Deriving Distributed Systems using B-Method

M. Waldén
K. Sere*

Åbo Akademi University, Department of Computer Science,
Lemminkäineng. 14 A, FIN-20520 Turku, Finland

* On leave from the University of Kuopio

Turku Centre for Computer Science
TUCS Technical Report No 31
July 1996
ISBN 951-650-799-9
ISSN 1239-1891

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 tool, the B-Toolkit. We describe how action systems are embedded in the B-Method, the method supported by the B-Toolkit. Furthermore, we show how a typical and nontrivial refinement rule, the superposition refinement rule, is formalized and applied on action systems within the B-Method. In addition to providing tool support for action systems refinement we also extend the application area of B-Method to cover parallel and distributed systems. A derivation towards a distributed load balancing algorithm is given as a case study.

Keywords: tools and techniques, action systems, superposition refinement, distributed systems

TUCS Research Group

Programming Methodology Research Group

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 by Sere and Waldén [13]. We want to use a mechanical tool to facilitate the verifying of the superposition refinement and to increase the confidence in the proofs. Rather than developing a new tool dedicated to action systems we study an existing tool and embed the action systems formalism in that tool. In this paper we show how superposition refinement of action systems can be performed using 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]. We will refer to the B-Method and the B-Toolkit with the name B-Tool.

In this paper we

- show how to embed action systems in the B-Tool,
- compare the refinement notions of the action systems and the B-Method, and
- show how to apply the superposition refinement rule formalized on action systems within the B-Tool.

Since superposition is a method for developing parallel and distributed systems, the above results give us the possibility to

- stepwise derive a parallel or distributed algorithm using the B-Method.

This feature has not been observed in the original approach of the B-Method. Though, we can note that the algorithms derived are parallel and distributed in an action systems sense [4].

We will first briefly describe action systems and superposition refinement in section 2. In section 3, we give a brief overview of the B-Tool relevant to this paper. 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 elsewhere [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

$$\mathcal{A} \stackrel{\text{def}}{=} |[\ \textbf{var}\ x^*, y; x, y := x_0, y_0; \textbf{do}\ A_1\ [\!]\ \dots\ [\!]\ A_m\ \textbf{od}\ ]| : z$$

on the *state variables* x, y and z, where the variables z are the *imported global* variables, x are the *exported global* variables and y 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 $x, y := x_0, y_0$ assigns initial values to the state variables x and y.

Each action A_i is of the form $g_i \to S_i$ 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. Furthemore, 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].

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

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

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

$$\begin{array}{ll} \text{(A1)} & \{gA'\}; sA \leq_R sA' \text{ and} \\ \text{(A2)} & R \wedge gA' \Rightarrow gA. \end{array}$$

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:

$$\begin{array}{lll} \mathcal{A} & \stackrel{\mathrm{def}}{=} & |[\ \mathbf{var}\ x^*, y; x, y := x_0, y_0; \\ & & \quad \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od} \\ & &]|: u, \\ & & \\ \mathcal{A}' & \stackrel{\mathrm{def}}{=} & |[\ \mathbf{var}\ x^*, y, z; x, y, z := x_0, y_0, z_0; \\ & & \quad \mathbf{do}\ A'_1\ [\!]\ \ldots\ [\!]\ A'_m\ [\!]\ B_1\ [\!]\ \ldots\ [\!]\ B_n\ \mathbf{od} \\ & &]|: u. \end{array}$$

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 $R(x, y, z, u)$, if

$$\begin{array}{ll} \text{(S1)} & R(x_0, y_0, z_0, u), \\ \text{(S2)} & A_i \leq_R A'_i, \text{ for } i = 1, \ldots, m, \\ \text{(S3)} & skip \leq_R B_j, \text{ for } j = 1, \ldots, n, \\ \text{(S4)} & R \wedge \neg(g\mathcal{A}' \vee g\mathcal{B}) \Rightarrow \neg g\mathcal{A}, \\ \text{(S5)} & R \Rightarrow \mathrm{wp}(\mathbf{do}\ B_1\ [\!]\ \ldots\ [\!]\ B_n\ \mathbf{od}, true). \end{array}$$

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 $\neg(gA'_1 \vee \ldots \vee gA'_m \vee gB_1 \vee \ldots \vee gB_n)$ of $\mathcal{A}'$ implies the exit condition $\neg(gA_1 \vee \ldots \vee gA_m)$ of $\mathcal{A}$ when R holds, and

(S5) the computation denoted by the actions $B_1, \ldots, B_n$ terminates provided R holds.

Successive superposition refinements of action systems are now modelled as follows: If $\mathcal{A}_0 \leq_{R_1} \mathcal{A}_1$ and $\mathcal{A}_1 \leq_{R_2} \mathcal{A}_2$ then $\mathcal{A}_0 \leq_{R_1 \wedge R_2} \mathcal{A}_2$.

3 Overview of the B-Tool

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

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, $\mathrm{wp}(S, R)$.

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.

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

4.1.1 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. These parameters are of type natural numbers or sets. The properties P of the 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

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

Figure 1: The syntactic structure of an abstract machine.

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.

4.1.2 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

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

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:

$$Operation\_name = \text{ PRE } true \text{ THEN (SELECT } P \text{ THEN } S \text{ END) END ;}$$

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

$$\text{SELECT } P \text{ THEN } S \text{ END .}$$

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

Let us now consider the following action system:

$$\mathcal{A} \stackrel{\text{def}}{=} |[\ \textbf{var } x^*, y; x, y := x_0, y_0; \textbf{do } A_1 \ [\!] \ A_2 \ [\!] \ A_3 \textbf{ od } \ ]| : u$$

as well as its invariant $R(x, y, u)$. We can now write the action system as the Abstract Machine specification *ActionSystem* as given in Figure 2. The imported global variables u of the action system $\mathcal{A}$ can be considered as parameters in the specification, since they are read only variables. Their constraints, here denoted $P(u)$, are then given in the *constraint* clause. Parameters in B-Tool can, however, only be of type natural numbers or sets. This may not always be the case with imported global variables. These variables can alternatively be given as constants with their constraints in the *properties* clause. In this case they can be of any type. Whether the global variables are given as parameters or constants they will be treated correctly in the proof obligations created by the B-Tool. The first alternative is, however, preferable considering that an action system also can contain real constants. For the rest, the translation of action systems into AMN is straightforward.

A consistency proof of the specification is obtained 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.

4.2.1 Abstract Machine Refinement

A refinement in B-Tool may either be a data refinement or an algorithmic refinement. The 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

$$
\begin{array}{lll}
\mathcal{A} \overset{\text{def}}{=} & |[& \mathbf{var}\ x^*, y; \\
& & x, y := x_0, y_0; \\
& & \mathbf{do} \\
& & \quad [\!]\ gA_1 \rightarrow sA_1 \\
& & \quad [\!]\ gA_2 \rightarrow sA_2 \\
& & \quad [\!]\ gA_3 \rightarrow sA_3 \\
& & \mathbf{od} \\
&]|: u &
\end{array}
$$

```
MACHINE
  ActionSystem(u)
CONSTRAINTS
  P(u)
VARIABLES
  x, y
INVARIANT
  R(x, y, u)
INITIALISATION
  x, y := x0, y0
OPERATIONS
  A1 = SELECT gA1 THEN sA1 END ;
  A2 = SELECT gA2 THEN sA2 END ;
  A3 = SELECT gA3 THEN sA3 END
END
```

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

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 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

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

Figure 3: The syntactic structure of a refinement.

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.

4.2.2 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:

$$
\begin{array}{lll}
\mathcal{A}' & \stackrel{\text{def}}{=} & |[\ \mathbf{var}\ x^*, y, z; x, y, z := x_0, y_0, z_0; \\
 & & \quad \mathbf{do}\ A'_1\ [\!]\ A'_2\ [\!]\ A'_3\ [\!]\ B_1\ [\!]\ B_2\ \mathbf{od} \\
 & & \ ]| : u
\end{array}
$$

and the invariant of the refinement as $R'(x, y, z, u)$. This refined action system can be embedded in an Abstract Machine refinement *RefActionSystem* as shown in Figure 4. We consider the invariant R' to include the invariant R of the action system being refined.

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 specifica-

$$
\begin{array}{lll}
\mathcal{A}' \overset{\text{def}}{=} & |[& \textbf{var } x^*, y, z; \\
 & & x, y, z := x_0, y_0, z_0; \\
 & & \textbf{do} \\
 & & \quad [\!] \; gA'_1 \to sA'_1 \\
 & & \quad [\!] \; gA'_2 \to sA'_2 \\
 & & \quad [\!] \; gA'_3 \to sA'_3 \\
 & & \quad [\!] \; gB_1 \to sB_1 \\
 & & \quad [\!] \; gB_2 \to sB_2 \\
 & & \textbf{od} \\
 &]|: u &
\end{array}
$$

REFINEMENT
  RefActionSystem
REFINES
  ActionSystem
VARIABLES
  x, y, z
INVARIANT
  $R'(x, y, z, u)$
INITIALISATION
  $x, y, z := x_0, y_0, z_0$
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

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

tion *ActionSystem*. The operations B_1 and B_2 are given as

$$B_i = \text{ BEGIN } skip \text{ END } \text{ for } i = 1, 2$$

in *ActionSystem*, 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. The variables x' and y' refer to the variables x and y in the refined construct in order to show how the proof obligations are created within B-Tool.

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

$$
\begin{array}{ll}
(S1): & R'(x_0, y_0, z_0, u) \\
(B2): & \text{wp}((x', y', z := x_0, y_0, z_0), \neg\text{wp}((x, y := x_0, y_0), \\
 & \qquad \neg(R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y)))),
\end{array}
$$

because

$$\begin{array}{ll} & \mathrm{wp}((x', y', z := x_0, y_0, z_0), \neg\mathrm{wp}((x, y := x_0, y_0), \\ & \qquad \neg(R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y)))) \\ \equiv & \mathrm{wp}((x', y', z := x_0, y_0, z_0), (R'(x', y', z, u) \wedge (x' = x_0) \wedge (y' = y_0))) \\ \equiv & (R'(x_0, y_0, z_0, u) \wedge (x_0 = x_0) \wedge (y_0 = y_0)) \\ \equiv & R'(x_0, y_0, z_0, u). \end{array}$$

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

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

We have that

$$\begin{array}{ll} & \mathrm{wp}(A'_i, \neg\mathrm{wp}(A_i, \neg(R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y)))) \\ \equiv & \{ \text{ def. of wp of actions and de Morgan } \} \\ & \mathrm{wp}(A'_i, \neg(gA_i \Rightarrow \mathrm{wp}(sA_i, (\neg R'(x', y', z, u) \vee \neg((x' = x) \wedge (y' = y)))))) \\ \equiv & \{ \text{ def. of } \Rightarrow\text{, de Morgan and double negation } \} \\ & \mathrm{wp}(A'_i, (gA_i \wedge \neg\mathrm{wp}(sA_i, (\neg R'(x', y', z, u) \vee \neg((x' = x) \wedge (y' = y)))))) \\ \equiv & \{ \text{ variables in } R' \text{ not assigned in } sA_i\text{, de Morgan and double neg. } \} \\ & \mathrm{wp}(A'_i, (gA_i \wedge R'(x', y', z, u) \wedge \neg\mathrm{wp}(sA_i, \neg((x' = x) \wedge (y' = y))))) \\ \equiv & \{ \text{ idempotence of } gA_i \text{ and prop. of wp } \} \\ & \mathrm{wp}(A'_i, gA_i) \wedge \mathrm{wp}(A'_i, R'(x', y', z, u)) \\ & \qquad \wedge\ \mathrm{wp}(A'_i, (gA_i \wedge \neg\mathrm{wp}(sA_i, \neg((x' = x) \wedge (y' = y))))) \\ \equiv & \{ \text{ variables in } gA_i \text{ not assigned in } A'_i\text{, de Morgan and def. of wp} \\ & \quad \text{of actions } \} \\ & (gA'_i \Rightarrow gA_i) \wedge \mathrm{wp}(A'_i, R'(x', y', z, u)) \\ & \qquad \wedge\ \mathrm{wp}(A'_i, \neg\mathrm{wp}(A_i, \neg((x' = x) \wedge (y' = y)))). \end{array}$$

B-Tool then generates the following proof obligations from ($B4[A_i]$) :

(i) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \Rightarrow$
$\quad \mathrm{wp}(A'_i, \neg\mathrm{wp}(A_i, \neg((x' = x) \wedge (y' = y))))$

(ii) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \Rightarrow \mathrm{wp}(A'_i, R'(x', y', z, u))$

(iii) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \wedge gA'_i \Rightarrow gA_i$

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-Method of condition (A2). Hence, (S2) corresponds to the condition ($B4[A_i]$).

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

$$\begin{array}{ll} (S3): & \mathit{skip} \leq_{R'} B_i \\ (B4[B_i]): & R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \Rightarrow \\ & \quad \mathrm{wp}(B_i, \neg\mathrm{wp}(\mathit{skip}, \neg(R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y)))). \end{array}$$

From $(B4[B_i])$ B-Tool generates the following proof obligations due to a similar calculation as in (2):

(i) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \Rightarrow \mathrm{wp}(B_i, (x' = x) \wedge (y' = y))$

(ii) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \Rightarrow \mathrm{wp}(B_i, R'(x', y', z, u))$

(iii) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \wedge gB_i \Rightarrow \mathit{true}$

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 $(B4[\mathit{exit\_cond}])$, where

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

The operation *exit_cond* models the exit condition in an action system. The exit condition $\neg(g\mathcal{A}' \vee g\mathcal{B})$ in the action system $\mathcal{A}'$ is defined as the operation

$$\begin{array}{l} \mathit{exit\_cond}' = \\ \qquad \text{SELECT } \neg(gA_1' \vee gA_2' \vee gA_3' \vee gB_1 \vee gB_2) \text{ THEN } \mathit{skip} \text{ END} \end{array}$$

in the machine refinement *RefActionSystem*. The exit condition $\neg g\mathcal{A}$ in the old action system $\mathcal{A}$ then needs to be introduced as the operation:

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

in the machine specification *ActionSystem*. From $(B4[\mathit{exit\_cond}])$ B-Tool then generates the following proof obligations due to a similar calculation as in (2):

(i) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \wedge g(\mathit{exit\_cond}') \Rightarrow$
$(x' = x) \wedge (y' = y)$

(ii) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \wedge g(\mathit{exit\_cond}') \Rightarrow R'(x', y', z, u)$

(iii) $R'(x', y', z, u) \wedge (x' = x) \wedge (y' = y) \wedge g(\mathit{exit\_cond}') \Rightarrow g(\mathit{exit\_cond})$

The proof obligations (i) and (ii) are trivially true and (iii) is 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.

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

4.2.3 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-Method.

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:

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

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.

4.2.4 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, z, e, c IN
  $x := x_0; y := y_0; z := z_0; e := e_0; c := c_0;$
   WHILE $(gB_1 \vee gB_2)$ DO
     $c \leftarrow Choice(2);$
     IF $(c = 1) \wedge gB_1$ THEN sB_1
     ELSIF $(c = 2) \wedge gB_2$ THEN sB_2
     ELSIF gB_1 THEN sB_1
     ELSIF gB_2 THEN sB_2
     END
     INVARIANT $R'(x, y, z, u) \wedge R''(z, u, e, c)$
     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-ELSIF-substitution. The nondeterminism in the choice of the actions is modelled with the function *Choice*. This function yields a positive natural number which is less than or equal to the number of auxiliary actions. If all the actions are enabled, *Choice* will choose one of them. The invariant $R'(x, y, z, u)$ of the Abstract Machine refinement is included in the invariant of the loop. The relation $R''(z, u, e, c)$ 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-Method 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 obligations (T1) and (T3) are partly proved by proving the obligations (B2) and $(B4[B_i])$ respectively, but additionally they check that the variant establishes the invariant $R''(z, u, e, c)$ in the initialisation and in the operations B_i. 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.

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

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 $\mathit{load}.i$ and the initial value of the load is given by the constant $\mathit{Load}.i$. The indices i, j denote nodes. The actions are defined as follows:

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

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

The total load of the system is constant 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 ($top > 0$) holds for the global variable top and that the constants $Load.i$ have the property $(\forall i \in V : Load.i \geq 0)$ in the load balancing algorithm. The following invariant then holds during the computation:

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

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:

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

It is now straightforward to give the action system as the Abstract Machine specification in Figure 5. We should, however, note that we are not allowed to use sequential composition in a specification in B-Tool. Instead we have to use parallel composition of substitutions.

For better readability, we have restricted the graph in our machine to a graph 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. By expressing the loads as functions, we could even have replication of nodes. This solution, though, gives rise to proof obligations that cannot be automatically proved by B-Tool.

The treshold top, the imported global variable of the load balancing system, is given as a parameter and ($top > 0$) as its constraint. Giving the imported global variable as a constant is not preferable here, since action system $\mathcal{B}$ has the constants $Load.i$. 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*.

```
MACHINE
  Load_Bal_1(top)
CONSTRAINTS
  top > 0
CONSTANTS
  Load1, Load2
PROPERTIES
  Load1 : NAT ∧ Load2 : NAT
VARIABLES
  load1, load2
INVARIANT
  load1 : NAT ∧ load2 : NAT
INITIALISATION
  load1 := Load1 || load2 := Load2
OPERATIONS
  init_12 =   BEGIN skip END ;

  bal_load_12 = SELECT load1 < top ∧ load2 ≥ top THEN
                       load1 := load1 + 1 || load2 := load2 − 1
                END ;
  init_21 =   BEGIN skip END ;

  bal_load_21 = SELECT load1 > top ∧ load2 ≤ top THEN
                       load1 := load1 − 1 || load2 := load2 + 1
                END ;
  exit_cond = SELECT  ¬(load1 < top ∧ load2 ≥ top)
                      ∧ ¬(load1 > top ∧ load2 ≤ top) THEN
                     skip
              END
END
```

Figure 5: The load balancing algorithm represented in AMN.

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 $F(i, j)$:

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

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.

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

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.

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

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

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

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

$$
\begin{array}{lll}
\mathit{bal\_load\_ij}' & \stackrel{\text{def}}{=} & \mathit{load}.i < \mathit{top} \land \mathit{load}.j \geq \mathit{top} \land Q.j.i \rightarrow \\
& & \quad Q.j.i := \mathit{false}; \mathit{load}.i, \mathit{load}.j := \mathit{load}.i + 1, \mathit{load}.j - 1
\end{array}
$$

$$
\begin{array}{lll}
\mathit{bal\_load\_ji}' & \stackrel{\text{def}}{=} & \mathit{load}.i > \mathit{top} \land \mathit{load}.j \leq \mathit{top} \land Q.i.j \rightarrow \\
& & \quad Q.i.j := \mathit{false}; \mathit{load}.i, \mathit{load}.j := \mathit{load}.i - 1, \mathit{load}.j + 1
\end{array}
$$

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 $\mathcal{B}'$:

$$I_2 \stackrel{\text{def}}{=} I_{21} \land I_{22} \land I_{23}$$

where

$$
\begin{array}{ll}
I_{21}: & (\forall i,j \in V : (i,j) \in E : Q.j.i \Rightarrow \mathit{load}.i < \mathit{top} \land \mathit{load}.j \geq \mathit{top}) \\
I_{22}: & (\forall i,j \in V : (i,j) \in E : Q.i.j \Rightarrow \mathit{load}.i > \mathit{top} \land \mathit{load}.j \leq \mathit{top}) \\
I_{23}: & (\forall i,j \in V : (i,j) \in E \lor (j,i) \in E : Q.i.j \Rightarrow (\neg Q.j.i \land \\
& \quad (\forall k \in V : ((i,k) \in E \lor (k,i) \in E) \land k \neq j : \neg Q.i.k \land \neg Q.k.i) \land \\
& \quad (\forall k \in V : ((j,k) \in E \lor (k,j) \in E) \land k \neq i : \neg Q.j.k \land \neg Q.k.j)))
\end{array}
$$

The invariant I_{21} states that if there is something on the link (i,j) 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.

```
REFINEMENT
  Load_Bal_2
REFINES
  Load_Bal_1
SEES
  Bool_TYPE
VARIABLES
  load1, load2, Q12, Q21
INVARIANT
  load1 : NAT ∧ load2 : NAT ∧ Q12 : BOOL ∧ Q21 : BOOL
  ∧ (Q12 = TRUE ⇒ load1 > top ∧ load2 ≤ top)
  ∧ (Q21 = TRUE ⇒ load1 < top ∧ load2 ≥ top)
  ∧ (Q12 = TRUE ⇒ Q21 = FALSE)
  ∧ (Q21 = TRUE ⇒ Q12 = FALSE)
INITIALISATION
  (lload1, load2 := Load1, Load2) || (Q12, Q21 := FALSE, FALSE)
OPERATIONS
  init_12 =
       SELECT load1 < top ∧ load2 ≥ top ∧ Q12 = FALSE ∧ Q21 = FALSE THEN
              Q21 := TRUE
       END ;
  bal_load_12 =
      SELECT load1 < top ∧ load2 ≥ top ∧ Q21 = TRUE THEN
             Q21 := FALSE; (load1 := load1 + 1 || load2 := load2 − 1)
      END ;
  init_21 =
       SELECT load1 > top ∧ load2 ≤ top ∧ Q12 = FALSE ∧ Q21 = FALSE THEN
              Q12 := TRUE
       END ;
  bal_load_21 =
      SELECT load1 > top ∧ load2 ≤ top ∧ Q12 = TRUE THEN
             Q12 := FALSE; (load1 := load1 − 1 || load2 := load2 + 1)
      END ;
  exit_cond =
       SELECT  ¬(load1 < top ∧ load2 ≥ top ∧ Q12 = FALSE ∧ Q21 = FALSE)
                ∧ ¬(load1 < top ∧ load2 ≥ top ∧ Q21 = TRUE)
                ∧ ¬(load1 > top ∧ load2 ≤ top ∧ Q12 = FALSE ∧ Q21 = FALSE)
                ∧ ¬(load1 > top ∧ load2 ≤ top ∧ Q12 = TRUE) THEN
              skip
       END
END
```

Figure 6: Refinement of the abstract machine for the load balancing algorithm.

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. The invariant I_2 above is given as the invariant of the refined machine together with the types of the variables. The initialisation and the operations are created in the same way here as in the machine specification. For verification purpose we introduce the operation *exit_cond*.

The termination condition for the auxiliary actions, *init_ij* and *init_ji*, are checked with a WHILE-loop in a machine implementation in B-Tool. 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. Furthermore, these operations are included in the IF-ELSIF-substitution within the loop. The *Choice*-function returns the value 1 or 2 and models the nondeterminism between the two operations *init_12* and *init_21*. If both would be enabled the value returned by the *Choice*-function determines which of them will be chosen. Finally, the invariant of the WHILE-loop consists of the invariant of the machine refinement *RefActionSystem* as well as of the relation on the link variables

$$
\begin{array}{ll}
\text{WHILE} & (load1 < top \wedge load2 \geq top \wedge Q12 = FALSE \wedge Q21 = FALSE) \\
& \quad \vee\ (load1 > top \wedge load2 \leq top \wedge Q12 = FALSE \wedge Q21 = FALSE) \\
& \text{DO} \\
& nn \leftarrow Choice(2); \\
& \text{IF } (load1 < top \wedge load2 \geq top \wedge Q12 = FALSE \wedge Q21 = FALSE) \\
& \quad \wedge\ (nn = 1) \\
& \quad\quad \text{THEN } Q21 := TRUE;\ C2 \leftarrow BTS\_BOOL(Q21) \\
& \text{ELSIF } (load1 > top \wedge load2 \leq top \wedge Q12 = FALSE \wedge Q21 = FALSE) \\
& \quad\quad \wedge\ (nn = 2) \\
& \quad\quad \text{THEN } Q12 := TRUE;\ C1 \leftarrow BTS\_BOOL(Q12) \\
& \text{ELSIF } (load1 < top \wedge load2 \geq top \wedge Q12 = FALSE \wedge Q21 = FALSE) \\
& \quad\quad \text{THEN } Q21 := TRUE;\ C2 \leftarrow BTS\_BOOL(Q21) \\
& \text{ELSIF } (load1 > top \wedge load2 \leq top \wedge Q12 = FALSE \wedge Q21 = FALSE) \\
& \quad\quad \text{THEN } Q12 := TRUE;\ C1 \leftarrow BTS\_BOOL(Q12) \\
& \text{END} \\
& \text{INVARIANT } (load1 : NAT \wedge load2 : NAT \\
& \quad\quad \wedge\ Q12 : BOOL \wedge Q21 : BOOL \\
& \quad\quad \wedge\ (Q12 = TRUE \Rightarrow load1 > top \wedge load2 \leq top) \\
& \quad\quad \wedge\ (Q21 = TRUE \Rightarrow load1 < top \wedge load2 \geq top) \\
& \quad\quad \wedge\ (Q12 = TRUE \Rightarrow Q21 = FALSE) \\
& \quad\quad \wedge\ (Q21 = TRUE \Rightarrow Q12 = FALSE)) \\
& \quad\quad \wedge\ nn : NAT1 \wedge nn \leq 2 \\
& \quad\quad \wedge\ C1 : NAT \wedge C2 : NAT \\
& \quad\quad \wedge\ (Q12 = TRUE \Rightarrow C1 = 1) \\
& \quad\quad \wedge\ (Q12 = FALSE \Rightarrow C1 = 0) \\
& \quad\quad \wedge\ (Q21 = TRUE \Rightarrow C2 = 1) \\
& \quad\quad \wedge\ (Q21 = FALSE \Rightarrow C2 = 0) \\
& \text{VARIANT } (2 - (C1 + C2)) \\
\text{END} &
\end{array}
$$

Figure 7: The auxiliary actions of the load balancing algorithm.

Q and the variables $C1$ and $C2$ of the variant. The relation gives the definition of the function BTS_BOOL used for the variant, returning one for a parameter with the value *true* and zero otherwise.

5.2.1 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. 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 the 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. We can also note that if the invariant would contain a quantifier expression, the user would need to perform the proof that the invariant is established as a proof by cases with the help of the interprover.

The proof obligations that are not discharged by the autoprover can first be discharged by the interprover using the user supplied rules. Then running the autoprover once again these obligations can be discharged by the autoprover as well using these same 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.

6 Conclusion

We have described how an action system is turned into an Abstract Machine specification in B-Tool. We have constructed refinements which give rise to proof obligations that correspond to the conditions of superposition refinement within the action systems framework. We are, thus, able to 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 proven automatically and the rest were easily proven in an interactive way. Here we only study the very first refinement step. The complete derivation is reported by Waldén [14]. A preliminar version of this manuscript using the B-Toolkit Release 2.0 has appeared elsewhere [15].

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, Martin Büchi and Philipp Heuberger 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 and the Cocos-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 Formal Software Development Methods, Vol. 2.* LNCS 552, Springer-Verlag, pages 398 – 405, Noordwijkerhout, the Netherlands, October 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 Symposium on Principles of Distributed Computing.* ACM, pages 131–142, Montreal, Quebec, Canada, August 1983.

[3] R. J. R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3), 1996.

[4] R. J. R. Back and K. Sere. From action systems to modular systems. In *Proc. of Formal Methods Europe (FME'94): Industrial Benefit of*

Formal Methods. LNCS 873, pp. 1 – 25, Barcelona, Spain, October 1994.

[5] M. Butler, E. Hedman, P. Nilsson, R. Rukšėnas, 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, Englewood Cliffs, New Jersey, 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. In *Proc. of Research Directions in High-Level Parallel Programming Languages*. LNCS 574, pages 338 – 348, Mont Saint-Michel, France, 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. Rukšėnas and J. von 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*. Including the *B-Toolkit User's Manual, Release 3.2*. B-Core (UK) Ltd., Oxford, U.K., 1996.

[12] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 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*, page 391, Los Angeles, USA, August 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. In *Proc. of Formal Methods Europe (FME'96): Symposium on Industrial Benefit and Advances in Formal Methods.* LNCS 1051, pages 84 – 103, Oxford, England, March 1996.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**