinement in B-Tool may either be a data refinement or an algorithmic refinement. Data refinement is achieved by a change of variables and the operations on them, while the algorithmic refinement allows the operations to be reformulated thereby making them more concrete without changing the state space. The refinement relation within B-Tool is transitive and monotonic.

If we have the machines N and M, where N refines M, then N and M must have identical operation signatures. This means that the corresponding operations in N and M must have identical names and, if the operations have parameters, these must also be identical. The machines N and M need, however, not contain the same variables. The machine N is produced by applying a syntactic construct, *Refinement*, to the machine M. The syntactic structure of an Abstract Machine refinement is given in Figure 3.

Although, a *Refinement* and a *Machine* resemble each other in many ways there are some differences. Firstly, a refinement has to state what it will refine, a machine or another refinement. Furthermore, the invariant R' of the refinement is an abstract relation that expresses the change of variables between the two

```
REFINEMENT
  Refinement_name
REFINES
  Machine_name
VARIABLES
  x'
INVARIANT
  R'
INITIALISATION
  T'
OPERATIONS
  Operation_name =  PRE L' THEN S' END ;
  ⋮
END
```

Fig. 3. The syntactic structure of a refinement.

constructs, i.e. the relation between the variables x and x' in a data refinement. The operations of the refinement refer only to the variables x'.

In order to prove the refinement correct, a number of proof obligations are automatically created by B-Tool. The proof obligations created for the machine in Figure 1 and its refinement in Figure 3 are given below:

$$\begin{array}{ll}
\text{(B1)} & (\exists(x,x').R \wedge R') \\
\text{(B2)} & \mathrm{wp}(T', \neg\mathrm{wp}(T, \neg R')) \\
\text{(B3)} & (\forall(x,x').(R \wedge R' \wedge L) \Rightarrow L') \\
\text{(B4)} & (\forall(x,x').(R \wedge R' \wedge L) \Rightarrow \mathrm{wp}(S', \neg\mathrm{wp}(S, \neg R'))).
\end{array}$$

The first proof obligation asserts that the new invariant R' does not contradict the previous invariant R, while the second proof obligation checks that the new initialisation T' establishes a situation where the previous initialisation T cannot fail to establish the invariant R'. The last two obligations ensure the correctness of each operation. According to them the precondition L of operation S implies the precondition L' of S' when the invariants hold, i.e. the precondition is weakened. Moreover, an operation S' establishes a situation where the old operation S cannot fail to maintain R'. Due to the construction of the invariants, obligations (B2) and (B4) will also involve type-checking.

Superposition Step within B-Tool Let us now consider a superposition refinement $\mathcal{A}'$ of the action system $\mathcal{A}$ given above. We define $\mathcal{A}'$ as:

$$\mathcal{A}' \overset{\mathrm{def}}{=} |[\ \textbf{var}\ x, y;\ x, y := x_0, y_0;\ \textbf{do}\ A_1' \parallel A_2' \parallel A_3' \parallel B_1 \parallel B_2\ \textbf{od}\ ]| : z$$

and the invariant of the refinement as $R'(x, y, z)$. This refined action system can be embedded in an Abstract Machine refinement *RefActionSystem* as shown in

```
$\mathcal{A}' \overset{\text{def}}{=}$ |[ var $x, y$;        REFINEMENT
      $x, y := x_0, y_0$;              RefActionSystem
      do                               REFINES
            ▯ $A'_1$                     ActionSystem
            ▯ $A'_2$                   VARIABLES
            ▯ $A'_3$                     $x', y$
            ▯ $B_1$                    INVARIANT
            ▯ $B_2$                      $R'(x', y, z) \wedge x' = x$
      od                               INITIALISATION
   ]|: $z$                               $x', y := x_0, y_0$
                                       OPERATIONS
                                         $A_1 =$ SELECT $gA'_1$ THEN $sA'_1$ END ;
                                         $A_2 =$ SELECT $gA'_2$ THEN $sA'_2$ END ;
                                         $A_3 =$ SELECT $gA'_3$ THEN $sA'_3$ END ;
                                         $B_1 =$ SELECT $gB_1$ THEN $sB_1$ END ;
                                         $B_2 =$ SELECT $gB_2$ THEN $sB_2$ END ;
                                         exit_cond =
                                             SELECT $\neg(gA'_1 \vee gA'_2 \vee gA'_3 \vee gB_1 \vee gB_2)$
                                                     THEN skip END
                                             END
```

Fig. 4. The refined action system and its embedding in a refinement machine.

Figure 4. The invariant R' includes the invariant R of the action system being refined.

In an action system refinement some variables x are left unchanged, these are the so called old variables. However, in B-Tool we cannot use the same variable names in the specification and the refinement. We, thus, rename the old variables to x' in the refinement and state the relationship $x' = x$ in the invariant of the refinement.

Furthermore, in B-Tool all the refinements use the same operation names, which means that all operations that will exist in the final refinement also have to exist in the first specification. Since we introduced the actions B_1, B_2 and *exit_cond* as operations in the machine refinement *RefActionSystem*, we also need to introduce corresponding operations in the machine specification *ActionSystem* as $B_i =$ BEGIN *skip* END for $i = 1, 2$, where we have skipped the precondition *true* as previously. We will return to *exit_cond* later.

We have expressed how an action system and its superposition refinement can be modelled as machines in B-Tool. Let us now consider the proof rule for superposition refinement of action systems. The five conditions (S1)-(S5) in the superposition rule are equivalent to refinement rules within B-Tool as will be shown below. We start by showing the equivalences for the first four superposition conditions. The treatment of the last condition (S5) is postponed.

(1) The condition (S1) is equivalent with the condition (B2), where

$(S1) : R'(x_0, y_0, z)$
$(B2) : \mathrm{wp}((x', y := x_0, y_0), \neg\mathrm{wp}(x := x_0, \neg(R'(x', y, z) \wedge (x' = x)))),$

because

$$\begin{aligned} &\mathrm{wp}((x', y := x_0, y_0), \neg\mathrm{wp}(x := x_0, \neg(R'(x', y, z) \wedge (x' = x)))) \\ \equiv\ &\mathrm{wp}((x', y := x_0, y_0), (R'(x', y, z) \wedge (x' = x_0))) \\ \equiv\ &(R'(x_0, y_0, z) \wedge (x_0 = x_0)) \\ \equiv\ &R'(x_0, y_0, z). \end{aligned}$$

(2) The condition (S2) corresponds to the condition $(B4[A_i])$, where

$(S2) : \quad A_i \leq_{R'} A'_i$
$(B4[A_i]) : R'(x', y, z) \wedge (x' = x) \Rightarrow$
$\qquad\qquad \mathrm{wp}(A'_i, \neg\mathrm{wp}(A_i, \neg(R'(x', y, z) \wedge (x' = x)))).$

B-Tool generates the following proof obligations from $(B4[A_i])$:
(i) $R'(x', y, z) \wedge (x' = x) \Rightarrow \mathrm{wp}(A'_i, \neg\mathrm{wp}(A_i, \neg(x' = x)))$
(ii) $R'(x', y, z) \wedge (x' = x) \Rightarrow \mathrm{wp}(A'_i, R'(x', y, z))$
(iii) $R'(x', y, z) \wedge (x' = x) \wedge gA'_i \Rightarrow gA_i$
These are, however, equivalent to the conditions (A1) and (A2) that define the data refinement between actions. The items (i) and (ii) correspond to (A1) and the item (iii) is the counterpart in B-Tool of condition (A2). Hence, (S2) corresponds to the condition $(B4[A_i])$.

(3) The condition (S3) corresponds to $(B4[B_i])$, where

$(S3) : \quad skip \leq_{R'} B_i$
$(B4[B_i]) : R'(x', y, z) \wedge (x' = x) \Rightarrow$
$\qquad\qquad \mathrm{wp}(B_i, \neg\mathrm{wp}(skip, \neg(R'(x', y, z) \wedge (x' = x)))),$

From $(B4[B_i])$ B-Tool generates the following proof obligations:
(i) $R'(x', y, z) \wedge (x' = x) \Rightarrow \mathrm{wp}(B_i, (x' = x))$
(ii) $R'(x', y, z) \wedge (x' = x) \Rightarrow \mathrm{wp}(B_i, R'(x', y, z))$
(iii) $R'(x', y, z) \wedge (x' = x) \wedge gB_i \Rightarrow true$
As above, these proof obligations correspond to the conditions $(A1)$ and $(A2)$ that are equivalent to $(S3)$ when instantiated appropriately.

(4) The condition (S4) corresponds to $(B4[exit\_cond])$, where

$(S4) : \quad R' \wedge \neg(g\mathcal{A}' \vee g\mathcal{B}) \Rightarrow \neg g\mathcal{A}$
$(B4[exit\_cond]) : R'(x', y, z) \wedge (x' = x) \Rightarrow$
$\qquad\qquad \mathrm{wp}(exit\_cond', \neg\mathrm{wp}(exit\_cond, \neg(R'(x', y, z) \wedge (x' = x)))).$

The operation *exit_cond* models the exit condition in an action system. The exit condition in the action system $\mathcal{A}'$ above is defined as $\neg(gA_1 \vee gA_2 \vee gA_3 \vee gB_1 \vee gB_2)$. The exit condition operation in the old action system $\mathcal{A}$ needs to be introduced as:

$$exit\_cond = \ \text{SELECT } \neg(gA_1 \vee gA_2 \vee gA_3) \text{ THEN } skip \text{ END}$$

From ($B4[exit\_cond]$) B-Tool generates the following proof obligations:
(i) $R'(x', y, z) \wedge (x' = x) \wedge g(exit\_cond') \Rightarrow (g(exit\_cond) \wedge (x' = x))$
(ii) $R'(x', y, z) \wedge (x' = x) \wedge g(exit\_cond') \Rightarrow (g(exit\_cond) \wedge R(x', y, z))$
(iii) $R'(x', y, z) \wedge (x' = x) \wedge g(exit\_cond') \Rightarrow g(exit\_cond)$
These are easily seen to correspond to (S4).

The proof obligations (B1) and (B3) do not correspond to any of the conditions in the superposition rule. Since the invariant R is included in the invariant R' due to the superposition refinement, and the preconditions L and L' both have the value *true*, they trivially hold for the embedded action system.

Preliminary to relating the condition (S5) to a condition in the B-Method we present some additional constructs of the method.

Abstract Machine Implementation The Abstract Machine implementation is the only machine that allows loop-constructions in the operations. Since condition (S5) refers to a loop, we need to consider the implementation with a loop-construct to create a similar condition within B-Tool.

The loop-construct consists, apart from the loop, of an initialisation, an invariant, and a variant as follows:

$$T;\ \text{WHILE } P \text{ DO } S \text{ INVARIANT } R \text{ VARIANT } E \text{ END },$$

where P and R are predicates, T and S are AMN substitutions and E is an integer expression. The proof obligation created for such a loop with postcondition Q is the following:

$$\begin{array}{l}
(T1)\ \mathrm{wp}(T, R) \\
(T2)\ R \Rightarrow E \in \mathbf{N} \\
(T3)\ (\forall l.(R \wedge P) \Rightarrow \mathrm{wp}(S, R)) \\
(T4)\ (\forall l.(R \wedge P) \Rightarrow \mathrm{wp}(n := E, \mathrm{wp}(S, E < n))) \\
(T5)\ (\forall l.(R \wedge \neg P) \Rightarrow Q).
\end{array}$$

Here l denotes the variables modified within the loop. Following the obligations the initialisation T should establish the invariant R and the variant E should be an expression yielding a natural number. Furthermore, when the guard P of the loop holds, the body S should maintain the loop invariant R and decrease the variant E. Finally, the postcondition Q should hold when the loop terminates, i.e., when P does not hold anymore.

Termination of Auxiliary Actions Let us now proceed with the condition (S5): $R \Rightarrow \mathrm{wp}(\mathbf{do}\ B_1\ [\!]\ \ldots\ [\!]\ B_n\ \mathbf{od}, \mathit{true})$. We need to find a variant such that the invariant R implies that the variant is a natural number and that the variant is decreased each time one of the actions in the loop is executed. These conditions are created as proof obligations for the WHILE-loop within B-Tool. We, thus, need to make a separate refinement step within B-Tool using a WHILE-loop to prove this condition.

In this refinement step the refined abstract specification has *true* as the invariant and *skip* as the initialisation and as the only operation. In the Abstract Machine implementation we then give a WHILE-loop with a variant E operating on some variables e. This operation generated from the action system $\mathcal{A}'$ is written as:

```
VAR $x', y, e$ IN
  $x' := x_0$; $y := y_0$; $e := e_0$;
  WHILE $(gB_1 \vee gB_2)$ DO
    IF $gB_1$ THEN $sB_1$
    ELSEIF $gB_2$ THEN $sB_2$
    END
    INVARIANT $R'(x', y, z) \wedge R''(y, z, e)$
    VARIANT $E(e)$
  END
END
```

The initialisation of the action system $\mathcal{A}'$ is the initialisation of the loop and the disjunction of the guards of the auxiliary actions is given as the WHILE-condition. Within the loop we represent the auxiliary actions with an *IF* – *ELSEIF*-substitution. Furthermore, the invariant $R'(x', y, z)$ of the Abstract Machine refinement is included in the invariant of the loop. The relation $R''(y, z, e)$ gives the definition of the variant and is also included in the invariant. A new expression $E(e)$ is created as the variant.

The condition (S5) in the superposition rule can now be translated into terms of proof obligations generated in B-Tool by:

(5) $(S5) \equiv (T2) \wedge (T4).$

The proof obligations (T1), (T3), and (T5) do not directly correspond to any condition in the superposition rule. The obligation (T1) (and (T3)) is partly proved by proving the obligation (B2) (and $(B4[B_i])$), but additionally they check that the variant establishes the invariant $R''(y, z, e)$ in the initialisation and the operations. Since the postcondition of the action loop is considered to be *true* here, proof obligation (T5) holds trivially

5 Case study: Load Balancing Algorithm

As a case study we will formalize the load balancing algorithm of Hofstee et al. [8] within action systems and B-Tool. A first refinement step towards a distributed implementation of the algorithm is used to exemplify superposition refinement with B-Tool.

5.1 Load Balancing Algorithm

We consider a connected graph (V, E), where V is a finite set of nodes and E a finite set of edges on V. Let the nodes denote processes and the edges denote communication links between the processes. Each process is assumed to know the identities of its direct neighbours and the number of tasks it posesses, i.e. the load. Communication can only take place between nodes directly connected by an edge and it can go in both directions. Even so, the graph is considered to be a rooted directed tree, where the edges are directed towards the root. This assumption forces the load balancing to concentrate most of the tasks to the leaves of the tree and make it possible for the other nodes to transfer tasks from one branch to another.

The load balancing algorithm is given as an action system $\mathcal{B}$ below.

$$
\begin{array}{lll}
\mathcal{B} \overset{\text{def}}{=} & |[& \textbf{var } \mathit{load.i} \in \mathit{int} \ \textbf{ for } i \in V; \\
 & & \mathit{load.i} := \mathit{Load.i} \ \textbf{ for } i \in V; \\
 & & \textbf{do } (\mathit{bal\_load\_ij} \ \| \ \mathit{bal\_load\_ji}) \ \textbf{ for } (i,j) \in E \textbf{ od} \\
 &]|\!: & \mathit{top} \in \mathit{int}
\end{array}
$$

Here *top* is a fixed positive number, the treshold, that states the preferable load of a process. In node i the number of tasks is denoted by *load.i* and the initial value of the load is given by the constant *Load.i*. The indices i, j denote nodes. The actions are defined as follows:

$$
\begin{array}{l}
\mathit{bal\_load\_ij} \ \overset{\text{def}}{=} \\
\quad \mathit{load.i} < \mathit{top} \wedge \mathit{load.j} \geq \mathit{top} \rightarrow \mathit{load.i}, \mathit{load.j} := \mathit{load.i} + 1, \mathit{load.j} - 1
\end{array}
$$

$$
\begin{array}{l}
\mathit{bal\_load\_ji} \ \overset{\text{def}}{=} \\
\quad \mathit{load.i} > \mathit{top} \wedge \mathit{load.j} \leq \mathit{top} \rightarrow \mathit{load.i}, \mathit{load.j} := \mathit{load.i} - 1, \mathit{load.j} + 1.
\end{array}
$$

The total load of the system is stable after initialisation. If node i does not have enough tasks, i.e. its load is less than the treshold, and its father, node j, in the tree structure has a load greater or equal to the treshold, the action *bal_load_ij* is enabled and a task can be moved from node j to node i. On the other hand, if node i has too many task and its father, node j, has a load less or equal to the treshold, a task can be sent from i to j since the action *bal_load_ji* is enabled. Following this computation pattern no process is idle forever if there is enough work to be done.

We assume that the constraint $(\mathit{top} > 0)$ holds for the global variable *top* and that the constants *Load.i* have the property $(\forall i \in V : \mathit{Load.i} \geq 0)$ in the load balancing algorithm. The following invariant then holds during the computation:

$$I_1 : (\forall i \in V : \mathit{load.i} \geq 0).$$

This is due to the fact that initially the load of a node i is assigned the value *Load.i* and during the computation the load is only decreased if it is greater or equal to top, otherwise it is increased.

At termination each node either has a load greater or equal to the treshold *top* or a load less or equal to *top*:

$$(\forall i \in V : \mathit{load.i} \geq \mathit{top}) \vee (\forall i \in V : \mathit{load.i} \leq \mathit{top}).$$

```
MACHINE
  $Load\_Bal\_1(top)$
CONSTRAINTS
  $top > 0$
CONSTANTS
  $Load1, Load2$
PROPERTIES
  $Load1 : NAT \wedge Load2 : NAT$
VARIABLES
  $load1, load2$
INVARIANT
  $load1 : NAT \wedge load2 : NAT$
INITIALISATION
  $load1 := Load1 \ || \ load2 := Load2$
OPERATIONS
  $init\_12 =$  BEGIN $skip$ END ;

  $bal\_load\_12 =$ SELECT $load1 < top \wedge load2 \geq top$ THEN
                        $load1 := load1 + 1 \ || \ load2 := load2 - 1$
                 END ;
  $init\_21 =$  BEGIN $skip$ END ;

  $bal\_load\_21 =$ SELECT $load1 > top \wedge load2 \leq top$ THEN
                        $load1 := load1 - 1 \ || \ load2 := load2 + 1$
                 END ;
  $exit\_cond =$ SELECT  $(load1 \geq top \vee load2 < top)$
                       $\wedge \ (load1 \leq top \vee load2 > top)$ THEN
                       $skip$
               END
END
```

Fig. 5. The load balancing algorithm represented in AMN.

It is now straightforward to give the action system as the Abstract Machine specification in Figure 5. There are, however, some restrictions for specifications in B-Tool. We are for example not allowed to use sequential composition in a specification. Instead we have to use parallel composition of substitutions. Since these substitutions have to refer to distinct variables, we cannot assign values to distinct elements in an array in parallel. Thus, these elements have to be considered as distinct variables and the replicator functionality is lost. In our machine we have, thus, restricted the graph to one with two nodes, node 1 and node 2. Node 2 is considered to be the root. It is, however, easy to extend the algorithm to contain more than two nodes [14]. The treshold *top* is given as a parameter and ($top > 0$) as its constraint.

The operations *init*_12 and *init*_21 are only represented as *skip*-statements. They will later be introduced as actions in the refinement. Also the exit condition is given explicitely as an operation *exit*_*cond* for verification purposes.

5.2 A Superposition Refinement Step

We will now do a first refinement step towards a distributed load balancing algorithm using superposition. We add a new variable Q for representing communication links between nodes. The link $Q.i.j$ denotes the link from node i to node j. There is a link in both directions for each edge. Thus, for the edge (i, j) in the graph we have the links $Q.i.j$ and $Q.j.i$. We consider the link as a one place buffer and since we are not concerned with what is sent over the link at this stage, we can represent the link as a boolean variable. The link $Q.i.j$ is defined to be *true* if something is sent over the link and *false* otherwise. All links are initialised to *false*.

In the refined action system we split the change of loads into two phases. First, a node chooses which neighbour to change loads with. In the second phase the change of loads takes place. A node must not commit to change loads with more than one neighbour at a time. This can be expressed as in the predicate $F(i,j)$:

$$\begin{aligned} F(i,j) : &(\forall k \in V : (i,k) \in E \vee (k,i) \in E : \neg Q.i.k \wedge \neg Q.k.i) \\ &\wedge (\forall k \in V : (j,k) \in E \vee (k,j) \in E : \neg Q.j.k \wedge \neg Q.k.j). \end{aligned}$$

The predicate F states that the nodes i and j have to be free of any commitment, i.e., all their links have to be empty.

The refined action system $\mathcal{B}'$ is given below.

$$\begin{array}{lll} \mathcal{B}' & \overset{\text{def}}{=} & |[\ \textbf{var}\ load.i \in int\ \ \textbf{for}\ i \in V; \\ & & \qquad\quad Q.i.j \in bool\ \ \textbf{for}\ i,j \in V; \\ & & \quad load.i := Load.i\ \ \textbf{for}\ i \in V; \\ & & \quad Q.i.j := false\ \ \textbf{for}\ i,j \in V (i \neq j); \\ & & \quad \textbf{do}\ (init\_ij\ \|\ bal\_load\_ij'\ \|\ init\_ji\ \|\ bal\_load\_ji')\ \ \textbf{for}\ (i,j) \in E\ \textbf{od} \\ & &]|: top \in int \end{array}$$

The new actions *init_ij* and *init_ji* describe the first phase of the change of loads, the commit, by setting the links $Q.j.i$ and $Q.i.j$, respectively, to *true* denoting that the loads are ready to be changed. Neither node i nor node j can be committed to any node for these actions to be enabled. The new actions are defined as follows.

$$init\_ij \overset{\text{def}}{=} load.i < top \wedge load.j \geq top \wedge F(i,j) \rightarrow Q.j.i := true$$

$$init\_ji \overset{\text{def}}{=} load.i > top \wedge load.j \leq top \wedge F(i,j) \rightarrow Q.i.j := true,$$

where $F(i,j)$ is given above.

In the changed actions *bal_load_ij'* and *bal_load_ji'* where the loads are actually changed the corresponding links are set to *false* again stating that the changing of loads has been completed.

$$\begin{aligned} \mathit{bal\_load\_ij}' &\overset{\text{def}}{=} \mathit{load}.i < \mathit{top} \wedge \mathit{load}.j \geq \mathit{top} \wedge Q.j.i \rightarrow \\ &\qquad Q.j.i := \mathit{false};\ \mathit{load}.i, \mathit{load}.j := \mathit{load}.i + 1, \mathit{load}.j - 1 \end{aligned}$$

$$\begin{aligned} \mathit{bal\_load\_ji}' &\overset{\text{def}}{=} \mathit{load}.i > \mathit{top} \wedge \mathit{load}.j \leq \mathit{top} \wedge Q.i.j \rightarrow \\ &\qquad Q.i.j := \mathit{false};\ \mathit{load}.i, \mathit{load}.j := \mathit{load}.i - 1, \mathit{load}.j + 1 \end{aligned}$$

The following addition to the invariant I_1 of the action system $\mathcal{B}$ defines how the variables are used in the new action system $\mathcal{B}'$:

$$I_2 \overset{\text{def}}{=} I_{21} \wedge I_{22} \wedge I_{23}$$

where

$$\begin{aligned} &I_{21} : (\forall i, j \in V : (i,j) \in E : Q.j.i \Rightarrow \mathit{load}.i < \mathit{top} \wedge \mathit{load}.j \geq \mathit{top}) \\ &I_{22} : (\forall i, j \in V : (i,j) \in E : Q.i.j \Rightarrow \mathit{load}.i > \mathit{top} \wedge \mathit{load}.j \leq \mathit{top}) \\ &I_{23} : (\forall i, j \in V : (i,j) \in E \vee (j,i) \in E : Q.i.j \Rightarrow (\neg Q.j.i \wedge \\ &\qquad (\forall k \in V : ((i,k) \in E \vee (k,i) \in E) \wedge k \neq j : \neg Q.i.k \wedge \neg Q.k.i) \wedge \\ &\qquad (\forall k \in V : ((j,k) \in E \vee (k,j) \in E) \wedge k \neq i : \neg Q.j.k \wedge \neg Q.k.j))) \end{aligned}$$

The invariant I_{21} states that if there is something on the link from node j to node i, i.e. the value of $Q.j.i$ is *true*, then node j is overloaded and node i has a load lower than the treshold. A similar reasoning holds for the invariant I_{22}. Invariant I_{23} says that if there is something on the link from node i to node j then the link in the other direction has to be empty as well as all other incoming and outgoing links of nodes i and j.

We will now write the same refinement within B-Tool. The Abstract Machine refinement is given in Figure 6. The *sees* clause, *SEES Bool_TYPE*, is needed for reading boolean values. Since the operations in the machine refinement cannot involve variables of the machine being refined, we need to introduce new variables for the loads, *lload*1 and *lload*2. They are, however, stated to be equal to the old load variables in the invariant, $(\mathit{lload}1 = \mathit{load}1) \wedge (\mathit{lload}2 = \mathit{load}2)$. The invariant I_2 given above is also included in the invariant of the refined machine. The initialisation and the operations are created in the same way here as for the machine specification. For verification purpose we introduce the operation *exit_cond*.

The termination condition for the auxiliary actions are checked with a WHILE-loop in a machine implementation. The WHILE-loop for the load balancing algorithm is given in Figure 7. The disjunction of the guards of the operations *init*_12 and *init*_21 compose the guard of the loop. These operations are also included in the IF-substitution within the loop. The relation between the link variables Q and the variables $C1$ and $C2$ of the variant are added to the invariant. These relations give the definition of the function *BTS_BOOL* used for the variant, returning one for a parameter with the value *true* and zero otherwise.

```
REFINEMENT
  $Load\_Bal\_2$
REFINES
  $Load\_Bal\_1$
SEES
  $Bool\_TYPE$
VARIABLES
  $lload1, lload2, Q12, Q21$
INVARIANT
  $lload1 : NAT \wedge lload2 : NAT \wedge Q12 : BOOL \wedge Q21 : BOOL$
  $\wedge\ (lload1 = load1) \wedge (lload2 = load2)$
  $\wedge\ (Q21 = TRUE \Rightarrow lload1 < top \wedge lload2 \geq top)$
  $\wedge\ (Q12 = TRUE \Rightarrow lload1 > top \wedge lload2 \leq top)$
  $\wedge\ (Q12 = TRUE \Rightarrow Q21 = FALSE)$
  $\wedge\ (Q21 = TRUE \Rightarrow Q12 = FALSE)$
INITIALISATION
  $(lload1, lload2 := Load1, Load2) \,||\, (Q12, Q21 := FALSE, FALSE)$
OPERATIONS
  $init\_12 =$
      SELECT $lload1 < top \wedge lload2 \geq top \wedge Q21 = FALSE \wedge Q12 = FALSE$ THEN
             $Q21 := TRUE$
      END ;
  $bal\_load\_12 =$
      SELECT $lload1 < top \wedge lload2 \geq top \wedge Q21 = TRUE$ THEN
             $Q21 := FALSE; (lload1 := lload1 + 1 \,||\, lload2 := lload2 - 1)$
      END ;
  $init\_21 =$
      SELECT $lload1 > top \wedge lload2 \leq top \wedge Q21 = FALSE \wedge Q12 = FALSE$ THEN
             $Q12 := TRUE$
      END ;
  $bal\_load\_21 =$
      SELECT $lload1 > top \wedge lload2 \leq top \wedge Q12 = TRUE$ THEN
             $Q12 := FALSE; (lload1 := lload1 - 1 \,||\, lload2 := lload2 + 1)$
      END ;
  $exit\_cond =$
      SELECT  $(lload1 \geq top \vee lload2 < top \vee Q21 = TRUE \vee Q12 = TRUE)$
              $\wedge\ (lload1 \geq top \vee lload2 < top \vee Q21 = FALSE)$
              $\wedge\ (lload1 \leq top \vee lload2 > top \vee Q21 = TRUE \vee Q12 = TRUE)$
              $\wedge\ (lload1 \leq top \vee lload2 > top \vee Q12 = FALSE)$ THEN
             $skip$
      END
END
```

Fig. 6. Refinement of the abstract machine for the load balancing algorithm.

```
WHILE ($lload1 < top \wedge lload2 \geq top \wedge Q21 = FALSE \wedge Q12 = FALSE$)
        $\vee$ ($lload1 > top \wedge lload2 \leq top \wedge Q21 = FALSE \wedge Q12 = FALSE$)
      DO

      IF ($lload1 < top \wedge lload2 \geq top \wedge Q21 = FALSE \wedge Q12 = FALSE$)
         THEN $Q21 := TRUE$; $C2 \leftarrow BTS\_BOOL(Q21)$
      ELSIF ($lload1 > top \wedge lload2 \leq top \wedge Q21 = FALSE \wedge Q12 = FALSE$)
            THEN $Q12 := TRUE$; $C1 \leftarrow BTS\_BOOL(Q12)$
      END
      INVARIANT ($lload1 : NAT \wedge lload2 : NAT$
                 $\wedge\ Q12 : BOOL \wedge Q21 : BOOL$
                 $\wedge\ C1 : NAT \wedge C2 : NAT$
                 $\wedge\ (Q12 = TRUE \Rightarrow C1 = 1)$
                 $\wedge\ (Q12 = FALSE \Rightarrow C1 = 0)$
                 $\wedge\ (Q21 = TRUE \Rightarrow C2 = 1)$
                 $\wedge\ (Q21 = FALSE \Rightarrow C2 = 0)$
                 $\wedge\ (Q21 = TRUE \Rightarrow lload1 < top \wedge lload2 \geq top)$
                 $\wedge\ (Q12 = TRUE \Rightarrow lload1 > top \wedge lload2 \leq top)$
                 $\wedge\ (Q12 = TRUE \Rightarrow Q21 = FALSE)$
                 $\wedge\ (Q21 = TRUE \Rightarrow Q12 = FALSE)$)
      VARIANT $(2 - (C2 + C1))$
END
```

Fig. 7. The auxiliary actions of the load balancing algorithm.

Verification The autoprover was able to discharge most of the proof obligations generated for the load balancing algorithm. When a proof obligation cannot be discharged using the mathematical library of rules during the autoproof session, the user may supplement the library with further rules. In our case study the following kind of proof obligations could not be discharged. For the proof that the new guard implies the old, the autoprover need to use the invariant relation $(lload1 = load1) \wedge (lload2 = load2)$. Furthermore, the proof obligations for the *exit_cond*-operation need to be transformed using logical rules to bring the obligation into an expression which can be shown to be true. This proof can easily be done by hand-waving. The autoprover would only need to be supplied with these logical rules by the user. Finally, the proof obligation stating that the variant is a natural number was left unproved. Here it is again sufficient to explicitely give the definition of the variant, already given in the invariant, as an extra rule.

The proof obligations that are not discharged by the autoprover can first be discharged by the interprover. Then running the autoprover once again these obligations can be discharged by the autoprover as well using the user supplied rules. Thus, using B-Toolkit we were able to discharge all the proof obligations created for the superposition refinement step of the load balancing algorithm. Sample outputs produced by B-Tool are included in the full version of this paper [15].

6 Conclusion

We have described how an action system is turned into an Abstract Machine specification in B-Tool. We have construeted refinements which give rise to proof obligations that correspond to the conditions of superposition refinement within the action systems framework. We can, thus, do superposition proofs within B-Tool. Since superposition proofs are used for deriving parallel and distributed systems, we can now use B-Toolkit as a tool for deriving parallel and distributed systems.

By using B-Tool we gain some extra features. B-Tool can for example assist in finding the invariant for the system as well as help to find logical errors in the system. These shortcomings of an algorithm can usually be found by studying the proof obligations that cannot be automatically proved by the B-Toolkit.

There are, however, also drawbacks of using B-Tool for deriving action systems. The substitutions allowed in the specifications and refinements are very restrictive. For example a while loop cannot be introduced until the last refinement step. Even sequential composition is not allowed in the specifications. Furthermore, extra operations need to be introduced in order to be able to prove the superposition rule within B-Tool. The more complex action system we have the more complex these extra conditions will be.

We used a load balancing algorithm as a case study to exemplify how to use B-Tool for refining action systems. Most of the proof obligations created by B-Tool could be proved automatically and the rest were easily proved in an interactive way. Here we only study the very first refinement step. The complete derivation is reported in [14] .

We have also looked at other tools that could be used for refining action systems, such as the Synthesizer Generator [12] and the Refinement Calculator [10]. However, these tools still require introduction of the superposition rules in order to be applicable for superposition refinement of action systems. Furthermore, we have specified a program derivation editor [5] for strucuring and manipulating formal program derivations.

Acknowledgements

The authors would like to thank Michael Butler for fruitful discussions about the usage of B-Tool and the implementation of action systems. We would also like to thank Emil Sekerinski for the useful discussions concerning the algorithm and Wolfgang Weck for his comments on a previous version of this manuscript. The work reported here has been carried out within the Irene-project supported by the Academy of Finland.

References

1. J.-R. Abrial. The B Method for large software specification design and coding. In *Proc. of VDM'91 Vol. 2.* Springer-Verlag, 1991.

2. 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.
3. R. J. R. Back and K. Sere. *Superposition refinement of reactive systems.* Series A–144, Reports on Computer Science and Mathematics, Åbo Akademi University, Finland, 1993. To appear in *Formal Aspects of Computing.*
4. R. J. R. Back and K. Sere. From action systems to modular systems. In *Proc. of FME'94: Industrial Benefit of Formal Methods.* LNCS 873, pp. 1 – 25, 1994.
5. M. Butler, E. Hedman, P. Nilsson, R. Ruksenas, M. Waldén and Y. Zhao. *Specification of a program derivation editor.* Series A–157, Reports on Computer Science and Mathematics, Åbo Akademi University, Finland, 1994.
6. E. W. Dijkstra. *A Discipline of Programming.* Prentice–Hall International, 1976.
7. N. Francez and I.R. Forman. Superimposition for interacting processes. In *Proc. of CONCUR '90 Theories of Concurrency: Unification and extension.* LNCS 458, pages 230–245, Amsterdam, the Netherlands, August 1990.
8. H. P. Hofstee, J. J. Lukkien and J. L. A. van de Snepscheut. *A distributed implementation of a task pool.* LNCS 574, pp. 338 – 348 , June 1991.
9. S. M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
10. T. Långbacka, R. Rukšenas and J. v. Wright. *TkWinHOL: A tool for doing window inference in HOL.* Series A–160, Reports on Computer Science and Mathematics, Åbo Akademi University, Finland, 1995.
11. D. S. Neilson and I. H. Sorensen. *The B-Technologies: A system for computer aided programming.* B-Core (UK) Ltd., 1994.
12. T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, NY,1988.
13. K. Sere. and M. Waldén *Verification of a distributed algorithm due to Chu.* Series A–156, Reports on Computer Science and Mathematics, Åbo Akademi University, Finland, 1994. Abstract appeared in *Proc. of The 13th Annual Symposium on the Principles of Distributed Computing*, Los Angeles, USA, page 391, 1994.
14. M. Waldén *Formal derivation of a distributed load balancing algorithm.* Series A–172, Reports on Computer Science and Mathematics, Åbo Akademi University, Finland, 1995.
15. M. Waldén and K. Sere. *Refining action systems within B-Tool.* Manuscript in preparation. Åbo Akademi University, Finland, 1995.