Data Refinement of Remote Procedures

To appear in
Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'97), LNCS, September 1997.

K. Sere
M. Waldén
Åbo Akademi University, Department of Computer Science
Lemminkäineng. 14 A, SF-20520 Turku, Finland

Turku Centre for Computer Science
TUCS Technical Report No 115
May 1997
ISBN 952-12-0018-9
ISSN 1239-1891

Abstract

Recently the action systems formalism for parallel and distributed systems has been extended with the procedure mechanism. This gives us a very general framework for describing different communication paradigms for action systems, e.g. remote procedure calls. Action systems come with a design methodology based on the refinement calculus. Data refinement is a powerful technique for refining action systems. In this paper we will develop a theory and proof rules for the refinement of action systems that communicate via remote procedures based on the data refinement approach. The proof rules we develop are compositional so that modular refinement of action systems is supported. As an example we will especially study the atomicity refinement of actions. This is an important refinement strategy, as it potentially increases the degree of parallelism in an action system.

Keywords: remote procedures, action systems, distributed systems, data refinement, atomicity refinement

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Remote procedure calls provide a very general communication mechanism between interacting systems. Hence, it is useful to have such a mechanism available when designing parallel and distributed systems. Remote procedure calls is a concept found in many programming languages and operating systems especially in client-server architectures [11, 19]. However, there seems to exist very little work on the formal derivation of systems with this communication mechanism in the literature [20]. In this paper we give a formal treatment of the remote procedure call mechanism and develop the needed proof rules for reasoning about the correctness and refinement of programs with remote procedures. We base our theory on the action systems framework. The reasoning is carried out relying on the refinement calculus of action systems.

An *action system* is a parallel or distributed program where parallel activity is described in terms of events, so called actions. The actions are atomic: if an action is chosen for execution, it is executed to completion without any interference from the other actions in the system. Several actions can be executed in parallel, as long as the actions do not share any variables. Atomicity guarantees that a parallel execution of an action system gives the same results as a sequential and nondeterministic execution.

The use of action systems permits the design of the logical behaviour of a system to be separated from the issue of how the system is to be implemented. The decision whether the action system is to be executed in a sequential or parallel fashion can be postponed to a later stage, when the logical behaviour of the action system has been designed. The construction of the program is thus done within a single unifying framework.

The action systems formalism was proposed by Back and Kurki-Suonio [5]. Later similar event-based formalisms have been put forward by several other researchers, see for example the work of Chandy and Misra [12], who describe their UNITY framework and Francez [14], who develops his IP-language. Recently, Back and Sere introduced the idea of remote procedures for action systems [8]. It has also been shown how this mechanism is used in practical program development [6, 21]. Furthermore, Kurki-Suonio has studied remote procedure calls in action systems using an object-oriented modeling formalism [15].

The action systems approach supports the stepwise refinement paradigm for the construction of parallel and distributed systems. The *refinement calculus* is a formalization of the stepwise refinement method of program construction. It was originally proposed by Back [2] and it has later been studied and extended by several researchers, see [17, 18] among others. In recent years *data refinement* within the refinement calculus has been a topic for extensive research [9, 10]. Back and Sere [4, 7] have extended the refinement calculus to handle *parallel algorithms* as well as *reactive programs*.

In both cases parallel and concurrent activity is modelled within a purely sequential framework. We shall here concentrate on reactive programs.

Procedures were added to the refinement calculus independently by Back [3] and Morgan [16], with slightly different ways in which parameter passing is handled. We extend the data refinement of reactive action systems as described by Back [4] to handle remote procedures. We extend the preliminary work by Back and Sere [8] considerably by developing a more general rule for the refinement of remote procedures together with a number of useful special cases of it. We pay special attention to the *compositionality* of the developed rules thus supporting a modular way of program development. Moreover, as an example we develop proof rules for an important refinement strategy for concurrent systems, namely the *refinement of atomicity* of atomic actions. This provides a convenient way to increase the degree of parallelism in these systems.

Overview of the paper. We proceed as follows. In section 2, we describe the action systems formalism using procedures. In section 3, we describe how action systems are composed into parallel systems and study the enabledness conditions for remote procedures. Section 4 develops the proof rules for handling action systems with remote procedures in a compositional manner within the refinement calculus. As an example on this we develop proof rules for the refinement of atomicity of remote procedures in section 5. We end in section 6 with some concluding remarks.

2 Action systems

An *action system* (with procedures) is a set of actions operating on local and global variables:

$$
\begin{array}{lll}
\mathcal{A} :: & |[& \mathbf{var}\ y^*, x := y0, x0; \\
 & & \mathbf{proc}\ p_1^* = P_1; \ldots ; p_n^* = P_n; \\
 & & \qquad q_1 = Q_1; \ldots ; q_l = Q_l; \\
 & & \mathbf{do}\ A_1 \ [\!]\ \ldots\ [\!]\ A_m\ \mathbf{od} \\
 &]|: & z, r
\end{array}
$$

Action system $\mathcal{A}$ describes a computation, in which the local variables x and the exported global variables y, marked with $*$, are first created and initialised. Then repeatedly any of the enabled actions $A_1, \ldots, A_n$ is nondeterministically selected for execution. The computation terminates if no action is enabled, otherwise it continues infinitely. Actions operating on disjoint sets of variables can be executed in any order or in parallel.

The local variables x are only referenced locally in $\mathcal{A}$, while the exported global variables, y, also can be referenced by other action systems. The imported global variables, z, of $\mathcal{A}$ are mentioned in $A_1, \ldots, A_n$, but not declared locally. The identifiers x, y and z are assumed to be pairwise distinct lists of variables. Thus, no redeclaration of variables is permitted.

A procedure is declared as $p = P$ with a *procedure header* p and a *procedure body* P. In action system $\mathcal{A}$ the procedures p are declared as exported procedures, marked with $*$. They can also be called from other action systems than $\mathcal{A}$. The procedures q, on the other hand, are declared as local procedures and are only called within $\mathcal{A}$. The procedures imported into $\mathcal{A}$ are denoted r. These are called from actions in $\mathcal{A}$, but are declared elsewhere. The local and global procedures are assumed to be distinct.

An action is said to be *local* to an action system, if it only refers to local variables of the action system. The procedures and actions are allowed to refer to all the state variables of an action system. Furthermore, each procedure and action may have local variables of its own.

Actions are taken to be *atomic*, meaning that only their input-output behaviour is of interest. They can be arbitrary sequential statements. Their behaviour can therefore be described by the weakest precondition predicate transformer of Dijkstra [13], where $\mathrm{wp}(A, Q)$ is the weakest precondition such that action A terminates in a state satisfying predicate Q. As we are only interested in the input-output behaviour of actions, we consider two actions to be equivalent if they always establish the same postcondition:

$$A = B \text{ iff } \forall Q : \mathrm{wp}(A, Q) = \mathrm{wp}(B, Q).$$

In addition to the statements considered by Dijkstra, we allow assumptions $[P]$ and assert statements $\{P\}$, where P is a predicate, as well as nondeterministic choice, $A \mathbin{[\!]} B$, between the actions A and B. The assumption $[P]$ can be thought of as stopping execution, if P does not hold.

$$\begin{array}{lcllcl}
\mathrm{wp}(\mathit{abort}, Q) & = & \mathit{false} & \mathrm{wp}(v := e, Q) & = & Q[v := e] \\
\mathrm{wp}(\mathit{skip}, Q) & = & Q & \mathrm{wp}(v := v'.(v' \in S), Q) & = & \\
\mathrm{wp}(\{P\}, Q) & = & P \wedge Q & \qquad\qquad (\exists v'.v' \in S) & \wedge & (\forall v'.v' \in S.Q[v := v']) \\
\mathrm{wp}([P], Q) & = & P \Rightarrow Q & \mathrm{wp}((A; B), Q) & = & \mathrm{wp}(A, \mathrm{wp}(B, Q)) \\
 & & & \mathrm{wp}((A \mathbin{[\!]} B), Q) & = & \mathrm{wp}(A, Q) \wedge \mathrm{wp}(B, Q)
\end{array}$$

Other operators can also be defined. The restriction we impose is that all actions are (finitely) conjunctive, hence excluding angelic nondeterminism [10]:

$$\mathrm{wp}(A, P \wedge Q) = \mathrm{wp}(A, P) \wedge \mathrm{wp}(A, Q)$$

All of the above operators are conjunctive or preserve conjunctivity. Conjunctivity implies monotonicity:

$$(P \Rightarrow Q) \Rightarrow (\mathrm{wp}(A, P) \Rightarrow \mathrm{wp}(A, Q))$$

Since the nondeterministic choice $A \mathbin{[\!]} B$ is included as an operator on actions, we can confine ourselves to action systems with only a single action. Hence, an action system $\mathcal{A}$ is in general of the form:

$$
\begin{array}{lll}
\mathcal{A} :: & |[& \textbf{var } y^*, x := y0, x0; \\
& & \textbf{proc } p_1^* = P_1; \ldots ; p_n^* = P_n; \\
& & \qquad q_1 = Q_1; \ldots ; q_l = Q_l; \\
& & \textbf{do } A \textbf{ od} \\
&]|\!: z, r &
\end{array}
$$

Definition of procedures. Procedure bodies and actions may contain procedure calls. The meaning of a call on a parameterless procedure $p = P$ in a statement S is determined by the *substitution principle*:

$$S \;=\; S[P/p],$$

i.e., the body P of procedure p is substituted for each call on this procedure in statement S.

Procedures in action systems can pass parameters. Three different parameter passing mechanisms for procedures are allowed, *call-by-value*, *call-by-result* and *call-by-value-result*. The parameter passing mechanism call-by-value is denoted with $p(\textbf{val}\, f)$, call-by-result with $p(\textbf{res}\, f)$, and call-by-value-result with $p(\textbf{valres}\, f)$, where f stands for the formal parameters. For simplicity, we will here assume that the procedures are not recursive.

Procedures with parameters can be handled by substitution in a similar way as parameterless procedures. Let $p(\textbf{val}\ x, \textbf{valres}\ y, \textbf{res}\ z) = P$ be a procedure declaration, where x denotes the formal call-by-value parameters, y the formal call-by-value-result parameters and z the formal call-by-result parameters. Then a call on p with the actual parameters a, b, c is removed by the substitution

$$S \;=\; S[P'/p(a, b, c)],$$

where P' is the statement

$$|[\ \textbf{var}\ x, y, z; x := a; y := b; P; b := y; c := z\ ]|.$$

If a procedure or action contains a call to a procedure that is not declared in the action system, then the behavior of the action system will depend on the way in which the procedures are declared in some other action system, which constitutes the environment of the action system as will be described later.

The definition of procedures is studied more carefully by Back [3] and Morgan [16].

Enabledness of an action. The procedure bodies and actions contain arbitrary program statements. A statement that establishes any postcondition is said to be miraculous. We take the view that a statement is only enabled in those initial states in which it behaves nonmiraculously. The guard of a statement characterises those states for which the statement is enabled:

$$gd(S) = \neg \mathrm{wp}(S, \mathit{false}).$$

The statement S is said to be *enabled* in a given state, when the guard is true in that state. The statement S is said to be *always enabled*, if $\mathrm{wp}(S, false) = false$ (i.e., $gd(S) = true$), and *always terminating*, if $\mathrm{wp}(S, true) = true$.

Both procedure bodies and actions will in general be *guarded commands*, i.e., statements of the form

$$C = g \to S,$$

where g is a boolean condition and S is a program statement. The guarded command $g \to S$ is defined using assumption as follows:

$$g \to S \;=\; [g]; S$$

In this case, the guard of C is $g \wedge \neg \mathrm{wp}(S, false)$. Hence, a guarded command $g \to S$ is only enabled when S is enabled and g holds. Moreover, the nondeterministic choice $A \mathbin{[\!]} B$ is enabled when either A or B is enabled:

$$gd(A \mathbin{[\!]} B) = gd(A) \vee gd(B)$$

If the body of each action and procedure of an arbitrary action system is always enabled, action systems coincide with the language of guarded commands. The *body* $bd(C)$ of C is defined by

$$bd(C) \;=\; gd(C) \to C \mathbin{[\!]} \neg gd(C) \to abort$$

We permit procedure bodies to have guards that are not identically true. Hence, it is possible that an action which is enabled calls a procedure which then turns out not to be enabled in the state in which it is called. This situation is then the same as if the action calling the procedure had not been enabled at all, and had therefore never initiated the call. In other words, the enabledness of an action is determined by the enabledness of the whole statement that is invoked when the action is executed, including enabledness of all procedures that might be invoked.

Example. Let us consider an example where $p = (b \to T)$ is a procedure and $g \to S; p$ an action that calls on p. Then the enabledness of this action is determined by the value of the action guard

$$gd(g \to S; p) \;=\; g \wedge gd(S; p)$$

where $gd(S;p)$ is calculated as follows:

$$
\begin{array}{ll}
 & gd(S;p) \\
= & gd(S;(b \to T)) \\
= & \neg\mathrm{wp}(S;(b \to T), false) \\
= & \neg\mathrm{wp}(S, \mathrm{wp}((b \to T), false)) \\
= & \neg\mathrm{wp}(S, b \Rightarrow \mathrm{wp}(T, false)) \\
= & \neg\mathrm{wp}(S, b \Rightarrow \neg gd(T)) \\
= & \{\text{assuming } T \text{ always enabled}\} \\
 & \neg\mathrm{wp}(S, \neg b)
\end{array}
$$

The calling action and the procedure will be considered as a single action, hence, an atomic entity:

$$g \wedge \neg\mathrm{wp}(S, \neg b) \to S; T.$$

3 Composing action systems

Consider two action systems, $\mathcal{A}$ and $\mathcal{B}$:

$$
\begin{array}{lll}
\mathcal{A} :: & |[& \textbf{var } v^*, x := v0, x0; \\
 & & \textbf{proc } r_1^* = R_1; \ldots ; r_m^* = R_m; \\
 & & \qquad p_1 = P_1; \ldots ; p_n = P_n; \\
 & & \textbf{do } A \textbf{ od} \\
 &]| : z &
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B} :: & |[& \textbf{var } w^*, y := w0, y0; \\
 & & \textbf{proc } s_1^* = S_1; \ldots ; s_k^* = S_k; \\
 & & \qquad q_1 = Q_1; \ldots ; q_l = Q_l; \\
 & & \textbf{do } B \textbf{ od} \\
 &]| : u &
\end{array}
$$

where $x \cap y = \emptyset$, $v \cap w = \emptyset$, and $r \cap s = \emptyset$. Furthermore, the local procedures declared in the two action systems are required to be distinct.

We define the *parallel composition* $\mathcal{A} \parallel \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ to be the action system $\mathcal{C}$ below

$$
\begin{array}{lll}
\mathcal{C} :: & |[& \textbf{var } b^*, x, y := b0, x0, y0; \\
 & & \textbf{proc } r_1^* = R_1; \ldots ; r_m^* = R_m; s_1^* = S_1; \ldots ; s_k^* = S_k; \\
 & & \qquad p_1 = P_1; \ldots ; p_n = P_n; q_1 = Q_1; \ldots ; q_l = Q_l; \\
 & & \textbf{do } A \parallel B \textbf{ od} \\
 &]| : a &
\end{array}
$$

where $a = z \cup u - (v \cup r \cup w \cup s)$, $b = v \cup w$, $r = (r_1 \cup \ldots \cup r_m)$, and $s = (s_1 \cup \ldots \cup s_k)$.

Thus, parallel composition will combine the state spaces of the two constituent action systems, merging the global variables and global procedures and keeping the local variables distinct. The imported identifiers denote those global variables and/or procedures that are not declared in either $\mathcal{A}$ or $\mathcal{B}$. The exported identifiers are the variables and/or procedures declared global in $\mathcal{A}$ or $\mathcal{B}$. The procedure declarations and the actions in the parallel composition consists of the procedure declarations and actions in the original systems.

Parallel composition is a way of associating a meaning to procedures that are called in an action system, but which are not declared there, i.e., the imported global procedures. The meaning can be given by a procedure declared in another action system, provided the procedure has been declared global in that action system.

Let $\mathcal{A}$ be an action system with exported and imported procedures. Let $\mathcal{E}$ be another action system. We call $\mathcal{E}$ the *full context* of $\mathcal{A}$ if there is no other action system than $\mathcal{E}$ that imports the exported procedures of $\mathcal{A}$ and if all the imported procedures of $\mathcal{A}$ are declared in $\mathcal{E}$. Hence, the exported procedures of $\mathcal{A}$ and $\mathcal{E}$ can be considered local in the parallel composition $\mathcal{A} \parallel \mathcal{E}$. Full context is an important concept when reasoning about the behaviour of action systems with remote procedures as will become clear later.

The behaviour of a parallel composition of action systems is dependent on how the individual action systems, the *reactive components*, interact with each other. We have for instance that a reactive component does not terminate by itself: termination is a global property of the composed action system. More on these topics can be found elsewhere [4].

Example. Consider a producer-consumer system

$$|[\textbf{ var } v^* \in \mathit{integer}, S, R \in \mathit{set\ of\ integer}; \mathcal{P}\mathit{rod} \parallel \mathcal{S}\mathit{nd} \parallel \mathcal{R}\mathit{ec} \parallel \mathcal{C}\mathit{ons} \ ]|$$

with four action systems $\mathcal{P}\mathit{rod}, \mathcal{S}\mathit{nd}, \mathcal{R}\mathit{ec}$, and $\mathcal{C}\mathit{ons}$ executing in parallel. The variable S denotes a set of messages to be transmitted between the producer $\mathcal{P}\mathit{rod}$ and the consumer $\mathcal{C}\mathit{ons}$. Similarly, the variable R denotes the set of messages received by the consumer action system $\mathcal{C}\mathit{ons}$. The producer uses the services of the sender $\mathcal{S}\mathit{nd}$ for the actual communication. These two systems communicate through the variable S. The consumer receives the messages via the receiver process $\mathcal{R}\mathit{ec}$. The receiver and the consumer communicate via the variable R. The variable v is some externally visible variable.

We study next the action systems $\mathcal{S}\mathit{nd}$ and $\mathcal{R}\mathit{ec}$ more carefully:

$$\begin{array}{lll} \mathcal{S}\mathit{nd} :: & |[& \textbf{var } x \in \mathit{integer}; \\ & & \textbf{do } S \neq \emptyset \rightarrow x := x'.(x' \in S); S := S - \{x\}; \mathit{Trans}(x) \textbf{ od} \\ &]|: & S, \mathit{Trans} \end{array}$$

$$\mathcal{R}ec :: \quad |[\ \textbf{proc}\ \mathit{Trans}^*(\textbf{val}\ v \in \mathit{integer}) = (R := R \cup \{v\})\ ]| : R$$

The action system $\mathcal{S}nd$ communicates with the receiver $\mathcal{R}ec$ via the global procedure *Trans*, which is in the import list of $\mathcal{S}nd$ and exported by $\mathcal{R}ec$. The two action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ are examples of reactive components. Therefore for instance the action system $\mathcal{S}nd$ does not terminate locally when S becomes empty, but rather it waits for new elements to appear in S.

The parallel composition of $\mathcal{S}nd$ and $\mathcal{R}ec$, $\mathcal{T} = \mathcal{S}nd \parallel \mathcal{R}ec$, is according to the rule above as follows:

$$\begin{array}{ll}\mathcal{T} :: & |[\ \ \textbf{var}\ x \in \mathit{integer};\\ & \quad\ \textbf{proc}\ \mathit{Trans}^*(\textbf{val}\ v \in \mathit{integer}) = (R := R \cup \{v\});\\ & \quad\ \textbf{do}\ S \neq \emptyset \rightarrow x := x'.(x' \in S); S := S - \{x\}; \mathit{Trans}(x)\ \textbf{od}\\ &]|: S, R\end{array}$$

Let us assume that $\mathcal{S}nd$ is the full context of $\mathcal{R}ec$. Hence, we can make the procedure *Trans* local, as it is only called by the single action in $\mathcal{S}nd$. This gives us the system $\mathcal{T}'$:

$$\begin{array}{ll}\mathcal{T}' :: & |[\ \ \textbf{var}\ x \in \mathit{integer};\\ & \quad\ \textbf{proc}\ \mathit{Trans}(\textbf{val}\ v \in \mathit{integer}) = (R := R \cup \{v\});\\ & \quad\ \textbf{do}\ S \neq \emptyset \rightarrow x := x'.(x' \in S); S := S - \{x\}; \mathit{Trans}(x)\ \textbf{od}\\ &]|: S, R\end{array}$$

We can remove the procedure by substituting the procedure body in place of the procedure call and the actual parameter for the formal parameter of the procedure. Substitution gives

$$\begin{array}{ll}\mathcal{T}'' :: & |[\ \ \textbf{var}\ x \in \mathit{integer};\\ & \quad\ \textbf{proc}\ \mathit{Trans}(\textbf{val}\ v \in \mathit{integer}) = (R := R \cup \{v\});\\ & \quad\ \textbf{do}\ S \neq \emptyset \rightarrow\\ & \qquad\qquad x := x'.(x' \in S); S := S - \{x\};\\ & \qquad\qquad |[\ \textbf{var}\ v \in \mathit{integer}; v := x; R := R \cup \{v\}\ ]|\\ & \quad\ \ \textbf{od}\\ &]|: S, R\end{array}$$

Finally, removing the redundant local variable v and procedure *Trans* that is not needed anywhere, gives us the action system $\mathcal{T}'''$:

$$\begin{array}{ll}\mathcal{T}''' :: & |[\ \ \textbf{var}\ x \in \mathit{integer};\\ & \quad\ \textbf{do}\ S \neq \emptyset \rightarrow x := x'.(x' \in S); S := S - \{x\}; R := R \cup \{x\}\ \textbf{od}\\ &]|: S, R\end{array}$$

Because all these versions are equivalent in the producer-consumer system, we can rewrite the system as

$$|[\ \textbf{var}\ v^* \in \mathit{integer}, S, R \in \mathit{set\ of\ integer}; \mathcal{P}rod \parallel \mathcal{T}''' \parallel \mathcal{C}ons\ ]|.$$

Enabledness. Since we permit procedure bodies of global as well as local procedures to have guards that are not identically true, it is possible that an action with a guard that evaluates to true, calls a procedure in another action system, which turns out not to be enabled in the state in which it is called. Then the action calling the global procedure is not either enabled in that state. We observe that we have a similar situation here as for the local procedure calls in section 2.

Let P be an exported procedure of some action system. We say that P is *locally enabled*, whenever the calling action cannot enable or disable P. An action A *cannot enable* another action B whenever

$$\neg gd(B) \Rightarrow \mathrm{wp}(A, \neg gd(B))$$

and A is always terminating. Moreover, A *cannot disable* B whenever

$$gd(B) \Rightarrow \mathrm{wp}(A, gd(B))$$

and A is always terminating. Hence, the enabledness of a locally enabled procedure can be determined independently of the caller. This is important when considering the enabledness of a remote procedure.

Let A be an action of the form $(g \to S; p)$ where p is a locally enabled remote procedure with the body P and S is always terminating. Then the guard of A, $gd(A)$, is computed as $gd(A) = g \wedge gd(S; p)$. Now we have that $gd(S; p) = \neg\mathrm{wp}(S, \neg gd(P))$ as seen previously. As P is locally enabled we have that $\neg\mathrm{wp}(S, \neg gd(P)) = gd(P)$ leaving us with $gd(A) = g \wedge gd(P)$. The following is an example of this situation.

Example. Assume that the set R of the producer-consumer action system is bounded above by an integer L. This is reflected in the action system $\mathcal{R}ec'$ as follows:

$$\mathcal{R}ec' :: \quad |[\ \mathbf{proc}\ \mathit{Trans}^*(\mathbf{val}\ v \in \mathit{integer}) = (|R| < L \to R := R \cup \{v\})\ ]| : R$$

Let $C \ = \ x := x'.(x' \in S); S := S - \{x\}$. We then have for the sending action in $\mathcal{S}nd$ that

$$\begin{array}{ll}
 & S \neq \emptyset \to x := x'.(x' \in S); S := S - \{x\}; \mathit{Trans}(x) \\
= & \quad \{\text{definitions of } C, \mathit{Trans}\} \\
 & S \neq \emptyset \to C; |[\ \mathbf{var}\ v; v := x; (|R| < L \to R := R \cup \{v\})\ ]| \\
= & \quad \{\text{removing local variable } v\} \\
 & S \neq \emptyset \to C; (|R| < L \to R := R \cup \{x\}) \\
= & \quad \{\text{definition of guard}\} \\
 & S \neq \emptyset \wedge \neg\mathrm{wp}(C, |R| \geq L) \to C; (|R| < L \to R := R \cup \{x\}) \\
= & \quad \{\text{calculation, definition of } C\} \\
 & S \neq \emptyset \wedge |R| < L \to x := x'.(x' \in S); S := S - \{x\}; R := R \cup \{x\}
\end{array}$$

Hence, the parallel composition of $\mathcal{S}nd$ and $\mathcal{R}ec'$ is the action system $\mathcal{T}''''$:

$$
\begin{array}{lll}
\mathcal{T}'''' :: & |[& \textbf{var } x \in \mathit{integer}; \\
& & \textbf{do } S \neq \emptyset \wedge |R| < L \rightarrow \\
& & \qquad\qquad x := x'.(x' \in S); S := S - \{x\}; R := R \cup \{x\} \\
& & \ \textbf{od} \\
&]|: S, R &
\end{array}
$$

Here the sending action is only enabled if $S \neq \emptyset$ and $|R| < L$ both hold. Moreover, *Trans* is locally enabled, even though the enabledness of its guard depends on the global variable R.

4 Refinement of action systems

As action systems are intended to be developed in a stepwise manner within the refinement calculus, this must be extended to support the added features. The methods for refinement of action systems with parallel composition that are proposed by Back and von Wright [4, 10] are here extended to the refinement of action systems with global procedures. The methods are mainly based on data refinement of action systems. We will here extend data refinement of action systems by Back [4] to data refinement of action systems with remote procedures by giving the necessary proof rules for these. The main addition is that when we make a data refinement of an action system, the procedure bodies of the action system have to be data refined as well, in addition to data refining the actions.

4.1 Data refinement

The refinement calculus for actions is based on the following definition. Action A is refined by action A', written $A \leq A'$, if, whenever A establishes a certain postcondition, so does A':

$$A \leq A' \text{ iff } (\forall Q.\mathrm{wp}(A, Q) \Rightarrow \mathrm{wp}(A', Q))$$

Together with the monotonicity of wp this implies that for a certain precondition, A' might establish a stronger postcondition than A, i.e., reduce the nondeterminism of A. Furthermore, A' might even establish postcondition *false*, i.e., behave miraculously.

Note that choice and sequential composition are both monotonic with respect to refinement in both operands. Moreover, the refinement relation itself is reflexive and transitive.

Let now A be an action that refers to the variables x, z and A' an action that refers to the variables x', z. Furthermore, let $R(x, x', z)$ be an *abstraction relation* between the abstract local variables x, the concrete local variables x' and the global variables z. Then action A is *data refined* by action A' using the abstraction relation R, denoted $A \leq_R A'$, if

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

where Q is a predicate on the variables x, z and $(\exists x.R \wedge Q)$ is a predicate on x', z. Note that the global variables z are not changed. From this definition it then follows that $A \leq_R A'$, if

(i) *Refinement of guards:* $R \wedge gd(A') \Rightarrow gd(A)$ and

(ii) *Refinement of bodies:*
$(\forall Q.R \wedge gd(A') \wedge \mathrm{wp}(bd(A), Q) \Rightarrow \mathrm{wp}(bd(A'), \exists x.R \wedge Q))$.

Data refinement of procedures. Let us now extend the data refinement of actions to the data refinement of procedures.

Let $\mathbf{proc}\ p(\mathbf{val}\ x, \mathbf{res}\ y) = P$ and let $R(u, u', z, x, y)$ be an abstraction relation on the involved variables such that

$$P \leq_R P',$$

where u and u' are the local variables of P and P', respectively, while z are the global variables of P and P'. We then have by the usual refinement rule above that

$$|[\ \mathbf{var}\ x, y; x := a; P; b := y\ ]| \ \leq\ |[\ \mathbf{var}\ x, y; x := a; P'; b := y\ ]|,$$

i.e., the call on $p(a, b)$ with declaration $\mathbf{proc}\ p(\mathbf{val}\ x, \mathbf{res}\ y) = P$ is refined by the call on $p(a, b)$ with the declaration $\mathbf{proc}\ p(\mathbf{val}\ x, \mathbf{res}\ y) = P'$.

This result holds for all parameter passing mechanisms, call-by-value, call-by-result and call-by-value-result, considered in this paper.

4.2 Data refinement of action systems

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

$$\begin{array}{lll}\mathcal{A} :: & |[& \mathbf{var}\ z^*, x := z0, x0; \\ & & \mathbf{proc}\ p_1^* = P_1; \ldots; p_n^* = P_n; \\ & & \qquad q_1 = Q_1; \ldots; q_l = Q_l; \\ & & \mathbf{do}\ A\ \mathbf{od} \\ &]|: u, r & \end{array}$$

$$\begin{array}{lll}\mathcal{A}' :: & |[& \mathbf{var}\ z^*, x' := z0, x'0; \\ & & \mathbf{proc}\ p_1^* = P_1'; \ldots; p_n^* = P_n'; \\ & & \qquad q_1 = Q_1'; \ldots; q_l = Q_l'; \\ & & \mathbf{do}\ A' \talloblong H\ \mathbf{od} \\ &]|: u, r & \end{array}$$

Let $R(x, x', z, u, f)$ be an abstraction relation on the local variables x and x', the global variables z and u, and the formal parameters f of the global procedures p and let every p be locally enabled. The action system $\mathcal{A}$ is then *data refined* by $\mathcal{A}'$ using R, denoted $\mathcal{A} \leq_R \mathcal{A}'$, if

(1) the abstraction relation is established by the initialisation for any initial values of u and f,

(2) each global procedure body P_i is data refined by the corresponding procedure body P'_i using R,

(3) if a global procedure P_i is enabled in action system $\mathcal{A}$, so is P'_i in $\mathcal{A}'$ or else an action in $\mathcal{A}'$ is enabled whenever R holds,

(4) the action A is data refined by the action A' using R,

(5) if the action A is enabled in $\mathcal{A}$, then either A' or H is enabled in $\mathcal{A}'$ whenever R holds,

(6) the auxiliary action H is a stuttering action (see [1, 4]) in the sense that it acts as a *skip* statement on the global variables u, z, and

(7) the auxiliary action H will terminate when executed in isolation.

Note: (a) We assume that the formal parameters $f_1, \ldots, f_n$ of the procedures are all disjoint, and let f be the list of all formal parameters, i.e., $f = f_1, \ldots, f_n$. (b) We do not introduce any new procedures here. They can be introduced into an action system as a separate step. (c) The refinement of the local procedures q is checked via the refinement of the main actions where they are called.

The requirements (1) - (7) above are sufficient to guarantee correct data refinement between action systems that are executed in isolation. When the action system $\mathcal{A}$ occurs in a parallel composition with other action systems, we have to take this context into account and add one more condition on the data refinement.

(8) the context will preserve the abstraction relation R.

This condition allows us to refine the individual action systems in a parallel composition separately. The only assumption we make about the components is that they all preserve the abstraction relation R.

The conditions (1) - (8) above are stated more formally in the following definition:

DEFINITION 1 *Let $\mathcal{A}$ and $\mathcal{A}'$ be action systems as above and $R(x, x', z, u, f)$ some abstraction relation. Furthermore, let every global procedure p_i be locally enabled. Then $\mathcal{A} \leq_R \mathcal{A}'$, if*

(1) *Initialisation:* $R(x0, x'0, z0, u, f)$,

(2) *Procedures:* $P_i \leq_R P'_i$,

(3) *Procedure guards:* $R \wedge gd(P_i) \Rightarrow (gd(P'_i) \ \vee \ gd(A') \ \vee \ gd(H))$,

(4) *Main actions:* $A \leq_R A'$,

(5) *Exit condition:* $R \wedge gd(A) \Rightarrow gd(A') \vee gd(H)$,

(6) *Auxiliary actions:* $skip \leq_R H$, and

(7) *Termination of auxiliary computation:* $R \Rightarrow \mathrm{wp}(\mathbf{do}\ H\ \mathbf{od}, true)$.

Furthermore, when $\mathcal{A}$ occurs in a parallel composition with another action system $\mathcal{E}$, then $\mathcal{A} \parallel \mathcal{E} \leq_R \mathcal{A}' \parallel \mathcal{E}$, if for every action E in $\mathcal{E}$:

(8) *Non-interference:* $R \wedge \mathrm{wp}(E, true) \Rightarrow \mathrm{wp}(E, R)$.

Relying on this definition we have the following theorem:

Theorem 1 *Let $\mathcal{A}$ and $\mathcal{A}'$ be as above and $\mathcal{A} \leq_R \mathcal{A}'$ for some R. Furthermore, let $\mathcal{B}$ be a full context of $\mathcal{A}$. Then*

$$\mathcal{A} \parallel \mathcal{B} \leq_R \mathcal{A}' \parallel \mathcal{B}.$$

Proof The correctness of this theorem is shown as follows. We assume that the conditions (1) - (8) hold for $\mathcal{A}$ and $\mathcal{A}'$. With these assumptions we then show that $\mathcal{A} \parallel \mathcal{B} \leq_R \mathcal{A}' \parallel \mathcal{B}$. Since $\mathcal{B}$ is a full context of $\mathcal{A}$, we can reduce data refinement with procedures to ordinary data refinement by expanding all the procedure calls in $\mathcal{A} \parallel \mathcal{B}$. Hence, there will be no global procedures in $\mathcal{A} \parallel \mathcal{B}$ and we can apply the corresponding proof rule of Back [4], which is the same as Definition 1, but without items (2) and (3) as Back considers action systems without procedures.

Let B be the action of $\mathcal{B}$. The main actions of the composed action system $\mathcal{A} \parallel \mathcal{B}$ are the actions A and B, while the action H is the auxiliary action. The conditions on the initialisation, the main action A and the auxiliary action H for the composed action system $\mathcal{A} \parallel \mathcal{B} \leq_R \mathcal{A}' \parallel \mathcal{B}$ follow directly from the conditions (1), (4), (6) and (7) for $\mathcal{A} \leq_R \mathcal{A}'$. The main action B where we substitute the calls on the procedures P and the exit condition must, however, be explicitely studied for the composed action system.

The refinement $B(P_i) \leq_R B(P'_i)$ of the main action B with prodedure P_i substituted for each call by the action B with P'_i substituted for each call is easily shown to be true by monotonicity of refinement. We have that by (2) and (8) for $\mathcal{A} \leq_R \mathcal{A}'$ the procedure refinement $P_i \leq_R P'_i$ holds and the action $B(P'_i)$ preserves the invariant.

We check the exit condition (5) for $\mathcal{A} \parallel \mathcal{B} \leq_R \mathcal{A}' \parallel \mathcal{B}$ more carefully. We must show that

$$R \wedge (gd(A) \vee gd(B(P_i))) \Rightarrow gd(A') \vee gd(H) \vee gd(B(P'_i))$$

which is proved as follows:

$$
\begin{array}{ll}
 & R \wedge (gd(A) \vee gd(B(P_i)) \\
\equiv & \{P_i \text{ locally enabled }\} \\
 & R \wedge (gd(A) \vee (gd(B) \wedge gd(P_i))) \\
\Rightarrow & \{(3)\} \\
 & R \wedge (gd(A) \vee (gd(B) \wedge (gd(P_i') \vee gd(A') \vee gd(H)))) \\
\Rightarrow & \{ \text{ logic } \} \\
 & R \wedge (gd(A) \vee (gd(B) \wedge gd(P_i')) \vee (gd(B) \wedge (gd(A') \vee gd(H)))) \\
\Rightarrow & \{(5) \text{ exit condition for } \mathcal{A} \leq_R \mathcal{A}'\} \\
 & R \wedge (gd(A') \vee gd(H) \vee (gd(B) \wedge gd(P_i')) \vee (gd(B) \wedge (gd(A') \vee gd(H)))) \\
\Rightarrow & \{ \text{ conjunction elimination } \} \\
 & gd(A') \vee gd(H) \vee (gd(B) \wedge gd(P_i')) \vee (gd(A') \vee gd(H)) \\
\Rightarrow & \{ \text{ idempotence } \} \\
 & gd(A') \vee gd(H) \vee (gd(B) \wedge gd(P_i')) \\
\equiv & \{P_i' \text{ locally enabled }\} \\
 & gd(A') \vee gd(H) \vee gd(B(P_i'))
\end{array}
$$

We have now shown that $\mathcal{A} \parallel \mathcal{B} \leq_R \mathcal{A}' \parallel \mathcal{B}$ is a correct data refinement in the sense of Back [4] for an arbitrary full context $\mathcal{B}$ of $\mathcal{A}$.

□

We observe that in case an action system $\mathcal{A}$ and its refinement $\mathcal{A}'$ import a global procedure p which is declared in action system $\mathcal{B}$, it follows trivially by monotonicity that $\mathcal{A} \leq_R \mathcal{A}'$ holds, since action system $\mathcal{B}$ containing the declaration of p is not changed in this refinement step.

The usefulness of data refinement for action systems comes from the fact that $\mathcal{A} \leq \mathcal{A}'$, if there exist an abstraction relation R such that $\mathcal{A} \leq_R \mathcal{A}'$.

Example. As an example of the data refinement we consider a more realistic version of the earlier sender-receiver system. Initially, we have three action systems, a sender $\mathcal{S}nd$, a receiver $\mathcal{R}ec$, and a buffer $\mathcal{B}uf$. The sender communicates with the buffer by making a (remote) call to the procedure *Insert* in $\mathcal{B}uf$. Similarly, the receiver extracts a messages from the buffer by making a call to the procedure *Delete* in $\mathcal{B}uf$.

We first show how the above system is described as the three action systems $\mathcal{S}nd, \mathcal{B}uf$, and $\mathcal{R}ec$:

$$
\begin{array}{lll}
\mathcal{S}nd :: & |[& \textbf{var } x \in \mathit{integer}; \\
 & & \textbf{do } S \neq \emptyset \rightarrow x := x'.(x' \in S); S := S - \{x\}; \mathit{Insert}(x) \textbf{ od} \\
 &]|: & S, \mathit{Insert}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}uf :: & |[& \textbf{var } B \in set\ of\ integer; l \in integer; \\
& & \textbf{proc } Insert^*(\textbf{val } v \in integer) = (B := B \cup \{v\}); \\
& & \textbf{proc } Delete^*(\textbf{res } v \in integer) = \\
& & \qquad (B \neq \emptyset \rightarrow v := v'.(v' \in B); B := B - \{v\}); \\
& & B := \emptyset; \\
& & \textbf{do } B \neq \emptyset \rightarrow\ l := size(B) \textbf{ od} \\
&]|: <> &
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{R}ec :: & |[& \textbf{var } x \in integer; \\
& & \textbf{do } true \rightarrow Delete(x); R := R \cup \{x\} \textbf{ od} \\
&]|: & R, Delete
\end{array}
$$

The action systems execute in parallel in the composed producer-consumer system $\mathcal{S}nd \parallel \mathcal{B}uf \parallel \mathcal{R}ec$ communicating through the global procedures *Insert* and *Delete*. All the activity in this system is sequentialized by the access to the shared resource B in the buffer process $\mathcal{B}uf$. Recall that only independent actions, i.e., actions that do not share variables either directly or indirectly through procedure calls, can be executed in a parallel fashion.

We next refine the buffer process $\mathcal{B}uf$ so that parallel activity becomes possible. The refinement $\mathcal{B}uf'$ is shown below:

$$
\begin{array}{lll}
\mathcal{B}uf' :: & |[& \textbf{var } B1, B2 \in set\ of\ integer; l \in integer; \\
& & \textbf{proc } Insert^*(\textbf{val } v \in integer) = (B1 := B1 \cup \{v\}); \\
& & \textbf{proc } Delete^*(\textbf{res } v \in integer) = \\
& & \qquad (B2 \neq \emptyset \rightarrow v := v'.(v' \in B2); B2 := B2 - \{v\}); \\
& & B1, B2 := \emptyset, \emptyset; \\
& & \textbf{do } B1 \neq \emptyset \vee B2 \neq \emptyset \rightarrow\ l := size(B1 \cup B2) \\
& & \ [\!]\ B1 \neq \emptyset \rightarrow \qquad\qquad\qquad\qquad\qquad\qquad [H] \\
& & \quad |[\ \textbf{var } x \in integer; \\
& & \qquad x := x'.(x' \in B1); B1 := B1 - \{x\}; B2 := B2 \cup \{x\}]| \\
& & \ \textbf{od} \\
&]|: <> &
\end{array}
$$

Here the buffer B has been implemented by two buffers $B1$ and $B2$.

Let us now prove that the refinement $\mathcal{B}uf \leq_I \mathcal{B}uf'$ is correct. We first have to choose an abstraction relation I. We have replaced the set B with the two sets $B1$ and $B2$ as:

$$I(B, B1, B2, Insert.v, Delete.v) =_{def} B = B1 \cup B2.$$

Following Definition 1 we have to show the following (we prefix the formal parameters with the corresponding procedure names in order to keep them distinct):

(1) *Initialisation:* $I(B_0, B1_0, B2_0, Insert.v, Delete.v)$ evaluates to $\emptyset = \emptyset \cup \emptyset$, which clearly holds.

(2) *Procedures:* There are two global procedures declared in $\mathcal{B}uf'$, *Insert* and *Delete*.

For *Insert* we have to show the condition

$$(B := B \cup \{Insert.v\}) \leq_I (B1 := B1 \cup \{Insert.v\}).$$

For *Delete* that is a guarded procedure we have to show both that

$$\begin{array}{ll} & \{B2 \neq \emptyset\}; (Delete.v := v'.(v' \in B); B := B - \{Delete.v\}) \\ \leq_I & \\ & (Delete.v := v'.(v' \in B2); B2 := B2 - \{Delete.v\}) \end{array}$$

and

$$B = B1 \cup B2 \wedge B2 \neq \emptyset \Rightarrow B \neq \emptyset$$

hold.

(3) *Procedure guards:* For *Insert* we trivially have

$$(B = B1 \cup B2) \wedge true \Rightarrow (true \;\vee\; (B1 \neq \emptyset \vee B2 \neq \emptyset) \;\vee\; (B1 \neq \emptyset)),$$

while the condition for *Delete* evaluates to:

$$(B = B1 \cup B2) \wedge (B \neq \emptyset) \Rightarrow ((B2 \neq \emptyset) \vee (B1 \neq \emptyset \vee B2 \neq \emptyset) \vee (B1 \neq \emptyset)).$$

(4) *Main actions:* There is one main action in $\mathcal{B}uf'$, which calculates the number of messages in the buffer. The refinement of this action is shown with the following conditions:

$$\{B1 \neq \emptyset \vee B2 \neq \emptyset\}; (l := size(B)) \;\leq_I\; (l := size(B1 \cup B2))$$

and

$$B = B1 \cup B2 \wedge (B1 \neq \emptyset \vee B2 \neq \emptyset) \Rightarrow (B \neq \emptyset).$$

(5) *Exit condition:* The exit condition follows easily by substitution:

$$(B = B1 \cup B2) \wedge (B \neq \emptyset) \Rightarrow (B1 \neq \emptyset \vee B2 \neq \emptyset) \;\vee\; (B1 \neq \emptyset)).$$

(6) *Auxiliary actions:* There is one auxiliary action in $\mathcal{B}uf'$, which transfers the messages from the buffer $B1$ to the buffer $B2$. The refinement of the auxiliary action H:

$$\begin{array}{l} B1 \neq \emptyset \rightarrow |[\ \mathbf{var}\ x \in integer; \\ \qquad\qquad x := x'.(x' \in B1); B1 := B1 - \{x\}; \\ \qquad\qquad B2 := B2 \cup \{x\}\ ]| \end{array}$$

is checked by substitution in $skip \leq_I H$. Since H refers only to local variables of $\mathcal{B}uf'$, condition (6) is trivially true.

(7) *Termination of auxiliary computation:* The termination of auxiliary actions in $\mathcal{B}uf'$ is proved by substituting the auxiliary action H of $\mathcal{B}uf'$ into $I \Rightarrow \mathrm{wp}(\mathbf{do}\ H\ \mathbf{od}, true)$. To prove this, we use the variant $|B1|$.

(8) *Environment:* As $\mathcal{B}uf$ occurs in a parallel composition with other action systems, we must check that they preserve the abstraction relation I. This is immediately seen to be the case as I only mentions local variables of $\mathcal{B}uf$ and $\mathcal{B}uf'$ and the actions in $\mathcal{S}nd$ as well as $\mathcal{R}ec$ only change the global variables S and R.

The conditions above for $\mathcal{B}uf$ are easily seen to be true and, thus, $\mathcal{B}uf \leq_I \mathcal{B}uf'$ is a correct data refinement. Due to this and since $\mathcal{S}nd \parallel \mathcal{R}ec$ is a full context of $\mathcal{B}uf$, it follows from Theorem 1 that

$$\mathcal{S}nd \parallel \mathcal{B}uf \parallel \mathcal{R}ec \ \leq_I \ \mathcal{S}nd \parallel \mathcal{B}uf' \parallel \mathcal{R}ec.$$

4.3 Special cases of data refinement

Theorem 1 gives a very general rule for refining action systems with exported procedures. In this section we study two possible special cases of the theorem w.r.t. the conditions on procedures. It follows from the theorem that in the refined system $\mathcal{A}'$ it may be the case that the procedure P'_i is never enabled even though the corresponding procedure P_i in $\mathcal{A}$ is enabled (see condition (3) of Definition 1). This has to do with the fact that nondeterminism is decreased by refinement steps.

If we want to assure that the procedure P'_i eventually will be enabled in $\mathcal{A}'$ in case P_i is enabled in $\mathcal{A}$, we need a stronger condition to replace the condition (3). The following corollary of Definition 1 gives such a condition:

Corollary 1 *Let $\mathcal{A}$ and $\mathcal{A}'$ be action systems and $R(x, x', u, z, f)$ some abstraction relation as in Definition 1. Then $\mathcal{A} \leq_R \mathcal{A}'$ under the conditions of Definition 1, if the condition (3) is replaced with the condition*

$$R \wedge gd(P_i) \Rightarrow (gd(P'_i) \ \vee \ \mathrm{wp}(\mathbf{do}\ \neg gd(P'_i) \rightarrow (A' \ [\!]\ H)\ \mathbf{od}, true)).$$

Proof We need to prove that the condition in Corollary 1 implies condition (3) in Definition 1:

$$(gd(P'_i) \vee \mathrm{wp}(\mathbf{do}\ \neg gd(P'_i) \rightarrow (A' \ [\!]\ H)\ \mathbf{od}, true)) \Rightarrow (gd(P'_i) \vee gd(A') \vee gd(H))$$

Let us assume that the left hand side of the implication holds. Then either $gd(P'_i)$ holds or it will eventually be enabled by $A' \ [\!]\ H$. In case $gd(P'_i)$ holds then the right hand side of the implication trivially follows. On the other hand, if $\neg gd(P'_i)$ holds, the loop $\mathbf{do}\ \neg gd(P'_i) \rightarrow (A' \ [\!]\ H)\ \mathbf{od}$ eventually terminates and it will terminate in a state where $gd(P'_i)$ holds. In order for

this to happen $A' \parallel H$ has to be enabled, i.e., $gd(A' \parallel H) = gd(A') \vee gd(H)$ has to hold. We have that the wp-expression on the left hand side implies $\neg gd(P') \wedge (gd(A') \vee gd(H))$, which in turn implies the right hand side.
□

An example of using this rule is given by Waldén [21] when deriving a distributed load balancing algorithm.

Since the global procedures in an action system are called from other action systems, it is sometimes preferred that the guards of the procedures before and after the refinement are equivalent under the abstraction relation R. The following corollary captures this:

COROLLARY 2 *Let $\mathcal{A}$ and $\mathcal{A}'$ be action systems and $R(x, x', u, z, f)$ some abstraction relation as in Definition 1. Then $\mathcal{A} \leq_R \mathcal{A}'$ under the conditions of Definition 1, if the condition (3) is replaced with the condition*

$$R \wedge gd(P_i) \Rightarrow gd(P_i').$$

PROOF The above condition trivially implies the condition in Corollary 1. This condition together with condition (2) implies that the guards are equivalent under R.
□

Observe that the above corollary corresponds to the refinement rule originally studied by Back and Sere [8].

Example. By refining the action system $\mathcal{B}uf$ in a slightly different manner than in the previous subsection, we need not strengthen the guard of the procedure *Delete*. We can, thus, use the condition (3) of Corollary 2 when proving that this new action system $\mathcal{B}uf''$ is a refinement of the old specification $\mathcal{B}uf$ according to Definition 1.

Let the action system $\mathcal{B}uf''$ be as follows:

$$
\begin{array}{ll}
\mathcal{B}uf'' :: & |[\ \ \mathbf{var}\ B1, B2 \in set\ \ of\ \ integer; \\
& \quad \mathbf{proc}\ Insert^*(\mathbf{val}\ v \in integer) = (B1 := B1 \cup \{v\}); \\
& \quad \mathbf{proc}\ Delete^*(\mathbf{res}\ v \in integer) = \\
& \qquad\qquad (B1 \neq \emptyset \vee B2 \neq \emptyset \rightarrow \\
& \qquad\qquad\quad \mathbf{do}\ B1 \neq \emptyset \rightarrow \\
& \qquad\qquad\qquad\quad |[\ \ \mathbf{var}\ x \in integer; \\
& \qquad\qquad\qquad\qquad x := x'.(x' \in B1); B1 := B1 - \{x\}; \\
& \qquad\qquad\qquad\qquad B2 := B2 \cup \{x\} \\
& \qquad\qquad\qquad\quad]| \\
& \qquad\qquad\quad \mathbf{od}; \\
& \qquad\qquad\quad (B2 \neq \emptyset \rightarrow v := v'.(v' \in B2); B2 := B2 - \{v\}) \\
& \qquad\qquad); \\
& \quad B1, B2 := \emptyset, \emptyset; \\
& \quad \mathbf{do}\ B1 \neq \emptyset \vee B2 \neq \emptyset \rightarrow\ \ l := size(B1 \cup B2)\ \mathbf{od} \\
&]| : <>
\end{array}
$$

The buffer B in $\mathcal{B}uf$ has been implemented by the two buffers $B1$ and $B2$ in $\mathcal{B}uf''$ and the abstraction relation $I(B, B1, B2, \mathit{Insert}.v, \mathit{Delete}.v)$ is $B = B1 \cup B2$. We now prove that the refinement $\mathcal{B}uf \leq_I \mathcal{B}uf''$ is correct.

The conditions (1)-(4) of Definition 1 for the initialisation, the procedure *Insert* and the main action were already shown for $\mathcal{B}uf \leq_I \mathcal{B}uf'$ in the example of the previous subsection, since these constructs are the same in $\mathcal{B}uf''$ as in $\mathcal{B}uf'$. Also the environment $\mathcal{S}nd \parallel \mathcal{R}ec$ considered here is the same as in the previous subsection and, hence, also condition (8) has been proven. Since there are no auxiliary actions in $\mathcal{B}uf''$, the conditions (6) and (7) are trivial. This leaves condition (2) and (3) for procedure *Delete*, as well as condition (5) to be shown.

(2) *Procedures:* For *Delete* we show that

$$(B \neq \emptyset \rightarrow \mathit{Delete}.v := v'.(v' \in B); B := B - \{\mathit{Delete}.v\})$$

is data refined by

$$\begin{array}{l}
(B1 \neq \emptyset \vee B2 \neq \emptyset \rightarrow \\
\quad \textbf{do } B1 \neq \emptyset \rightarrow \\
\qquad\quad |[\ \textbf{var } x \in \mathit{integer}; \\
\qquad\qquad x := x'.(x' \in B1); B1 := B1 - \{x\}; B2 := B2 \cup \{x\}\ ]| \\
\quad\ \textbf{od}; \\
\quad (B2 \neq \emptyset \rightarrow v := v'.(v' \in B2); B2 := B2 - \{v\}) \\
)
\end{array}$$

using I. It is clear that the statement part is correctly data refined, as relying on R if an element is removed from B, it is the same as removing an element from $B1 \cup B2$. As $B1 \neq \emptyset \vee B2 \neq \emptyset$ holds, we may have the $B1$ contains a number of elements. These are then in an arbitrary order moved to $B2$ making $B1$ empty. The loop holds $B1 \cup B2$ invariant and it terminates, as $|B1|$ decreases at each iteration with one. Thereafter, $B2 \neq \emptyset$ necesserily holds, and an element can be removed from $B2$. We also have to show that

$$(B = B1 \cup B2) \wedge (B1 \neq \emptyset \vee B2 \neq \emptyset) \Rightarrow B \neq \emptyset,$$

which as well is trivially true, since the guards of the procedures *Delete* in $\mathcal{B}uf$ and $\mathcal{B}uf''$ are even equivalent under the abstraction relation I.

(3) *Procedure guards:* For the guards of *Delete* we further show that

$$(B = B1 \cup B2) \wedge B \neq \emptyset \Rightarrow B1 \neq \emptyset \vee B2 \neq \emptyset.$$

This is trivially true, due to the equivalence between these guards under I.

(5) *Exit condition:* The exit condition is proved by

$$(B = B1 \cup B2) \wedge B \neq \emptyset \Rightarrow B1 \neq \emptyset \vee B2 \neq \emptyset,$$

since the main actions are the only actions in $\mathcal{B}uf$ and $\mathcal{B}uf''$. The condition is trivially true, because the guards of the main actions in $\mathcal{B}uf$ and $\mathcal{B}uf''$ are equivalent under I.

Due to the above reasoning and since $\mathcal{S}nd \parallel \mathcal{R}ec$ is a full context of $\mathcal{B}uf$, it follows from Theorem 1 that

$$\mathcal{S}nd \parallel \mathcal{B}uf \parallel \mathcal{R}ec \ \leq_I \ \mathcal{S}nd \parallel \mathcal{B}uf'' \parallel \mathcal{R}ec.$$

Superposition refinement. In the so called *superposition refinement* new functionality is added to a system while preserving the old functionality. Theorem 1 can be used for proving the correctness of superposition refinement of action systems with procedures by restricting the way old variables are replaced by new ones. The variables x' in $\mathcal{A}'$ are defined as $x' = x \cup y$, where x are the variables of $\mathcal{A}$ and y are some auxiliary variables added to the refined action system. The superposition method with procedures is studied and used to derive and analyse distributed algorithms in various ways by Waldén [21].

5 Atomicity refinement of global procedures

An important refinement rule for action systems is the atomicity refinement. This possibly leads to an increased degree of parallelism in the system, because concurrency within action systems is modelled by allowing independent actions to be executed in parallel. In the presence of procedures, we can increase the number of such actions by splitting a procedure body into a number of actions and allowing the return from a procedure call to take place before the completion of the procedure body. The rest of the procedure body then constitutes an action of its own. In the action systems framework this amounts to refining the atomicity of an action.

In this section we consider two typical cases of atomicity refinement as an example on the general data refinement rule for action systems with global procedures. The proofs are given here in a somewhat informal manner, but they can be fully formalized and hints for that are given in connection to the proofs.

Transformation 1. Consider an action system

$$\begin{array}{lll} \mathcal{A} :: & |[& \mathbf{var}\ v^*, x := v0, x0; \\ & & \mathbf{proc}\ p^* = (b \vee gd(B) \rightarrow \mathbf{do}\ (b \rightarrow T) \ [\!]\ B\ \mathbf{od}); \\ & & \mathbf{do}\ A\ \mathbf{od} \\ &]|\!: z & \end{array}$$

where the global procedure p is of the given form. Note that a call on p and the calling action together form an atomic action as usual. Moreover, p is required to be locally enabled.

We want to refine $\mathcal{A}$ to

$$\begin{array}{ll} \mathcal{A}' :: & |[\ \ \textbf{var}\ v^*, x := v0, x0; \\ & \quad \textbf{proc}\ p^* = (b \rightarrow T); \\ & \quad \textbf{do}\ A\ [\!]\ B\ \textbf{od} \\ &]|:\ z \end{array}$$

The effect of the above transformation becomes clear when we look at the action systems in a full context $\mathcal{E}$,

$$\begin{array}{ll} \mathcal{E} :: & |[\ \ \textbf{var}\ w^*, y := w0, y0; \\ & \quad \textbf{do}\ C\ [\!]\ D\ \textbf{od} \\ &]|:\ u, p \end{array}$$

where action C calls p and D is some other action not calling p. We have that

$$\begin{array}{ll} \mathcal{A} \parallel \mathcal{E} :: & |[\ \ \textbf{var}\ v^*, w^*, x, y := v0, w0, x0, y0; \\ & \quad \textbf{proc}\ p = (b \vee gd(B) \rightarrow \textbf{do}\ (b \rightarrow T)\ [\!]\ B\ \textbf{od}); \\ & \quad \textbf{do}\ A\ [\!]\ C\ [\!]\ D\ \textbf{od} \\ &]|:\ z, u \end{array}$$

Hence, action B is only executed as part of calling p in a single atomic action C in $\mathcal{A} \parallel \mathcal{E}$, while B is executed interleaved with actions A, C, D in the composed action system $\mathcal{A}' \parallel \mathcal{E}$:

$$\begin{array}{ll} \mathcal{A}' \parallel \mathcal{E} :: & |[\ \ \textbf{var}\ v^*, w^*, x, y := v0, w0, x0, y0; \\ & \quad \textbf{proc}\ p = (b \rightarrow T); \\ & \quad \textbf{do}\ A\ [\!]\ B\ [\!]\ C\ [\!]\ D\ \textbf{od} \\ &]|:\ z, u \end{array}$$

Observe that we consider a full context $\mathcal{E}$ that makes p local to $\mathcal{A} \parallel \mathcal{E}$ and $\mathcal{A}' \parallel \mathcal{E}$.

Correctness. We need to show under which conditions the system $\mathcal{A}$ is correctly data refined by $\mathcal{A}'$. These conditions are given by the following lemma:

LEMMA 1 *Let $\mathcal{A}$ and $\mathcal{A}'$ be as above. Furthermore, let R be the trivial abstraction relation that does not change the local variables ($x = x'$) of $\mathcal{A}$ and let p as well as B be locally enabled in $\mathcal{A}$. Then $\mathcal{A} \leq_R \mathcal{A}'$, if we have that*

(i) $R \Rightarrow \mathrm{wp}(T, \neg b \wedge \neg gd(B))$,

(ii) $R \wedge (b \vee gd(B)) \Rightarrow \mathrm{wp}(\textbf{do}\ \neg b \rightarrow B\ \textbf{od}, true)$,

(iii) *$skip \leq_R B$, and*

(iv) $R \Rightarrow \mathrm{wp}(\mathbf{do}\ B\ \mathbf{od}, \mathit{true})$.

Furthermore, when $\mathcal{A}$ occurs in a parallel composition with another action system $\mathcal{F}$ then $\mathcal{A} \parallel \mathcal{F} \leq_R \mathcal{A}' \parallel \mathcal{F}$, if for every action F in $\mathcal{F}$

$$R \wedge \mathrm{wp}(F, \mathit{true}) \Rightarrow \mathrm{wp}(F, R).$$

The effect of the above transformation is that of refining the atomicity of the procedure body of p. Let us see what the conditions of the lemma tell us. The procedure p is given in a general form in the action system $\mathcal{A}$. In order to be able to refine the procedure p to $(b \to T)$ in $\mathcal{A}'$, the actions $(b \to T)$ and B of procedure p in $\mathcal{A}$ have to fulfill the following restrictions. Firstly, the action $(b \to T)$ has to disable itself, condition (i), since it should be executed only once at a call on p, because this is the effect of p in the refined system $\mathcal{A}'$. Secondly, the action B should eventually enable b, so that $(b \to T)$ is executed at least once. Since B enables b, condition (ii), T has to disable the action B, condition (i), to prohibit the action $(b \to T)$ to be enabled more than once within the loop in $\mathcal{A}$.

For the proof we will need the following property of actions A and B. We say that A *commutes* with B, if $A; B \leq B; A$. This again holds, if

(a) B cannot enable A,

(b) A cannot disable B, and

(c) $\{gd(A) \wedge gd(B)\}; bd(A); bd(B) \leq bd(B); bd(A)$.

This and other properties have been studied by Back and Sere [7].

We note that the conditions (i) and (ii) in Lemma 1 are strengthened compared to the conditions in the corresponding lemma of Back and Sere [8].

Proof The above lemma is a special case of Theorem 1. Hence, we need to show the correctness of conditions (1)–(8) of Definition 1.

Since the initialisation and the action A are the same in $\mathcal{A}$ and $\mathcal{A}'$, and due to the nature of the guard of procedure p, the conditions (1), (3), (4) and (5) hold trivially. Furthermore, since the conditions (iii) and (iv) as well as the environment condition in the above lemma hold, also conditions (6) - (8) hold trivially. Hence, the refinement of the procedure p, condition (2), is the only non-trivial condition to prove:

$$(b \vee gd(B) \to \mathbf{do}\ (b \to T) \ [\!]\ B\ \mathbf{od}) \ \leq_R \ (b \to T).$$

We proceed by considering the three different cases when the procedure p in $\mathcal{A}$ is enabled.

- In case $b \wedge gd(B)$ holds, either T or B can be executed. If T is executed first then the loop terminates due to condition (i) which states that T disables $b \rightarrow T$ and B. On the other hand, if B is executed first, eventually b will hold due to condition (ii) and the actions B will not be executed forever due to condition (iv).

- In case $b \wedge \neg gd(B)$ holds, the action $b \rightarrow T$ can be executed and the loop terminates immediately due to condition (i).

- In case $\neg b \wedge gd(B)$ hold, the action B is executed first enabling the action $b \rightarrow T$ at some point due to condition (ii) as in the first case above.

Thus, T will be executed once and $b \rightarrow T$ is a refinement of the original procedure body in all the possible cases.

To formally prove the above cases we need the fact that the action calling $b \rightarrow T$ commutes with the action B. These commute, due to condition (iii) which states that B only assigns to local variables, i.e., it is a refinement of *skip*, and the fact that p is locally enabled. Hence, the calling action cannot enable or disable B.

□

Example. Let us now give an example of the atomicity refinement. We again consider a buffer system as in the previous sections. The buffer system $\mathcal{B}uf_1$ is as follows:

$$
\begin{array}{lll}
\mathcal{B}uf_1 :: & |[& \mathbf{var}\ B1, B2 \in set\ \ of\ \ integer; trans \in boolean; \\
& & \mathbf{proc}\ Insert^*(\mathbf{val}\ v \in integer) = (B1 := B1 \cup \{v\}); \\
& & \mathbf{proc}\ Delete^*(\mathbf{res}\ v \in integer) = \\
& & \qquad ((B1 \neq \emptyset \vee B2 \neq \emptyset) \wedge \neg trans \rightarrow \\
& & \qquad\quad \mathbf{do}\ B1 \neq \emptyset \wedge \neg trans \rightarrow \\
& & \qquad\qquad\quad |[\ \mathbf{var}\ x \in integer; \\
& & \qquad\qquad\qquad x := x'.(x' \in B1); B1 := B1 - \{x\}; \\
& & \qquad\qquad\qquad B2 := B2 \cup \{x\}\ \ ]| \\
& & \qquad\quad [\!]\ \ B2 \neq \emptyset \wedge \neg trans \rightarrow \\
& & \qquad\qquad v := v'.(v' \in B2); B2 := B2 - \{v\}; trans := true \\
& & \qquad\quad \mathbf{od} \\
& & \qquad); \\
& & B1, B2 := \emptyset, \emptyset; trans := false; \\
& & \mathbf{do}\ trans \rightarrow trans := false\ \mathbf{od} \\
&]|:<> &
\end{array}
$$

As in the examples in the previous section the procedure *Insert* adds elements to the buffer $B1$. The procedure *Delete* either transfers an element from the buffer $B1$ to the buffer $B2$ or removes an element from $B2$. When an element is removed from $B2$, the buffer system $\mathcal{B}uf_1$ assigns true to the boolean variable *trans*. In this way it states that an element has been removed from $B2$ as the result of calling the *Delete*-procedure and the loop within *Delete* terminates. An action in $\mathcal{B}uf_1$ takes care of assigning false to the variable *trans* again, in order to allow another element to be removed from $B2$.

We refine the atomicity of $\mathcal{B}uf_1$ according to Lemma 1 and thereby get the action system $\mathcal{B}uf_1'$ below.

$$
\begin{array}{lll}
\mathcal{B}uf_1' :: & |[& \mathbf{var}\ B1, B2 \in \mathit{set\ of\ integer}; \mathit{trans} \in \mathit{boolean};\\
& & \mathbf{proc}\ \mathit{Insert}^*(\mathbf{val}\ v \in \mathit{integer}) = (B1 := B1 \cup \{v\});\\
& & \mathbf{proc}\ \mathit{Delete}^*(\mathbf{res}\ v \in \mathit{integer}) =\\
& & \qquad (B2 \neq \emptyset \wedge \neg \mathit{trans} \rightarrow\\
& & \qquad\quad v := v'.(v' \in B2); B2 := B2 - \{v\}; \mathit{trans} := \mathit{true});\\
& & B1, B2 := \emptyset, \emptyset; \mathit{trans} := \mathit{false};\\
& & \mathbf{do}\ \mathit{trans} \rightarrow \mathit{trans} := \mathit{false}\\
& & [\!]\ \ B1 \neq \emptyset \wedge \neg\ \mathit{trans} \rightarrow \qquad\qquad\qquad\qquad\qquad [H]\\
& & \quad |[\ \ \mathbf{var}\ x \in \mathit{integer};\\
& & \qquad x := x'.(x' \in B1); B1 := B1 - \{x\}; B2 := B2 \cup \{x\}\ ]|\\
& & \ \mathbf{od}\\
&]|{:}{<}{>} &
\end{array}
$$

In $\mathcal{B}uf_1'$ the procedure *Delete* is reduced to deleting an element from $B2$ and an auxiliary action, marked H, transferring data from $B1$ to $B2$ is introduced. The variables are not changed in this refinement step, so the abstraction relation R is

$$B1 = B1' \wedge B2 = B2' \wedge \mathit{trans} = \mathit{trans}'.$$

The unprimed variables refer to variables in $\mathcal{B}uf_1$ and the primed to variables in $\mathcal{B}uf_1'$.

We can now verify that the data refinement $\mathcal{B}uf_1 \leq_R \mathcal{B}uf_1'$ is correct by showing that the conditions in Lemma 1 hold.

(i) First we prove that the procedure *Delete* in $\mathcal{B}uf_1'$ will disable itself and the auxiliary procedure:

$$
\begin{array}{ll}
R \wedge B2 \neq \emptyset \Rightarrow & \mathrm{wp}((v := v'.(v' \in B2); B2 := B2 - \{v\}; \mathit{trans} := \mathit{true}),\\
& \qquad \neg(B1 \neq \emptyset \vee B2 \neq \emptyset) \vee \mathit{trans}).
\end{array}
$$

Since *trans* is assigned true, condition (i) holds trivially.

(ii) The auxiliary action H will establish the guard of the procedure *Delete* in $\mathcal{B}uf_1'$. This can be shown by substitution as follows:

$$
\begin{array}{l}
R \wedge (B1 \neq \emptyset \vee B2 \neq \emptyset) \wedge \neg\ \mathit{trans} \Rightarrow\\
\qquad \mathrm{wp}(\mathbf{do}\ (B2 = \emptyset\ \vee\ \mathit{trans}) \rightarrow H\ \mathbf{od}, \mathit{true})
\end{array}
$$

We consider two cases for the proof. In the first case we consider $B2$ not to be empty and $\neg trans$ to hold, then *Delete* is immediately enabled and the loop **do** $(B2 = \emptyset \;\vee\; trans) \rightarrow H$ **od** terminates. In the other case where $B2$ is empty and $\neg trans$ holds, $B1$ is not empty according to the lefthand side of the implication. The auxiliary action H is then enabled and executed. It thereby transfers an element to $B2$, enables *Delete* and terminates the loop. Hence, condition (ii) holds.

(iii) The auxiliary action H refines *skip*, since it only refers to local variables of $\mathcal{B}uf_1'$ and, thus, condition (iii) holds.

(iv) By taking $|B1|$ as a variant for the loop **do** H **od**, we can easily see that the last condition for atomicity refinement holds:

$$R \Rightarrow \mathrm{wp}(\mathbf{do}\ H\ \mathbf{od}, true).$$

Each time H is executed, the number of elements in $B1$ will be decreased by one. Since there is a finite number of elements in $B1$ at the beginning of the loop execution, the loop will eventually terminate and, hence, condition (iv) holds.

Hence, $\mathcal{B}uf_1'$ is a correct atomicity refinement of $\mathcal{B}uf_1$ using R.

Transformation 2. The condition (i) in Lemma 1 states that the procedure body $b \rightarrow T$ needs to be of a certain restricted form. The restrictions on T are not needed if we instead consider the procedure body p of the action system $\mathcal{A}$ to be:

$$b \vee gd(B) \rightarrow (\mathbf{do}\ B\ \mathbf{od}; (b \rightarrow T)).$$

The correctness of $\mathcal{A} \leq_R \mathcal{A}'$ is then stated in the following corollary of Lemma 1.

COROLLARY 3 *Let the procedure p be $b \vee gd(B) \rightarrow (\mathbf{do}\ B\ \mathbf{od}; (b \rightarrow T))$ in $\mathcal{A}$ above. For the rest let $\mathcal{A}, \mathcal{A}'$ and R be as above. Then $\mathcal{A} \leq_R \mathcal{A}'$, if we have that*

(i) $R \wedge (b \vee gd(B)) \Rightarrow \mathrm{wp}(\mathbf{do}\ B\ \mathbf{od}, b)$,

(ii) *$skip \leq_R B$, and*

(iii) $R \Rightarrow \mathrm{wp}(\mathbf{do}\ B\ \mathbf{od}, true)$.

Furthermore, when $\mathcal{A}$ occurs in a parallel composition with another action system $\mathcal{F}$ then $\mathcal{A} \parallel \mathcal{F} \leq_R \mathcal{A}' \parallel \mathcal{F}$, if for every action F in $\mathcal{F}$

$$R \wedge \mathrm{wp}(F, true) \Rightarrow \mathrm{wp}(F, R).$$

Here we require that the action B when executed in isolation will terminate, condition (iii), and at termination b will hold, condition (i), whenever R holds.

PROOF We again proceed proving the refinement of the procedure p:

$$(b \vee gd(B) \rightarrow (\mathbf{do}\ B\ \mathbf{od}; (b \rightarrow T))) \ \leq_R \ (b \rightarrow T)$$

as the only non-trivial condition from Definition 1.

We consider the same cases as for Lemma 1 when procedure p in $\mathcal{A}$ is enabled.

- In case $b \wedge gd(B)$ holds, the action B is executed a number of times. It will terminate due to condition (iii) and b will hold at termination due to condition (i). After that T is executed.

- In case $b \wedge \neg gd(B)$ holds, the loop with B actions will terminate immediately and the action $b \rightarrow T$ is executed, because b holds.

- In case $\neg b \wedge gd(B)$ holds, the behaviour is as in the first case, where $b \rightarrow T$ is enabled upon termination of the loop due to condition (i) and T is executed.

Thus, T will be executed once and $b \rightarrow T$ is a refinement of the original procedure body in all the possible cases.

For the formal proof, the same facts are needed as above in the proof of the previous lemma.

□

Example. We now give an example of this kind of atomicity refinement. Again we consider a variant of the buffer system, $\mathcal{B}uf_2$:

$$
\begin{array}{lll}
\mathcal{B}uf_2 :: & |[& \mathbf{var}\ B1, B2 \in set\ of\ integer; \\
& & \mathbf{proc}\ Insert^*(\mathbf{val}\ v \in integer) = (B1 := B1 \cup \{v\}); \\
& & \mathbf{proc}\ Delete^*(\mathbf{res}\ v \in integer) = \\
& & \qquad (B1 \neq \emptyset \vee B2 \neq \emptyset \rightarrow \\
& & \qquad\quad \mathbf{do}\ B1 \neq \emptyset \rightarrow \\
& & \qquad\qquad\quad |[\ \mathbf{var}\ x \in integer; \\
& & \qquad\qquad\quad\ \ x := x'.(x' \in B1); B1 := B1 - \{x\}; \\
& & \qquad\qquad\quad\ \ B2 := B2 \cup \{x\}\ \ ]| \\
& & \qquad\quad\ \mathbf{od}; \\
& & \qquad\quad (B2 \neq \emptyset \rightarrow v := v'.(v' \in B2); B2 := B2 - \{v\}) \\
& & \qquad); \\
& & B1, B2 := \emptyset, \emptyset; \\
&]|:<> &
\end{array}
$$

The procedure *Insert* is the same as in the previous examples. However, the procedure *Delete* now transfers the elements in the buffer $B1$ to the buffer $B2$ and then removes an element from $B2$.

We refine the atomicity of the action system $\mathcal{B}uf_2$ following Corollary 3 into the action system $\mathcal{B}uf_2'$:

$$
\begin{array}{lll}
\mathcal{B}uf_2' :: & |[& \textbf{var } B1, B2 \in set\ of\ integer; \\
& & \textbf{proc } Insert^*(\textbf{val } v \in integer) = (B1 := B1 \cup \{v\}); \\
& & \textbf{proc } Delete^*(\textbf{res } v \in integer) = \\
& & \qquad (B2 \neq \emptyset \rightarrow v := v'.(v' \in B2); B2 := B2 - \{v\}); \\
& & B1, B2 := \emptyset, \emptyset; \\
& & \textbf{do } B1 \neq \emptyset \rightarrow \qquad\qquad\qquad\qquad\qquad\qquad [H] \\
& & \qquad |[\textbf{ var } x \in integer; \\
& & \qquad\quad x := x'.(x' \in B1); B1 := B1 - \{x\}; B2 := B2 \cup \{x\} \quad]| \\
& & \textbf{od} \\
&]|:<> &
\end{array}
$$

As in the previous example on atomicity refinement *Delete* is reduced to remove elements from buffer $B2$, while the auxiliary action, H, transfers the elements from $B1$ to $B2$. The abstraction relation R again states that the variables are not changed: $B1 = B1' \wedge B2 = B2'$.

We can now show that the atomicity refinement $\mathcal{B}uf_2 \leq_R \mathcal{B}uf_2'$ is correct by proving the three conditions of Corollary 3.

In condition (i) the auxiliary action H enables the procedure *Delete* as follows:

$$R \wedge (B1 \neq \emptyset \vee B2 \neq \emptyset) \Rightarrow \mathrm{wp}(\textbf{do } H \textbf{ od}, B2 \neq \emptyset).$$

We assume that either $B1$ or $B2$ is not empty. In case $B1$ would be empty, $B2$ is not empty and the loop immediately terminates in $(B2 \neq \emptyset)$. On the other hand, if $B1$ is not empty, then the auxiliary action H will be enabled and elements moved to $B2$ which then not will be empty upon termination of the loop. Hence, condition (i) holds.

Conditions (ii) and (iii) for the auxiliary action H hold due to a similar reasoning as in the previous example on atomicity refinement.

Hence, $\mathcal{B}uf_2'$ is a correct atomicity refinement of $\mathcal{B}uf_2$ using R according to Corollary 3.

6 Concluding remarks

The methods for handling procedure calls and action systems developed within the refinement calculus were extended in order for them to be applicable to the action systems framework with remote procedures. We developed a general data refinement rule together with some realistic rules that have stronger assumptions. In practice, the stronger rules and especially the rules for atomicity refinement might be preferable.

The main requirement we impose on procedures is that they are locally enabled. This is, however, not really a restriction in practice, because the guards of the remote procedures are not required to refer to local variables only, but are allowed to refer also to global variables.

The work of Udink [20] is related to ours in the sense that Udink extends the UNITY framework with remote procedure calls and develops the needed proof rules. His procedures are, however, of a more restricted format, as he does not allow guarded procedures. Furthermore, Udink does not develop special rules for atomicity refinement as we do here.

The general mechanism for communication, remote procedures with guards, gives rise to some interesting implementation issues, where efficiency of implementation has to be traded against generality of the mechanism. The main problem here is to detect enabledness of an action. This problem is similar to the problem of guaranteeing atomicity of transactions in distributed data bases. The advantage of such a communication mechanism is that it provides us with a high level way of describing the intended behavior of a parallel or reactive system. The concrete behavior can be derived within the refinement calculus as proposed here.

Acknowledgements

The work reported here was carried out within the Cocos-project supported by the Academy of Finland. The authors would like to thank Ralph Back and Michael Butler for comments on the topics treated here.

References

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

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

[3] R. J. R. Back. *Procedural abstraction in the refinement calculus.* Department of Computer Science, Åbo Akademi University, Turku, Finland, 1987. Report A–55.

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

[5] 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*, pp. 131–142, 1983.

[6] R. J. R. Back, A. J. Martin and K. Sere. Specifying the Caltech asynchronous microprocessor. *Science of Computer Programming* 26(1996), pp. 79–97, Elsevier.

[7] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming* 13, pp. 133–180, 1989.

[8] R. J. R. Back and K. Sere. Action systems with synchronous communication. In E.–R. Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET'94)*, IFIP Transactions A–56, pp. 107–126, North–Holland 1994.

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

[10] R. J. R. Back and J. von Wright. Trace refinement of action systems. In *Proc. of CONCUR'94.* LNCS 836, pages 367 – 384, Uppsala, Sweden, August 1994. Springer–Verlag.

[11] M. Broy, S. Merz and K. Spies, editors. *Formal Systems Specification: The RPC-Memory Specification Case Study. Proceedings*, volume 1169 of *Lecture Notes in Computer Science*. Springer–Verlag, 1996.

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

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

[14] N. Francez. Cooperating proofs for distributed programs with multiparty interactions. *Information Processing Letters*, 32:235–242, 1989.

[15] R. Kurki-Suonio. Incremental specification with joint actions: The RPC-memory specification problem. In [11].

[16] C. Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17–28, 1988.

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

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

[19] A. S. Tanenbaum. *Modern Operating Systems.* Prentice–Hall International, 1992.

[20] R. Udink *Program Refinement in UNITY–like Environments.* PhD thesis, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1995.

[21] M. Waldén *Formal derivation of a distributed load balancing algorithm.* Series A–172, Reports on Computer Science and Mathematics, Åbo Akademi University, Finland, 1995.

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**