Software—Concepts and Tools
© Springer-Verlag 1996

From Action Systems to Modular Systems

Ralph J. R. Back

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

Kaisa Sere

University of Kuopio, Department of Computer Science and Applied Mathematics, P.O.Box 1627, FIN-70211 Kuopio, Finland
e-mail: Kaisa.Sere@uku.fi

Abstract. Action systems are used to extend program development methods for sequential programs to the design of parallel and reactive systems. They provide a general description of reactive systems capable of modelling *terminating*, *possibly aborting* and *infinitely repeating* systems. We show how to use the action system model to develop *modular systems*. A module may export and import variables, it may provide access procedures for other modules, and it may itself access procedures of other modules. Modules may have autonomous internal activity and may execute in parallel or in sequence. Modules may be nested within each other. They may communicate by shared variables, shared actions, a generalized form of remote procedure calls and persistent data structures. Both synchronous and asynchronous communication between modules is supported. The paper shows how a single framework can be used for the specification of large systems, the modular decomposition of the system into smaller units, and the transformation of the modules into program modules that can be described in a standard programming language and executed on standard hardware.

Keywords: action system, module, modularization, reactive system, system design, specification, decomposition, parallel design, language extension

1. Introduction

An important issue in programming language design is to identify methods for program composition which permit the replacement of one component with little or no regard for the rest of the program. A program that is structured in a way that permits this is loosely said to be *modular*, and the composition method is called a *modularization mechanism*. Here we will study three important modularization mechanisms: *procedures*, *parallel composition* and *data encapsulation*. These are common mechanisms that are found in many programming languages and have proved to be indespensable in mastering the complexity of constructing large programs.

The modularization facilities that we put forward here are based on the *action system* approach, introduced by Back and Kurki-Suonio [5, 6]. An *action system* describes the behaviour of a parallel system in terms of the atomic actions that can take place during the execution of the system. Action systems provide a general description of reactive systems capable of modelling systems that may or may not terminate and where atomic actions need not terminate themselves. Arbitrary sequential program statements can be used to describe an atomic action. Related formalisms, like the UNITY approach [12] and the IP-language [14], lack most of the modularization facilities described here.

The focus in this paper is on describing how modular constructs can be modelled in an action system framework. We show how a single framework can be used for the specification of large systems, the modular decomposition of the system into smaller units, and the transformation of the modules into program constructs that can be described in a standard programming language and executed on standard hardware.

Overview

We show how to extend a simple base language with modularization mechanisms, to support the development of large programs. We take the *guarded commands* language of Dijkstra [13] as our basis. First we describe the extensions to this language that permit convenient expression of high-level specifications. Then we extend it with a standard block construct to permit the use of local variables in program stamenents. This is our *base language*.

Our first extension of the base language is to permit local constants and variables in a block. This allows us to model parameterless procedures. We explain the meaning of constants by *syntactic reduction*, i.e., by showing how a statement with constants can be reduced to an equivalent statement in the base language (which has no constants).

Procedures with parameters require an extension of statements and, in particular, of the block construct. We extend blocks with formal parameter declarations and define an *adaption operation*, which takes a block with formal parameters and a list of actual parameters and constructs a standard block without formal parameters that serves as an ordinary program statement. Adaption models *parameter passing* and is again defined by syntactic reduction.

Our next extension is to add parallellism to our language in the form of *action systems*. An action system is basically a block with some local variables and with a single iteration that can be executed in a parallel fashion under appropriate atomicity constraints. However, any parallel execution of the iteration statement is equivalent in effect to some sequential nondeterministic execution. Hence, from a logical point of view, it suffices to treat action systems as ordinary sequential statements.

We define *parallel composition* of action systems in order to model *reactive systems*. The definition is again by syntactic reduction, showing how any parallel composition of action systems can be reduced to a single action system. To get more flexibility in forming parallel composition, we generalize the block construct by permitting local variables to be *exported* (or *revealed*), so that they can be accessed from other action systems in a parallel composition. We also make the distinction between local variables that are created at block entry and destroyed at block exit, and local variables that exist before block entry and continue to exist after the action system has terminated. The latter models *persistent data structures.*

With these extensions, we have in fact a *module description language* where, in addition to parallel composition of modules, we also define *sequential composition* of modules and *nested* modules with *hiding*. The modules that we define in this way provide both *data encapsulation* and *parallel processing*.

Having defined the module facility, we finally show how these modules communicate with each other. Communication between modules does not need any further extension of the language; it is a consequence of features already defined. Basically, we show that different ways of partitioning an action system into parallel components corresponds to different kinds of communication mechanisms. We exemplify three such mechanisms: *shared variable* communication, *shared action* communication (of which CSP/Occam communication is a special case [16, 17) and *remote procedure calls* (of which, e.g., Ada rendezvous is a special case [22]). A fourth communication mechanism is provided by *persistent variables* in sequential composition of action systems.

We end this treatment with a short discussion on how to extend an existing modular programming language to provide the features that we have defined here. We choose the *Oberon* language [23] for this purpose, as it is a very simple language, has certain basic design decissions that are very close to what we need, and does not have any explicit facilities for parallellism.

We describe the modularization facilities as a sequence of syntactic and semantic extensions of a simple base language because this is a simple way of presenting the underlying ideas. This should not be treated as a concrete language definition because we have simplified a number of issues and omitted a number of restrictions that would be needed in order to turn the proposal into a rigorous language definition.

2. Guarded Command

Statements S in the *guarded command language* of Dijkstra [13] are defined by the following syntax:

$$
\begin{array}{rll}
S ::= & \mathit{abort} & \{\text{abortion, nontermination}\} \\
\mid & \mathit{skip} & \{\text{empty statement}\} \\
\mid & w := e & \{\text{(multiple) assignment}\} \\
\mid & S_1;\ S_2 & \{\text{sequential composition}\} \\
\mid & \text{if } A \text{ fi} & \{\text{conditional composition}\} \\
\mid & \text{do } A \text{ od} & \{\text{iterative composition}\} \\
\\
A ::= & g \rightarrow S_1 & \{\text{guarded command, action}\} \\
\mid & A_1 \,[]\, A_2 & \{\text{nondeterministic choice}\}
\end{array}
$$

Here $w = w_1, ..., w_n$ is a list of program variables and $e = e_1, ..., e_n$ a corresponding list of expressions ($n >$ 0). Also, g is a boolean expression. Each variable w_i is associated with a type that restricts the values that the variable can take. We do not define the syntactic classes of variables, types and expressions here; any standard definition of their syntax may be assumed.

Sequential composition of statements and nondeterministic choice are associative, so we may write S_1; S_2; ...; S_n and $A_1 \,[]\, A_2 \,[]\, ... \,[]\, A_n$ without explicit bracketing. Nondeterministic choice is also commutative, so the order in which the alternatives are listed is not important. We write gA for the *guard g* and sA for the *body S* of an action $A = g \rightarrow S$.

A statement operates on a set of program variables. It describes how a program state is changed. The program state is determined by the values of the variables that are manipulated by the statements. We write $S : v$ when we want to explicitly indicate that the variables v are the program variables in the state that S operates on.

The *abort* statement will be considered equivalent to a nonterminating loop. The *skip* statement has no effect on the state. The *assignment statement* is well defined if w_i is a variable and e_i and w_i have the same type, $i = 1, \dots n$. The effect is to first compute the values e_i, $i = 1, \dots, n$ and then assign w_i the value e_i, for $i = 1, \dots, n$. Sequential composition $S_1; S_2; \dots; S_n$ is executed by first executing S_1 and, if S_1 terminates, continuing by executing S_2, and so on, executing S_n last.

An *action* $A_i = g_i \rightarrow S_i$ in the *conditional statement* $A_1 [] \dots [] A_n$ *fi* is said to be *enabled* in a given state if the *guard* g_i is true in that state. The conditional statement is executed by choosing some enabled action A_i and executing its *body* S_i. If no action is enabled in the state, then the conditional statement behaves as an *abort* statement. If more than one action is enabled, then one of these is chosen nondeterministically. The nondeterminism is *demonic* in the sense that there is no way of influencing which action is chosen.

The iteration statement *do* $A_1 [] \dots [] A_n$ *od* is executed by first choosing an enabled action $A_i = g_i \rightarrow S_i$, executing its body S_i, and then repeating this until a state is reached where no action is enabled anymore and the iteration stops. If there is more than one action for which the guard is satisfied, then the choice is nondeterministic in the same way as for the conditional statement.

The iteration statement terminates only if a state is eventually reached where no action is enabled. Otherwise, the execution goes on forever. If no action is enabled initially, iteration terminates immediately and the iteration statement behaves as a *skip* statement.

The precise semantics of guarded commands can be defined in terms of *weakest preconditions*, as is done in the original work by Dijkstra [13]; or we can give operational semantics for these constructs, essentially describing the sequence of states that an execution of a guarded command gives rise to. The specific choice of semantic definition does not matter here; the essential point to note is that the semantic meaning of guarded commands can be given in a simple and precise manner.

Example
The following statement is intended to sort the values of the variables $x = x1, x2, \dots, x5$ into nondecreasing order according to the variable index:

```
SORT0 :: do $x1 > x2 \rightarrow x1, x2 := x2, x1$    {EX1}
         [] $x2 > x3 \rightarrow x2, x3 := x3, x2$    {EX2}
         [] $x3 > x4 \rightarrow x3, x4 := x4, x3$    {EX3}
         [] $x4 > x5 \rightarrow x4, x5 := x5, x4$    {EX4}
         od
```

The only actions that can take place are exchanging two neighbouring values when they are in the wrong order. Termination is guaranteed because each executed action decreases the number of pairs that are in the wrong order. Upon termination, the values of the variables have been sorted. Execution is nondeterministic because two actions may be enabled at the same time. This nondeterminism, however, does not affect the outcome of the execution.

3. Specification Facilities

The following extensions to guarded command language make the language considerably more powerful as a specification language. These are the *nondeterministic assignment* [1] (called *specification statement* in [19]) and *miraculous statements* [4, 19, 20, 21]. We can include them in the language by the following extensions to the syntax:

```
S ::= ...
   | $w := w'.Q$    {nondeterministic assignment}
   | A              {action}
```

Here Q is a predicate on the initial and final values of the state variables.

An example of a nondeterministic assignment statement is:

$$x, y := x', y'.(x' = x + 1 \wedge y \leq y').$$

This statement assigns new values x', y' to the variables x and y, where the values satisfy the condition $x' = x + 1 \wedge y \leq y'$. The statement would abort if no such values x', y' exist. If there is more than one choice for the values x' and y', then the choice is nondeterministic.

Permitting an action as a statement means that a statement about to be executed in a sequential composition need not be enabled, e.g., as in

$$x := x + 1; (x \neq 1 \rightarrow x := y).$$

If the action is reached in a state where the guard does not hold, then the statement is considered to terminate *miraculously* in the weakest precondition semantics, in the sense that it will establish any postcondition that we want. A statement is said to be *enabled* when it can avoid miraculous termination. In the example above, the whole statement is enabled if $x \neq 0$ (because then $x \neq 1$ when the second part is about to be executed). Thus, this statement is in fact equivalent to the statement

$$x \neq 0 \to x := x + 1; x := y,$$

which is an action of the form permitted by the ordinary guarded commands language.

Operationally, we can view a statement that is not enabled in a certain state as being *deadlocked*; i.e., execution cannot proceed because executing the statement is not permitted. We assume that the system tries to avoid this kind of deadlock, so that when it reaches a deadlock, it will backtrack and try to execute some alternative statement. Only when all alternatives have been tried does the system actually recognize that it is in a real deadlock. Consider as an example the statement

$$(x := x + 1; (x \neq 1 \to x := y)) [] (x = 0 \to x := y).$$

Here the first alternative is chosen if $x \neq 0$, while the second alternative is chosen when $x = 0$. Although both alternatives can deadlock, the combined statement does not deadlock because the system will choose the alternative that does not deadlock.

Neither a nondeterministic assignment statement nor a naked guarded command are in general executable on a real computer. However, they are executable in an idealized sense and do not present any difficulties in a logical analysis of program correctness. Hence, they are included in the specification language as abstractions that are convenient during program specification and development, but which need to be removed during the program development process in order to get an executable program.

4. Blocks with Local Variables

The *block* construct allows local variables to be introduced in guarded commands. We extend the syntax of guarded commands as follows:

S ::= ... |	{block}
D ::= < *empty* >	{empty declaration}
| var $w := e$	{local variable declaration}
| $D_1; D_2$	{declaration composition}
B ::= begin D; S end	{block}.

The types of local variables in a block construct have to be indicated explicitly if they cannot be inferred from the expression e. Composition of declarations is associative.

The block *begin D; S end* is executed by adding the new *local variables* declared in D to the state, initializing their values to the given expressions respectively, then executing S and, upon termination of S, deleting the local variables. We assume that the local variables in a declaration are all distinct from each other. They also have to be different from the global variables of a block; i.e., redeclaration of variables is not allowed. The initialization expressions may not refer to any local variables introduced in the block. The local variables declared in a block are only accessible to statements inside the block.

We choose the guarded commands language extended with nondeterministic assignment, actions and blocks as our basic statement language. Extending the weakest precondition semantics of Dijkstra's guarded commands to incorporate these extensions is straightforward (see, e.g., [2, 8]) as is the extension of an operational semantics of guarded commands with these constructs. Thus, in the sequel we assume that the semantics of the basic statements is well defined and well understood. We describe the meaning of subsequent extensions to the basic statement language by showing how to reduce the extended statements to these basic statements.

5. Constants

We extend the block construct by also permitting declaration of *local constants* in blocks, in addition to local variables. We extend the syntax as follows:

S ::= ... | c	{constant}
D ::= ... | const $c = C$	{local constant declaration}

Here c is an indentifier that stands for the value C. The constant c can be used anywhere in the program where C can be used. The constant may denote an expression, statement, action or block. Above we extended statements by also permitting constants as statements. The syntax of expressions also needs to be extended in a similar way.

We treat constants as mere syntactic sugar, which can be removed by simple rewriting (macro expansion):

begin D; const $c = C$; D; S end
= begin D; D; $S[C/c]$ end.

The reduction substitutes C for each occurrence of the identifier c in S. The semantics of programs with constants is thus explained by showing how to reduce such a program to another, equivalent program that does not have any constants. We refer to a definition of this form as a *syntactic reduction.*

A (parameterless) procedure is a special case of a constant. The procedure declaration *const* $p = S$ is usually written as *proc* $p = S$. A procedure call p is handled by substituting the body S for each call p. We assume that recursive calls are disallowed, so the substitution eliminates the procedure calls. Recursive procedures do not present any real difficulty, but require a fixpoint operator on statements [2] which, for simplicity, we choose not to introduce here.

Because redeclaration of global variables in a block is not permitted, dynamic and static scoping will be the same. Hence, substitution does give the correct binding for global variables.

6. Parameters and Adaption

Procedures with parameters are also a special case of constant declarations, but we need to introduce (formal) parameter declarations and a new statement construct called *adaption* to handle parameter passing. Hence we extend the syntax of our language as follows:

```
S ::= ... | B(e, x, y)     {adaption}

D ::= ...
    | val w                {value parameter}
    | upd w                {update parameter}
    | res w:= w0           {result parameter}
```

The values w_0 are default values for the result parameters. The formal parameters are considered as local declarations of a block. These need to be instantiated to actual parameters e, x, y in an adaption statement before the block can be used as an ordinary statement.

To conform with standard practice, we usually write a procedure declaration

const p = begin val v; upd w; res $r := r_0$; D; T end,

where D contains no parameter declarations, in the form

proc p (val v; upd w; res $r := r_0$;) = begin D; T end.

We define *adaption* of a block B with (formal) parameters to an actual parameter list (e, x, y) by syntactic reduction, as follows:

begin val v; upd w; res $r := r_0$; D; T end (e, x, y)
= begin var $v, w, r := e, x, r_0$; D; T; $x, y := w, r$ end.

All parameters become local variables after the reduction. The value parameters are assigned the actual parameter values on block entry, and the update parameters are assigned the values of the variables that they are bound to. The actual result parameter is assigned its value at block exit. Note that we have a default value for the result parameter on block entry. This means that the result parameter will have a well-defined value even if it is never assigned to.

A procedure declaration is eliminated in the same way as before: each procedure identifier is replaced by the block that it denotes. If the procedure has parameters, then adaption is subsequently required for parameter passing.

Example
We could write the sort program *SORT*0 using a local procedure as follows:

```
SORT0' :: begin const swap =
                      begin upd l, r; l; r := r, l end
          do x1  > x2 → swap(x1, x2)
          [] x2  > x3 → swap(x2, x3)
          [] x3  > x4 → swap(x3, x4)
          [] x4  > x5 → swap(x4, x5)
          od end.
```

Syntactic substitution expands the call swap$(x1, x2)$ as follows:

swap$(x1, x2)$
= {substitute definition of swap for the identifier}
begin *upd* l, r; $l, r := r, l$ end $(x1,x2)$
= {adaption}
begin var $l, r := x1, x2$; $l, r := r, l$; $x1, x2 := l, r$ end
= {simplification}
$x1, x2 := x2, x1$.

Removing the procedure declaration will thus reduce *SORT*0' to the equivalent program *SORT*0.

7. Action System

We will model parallel and reactive systems as guarded commands of a special form, which we refer to as *action systems* $\mathcal{A}$:

$\mathcal{A}$ ::= begin D; do A_1 [] ... [] A_m od end {action system}

The local variables declared in D and the global variables together form the *state variables* of the action system. The set of state variables accessed in action A is denoted vA and the set of variables declared in D is denoted vD. The action system may also declare constants, such as procedures. For simplicity, we assume here that there are no parameter declarations in an action system.

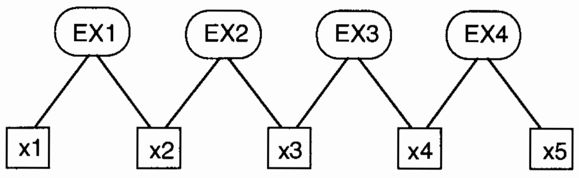

Figure 1. Access relation for *SORT0*.

In the absence of local declarations, the action system reduces to a simple iteration statement. The sorting program *SORT0* of the previous section is an example of such an action system.

An action system provides a global description of the system behavior. The state variables determine the state space of the system. The actions determine what can happen during an execution. The execution terminates when no action is enabled anymore.

Access Relation

Let us denote by rA the variables that are read by action A (the *read access* of A) and by wA the variables that are written by A (the *write access* of A). Thus $vA = rA \cup wA$.

Figure 1 shows the *access graph* of the system *SORT0*. The actions and variables of the system form the nodes in this graph. An edge connects action A to variable x if $x \in vA$. A read access is denoted by an arrow from x to A and a write access by an arrow from A to x. A read-write access is denoted by an undirected edge.

Sequential and Parallel Execution.

The semantics given to guarded commands above prescribes a sequential execution for action systems. This gives us the *interleaving semantics*, where only one action is executed at a time, in a nondeterministic order constrained only by the enabledness of actions.

However, action systems are intended to describe parallel systems. Hence, we permit the execution of actions to proceed in parallel also; i.e., we permit *overlapping* action executions.

Consider again our simple sorting program. If both actions

$$x2 > x3 \rightarrow x2, x3 := x3, x2 \quad \text{and}$$
$$x4 > x5 \rightarrow x4, x5 := x5, x4$$

are enabled, then either one could be executed next. Because the two actions do not share any variables, the effect is the same if we execute the actions one after the other, in either order, or if we execute them in parallel, so that the executions are in fact overlapping.

On the other hand, even if the actions

$$x1 > x2 \rightarrow x1, x2 := x2, x1 \quad \text{and}$$
$$x2 > x3 \rightarrow x2, x3 := x3, x2$$

are both enabled, we do not want to execute them in parallel because their simultaneous execution could interfere with each other, as they both access variable $x2$.

Atomicity Constraint

Two actions are said to be in *conflict* with each other if they both refer to a common variable, and one of them may update this variable. More precisely, actions A and B are in conflict with each other, if

$$vA \cap wB \neq \emptyset \quad \text{or} \quad wA \cap vB \neq \emptyset.$$

There is a *write-write* conflict between A and B if $wA \cap wB \neq \emptyset$, and there is a *read-write* conflict between these actions if $rA \cap wB \neq \emptyset$ or $wA \cap rB \neq \emptyset$. Actions that are not in conflict with each other are said to be *independent.*

We permit parallel execution of independent actions. We assume that the actions are started one by one, but in such a way that an enabled action is only started if it does not conflict with any action that is already being executed. We say that a parallel execution of an action system *respects atomicity* if it satisfies this constraint.

Correctness of Parallel Execution

When atomicity is respected, an interleaving semantics is also an appropriate abstraction for parallel execution. Parallel execution of actions gives the same result as a nondeterministic sequential execution. A parallel execution is guaranteed to terminate if and only if the sequential execution is guaranteed to terminate. It will be guaranteed to establish a certain postcondition if and only if the purely sequential execution is guaranteed to establish the same postcondition. We can therefore reason about a parallel execution of a guarded command as if it was a purely sequential execution. As long as the parallel execution of the guarded command respects atomicity, any total correctness results that hold for the purely sequential program will also hold for any parallel execution of it. (Note that we do not assume *fairness* of action system execution.)

Note that even under such a parallel execution regime, there may still be nondeterministic choices that have to be made. This occurs when there are two actions that both become enabled as the result of some other action terminating, but which are in conflict with each other. In this case, we nondeterministically choose either one for execution. Once this action is being executed, it prevents the other action from being started.

8. Parallel Composition of Action Systems

Action systems provide us with a model for parallel program execution. In the same way as procedures provide a mechanism for modularizing sequential programs, we want a mechanism for modularizing parallel programs or action systems. The standard way of achieving this modularization is by parallel composition of process. We follow this same paradigm, considering action systems as processes and defining parallel composition of action systems.

We extend the language of action systems by a *parallel composition* operation:

$\mathcal{A} ::= \ldots \mid \mathcal{A}_1 \parallel \mathcal{A}_2$ {parallel composition}

We define the meaning of parallel composition by syntactic reduction:

begin D; do A od end || begin D'; do A' od end
= begin D; D'; do A [] A' od end.

The parallel composition is defined when the local constant, variable and procedure names in D and D' are all disjoint (so that D; D' is a proper declaration). This can always be achieved by renaming inside the block, prior to forming the composition. Parallel composition thus merges local variables and constants as well as the actions of the two systems.

Any parallel construct may be reduced away by syntactic reduction, resulting in a program without parallel composition. Combining this with the definition of procedures, we can thus reduce any program with parallel composition and procedures to a simple guarded command in our base languge.

Parallel composition is associative and commutative because the ordering of the variable declarations and actions does not affect the meaning of the action system. Hence parallel composition generalizes directly to composition of more than two action systems.

Example
We can describe the sorting program as a parallel composition of two smaller action systems:

```
SORT0" :: LOW0 || HIGH0
  LOW0 :: do $x1 > x2 \rightarrow x1, x2 := x2, x1$     {EX1}
            [] $x2 > x3 \rightarrow x2, x3 := x3, x2$   {EX2}
          od
 HIGH0 :: do $x3 > x4 \rightarrow x3, x4 := x4, x3$     {EX3}
            [] $x4 > x5 \rightarrow x4, x5 := x5, x4$   {EX4}
          od.
```

Carrying out the (trivial) syntactic reduction shows that *SORT*0" = *SORT*0.

Arbitrary Partitioning
The parallel composition of a collection of action systems yields a new action system which is the union of the original systems. Thus parallel composition can be seen as a partitioning of the whole system into a number of parts (action systems), assigning each local variable and procedure to some part. Graphically, parallel composition corresponds to a partitioning of the access graph of an action system into a collection of subgraphs, where each subgraph forms an action system of its own.

An interesting question is now whether every partitioning of an action system (or, equivalently, of the access graph) can be described as a parallel composition of some action systems. In fact, it is easy to see that this is not the case. If we have an action that accesses a given local variable, then we cannot put that action in one part and the local variable in another, because the scope rules for blocks will then prevent the action from accessing the local variable.

In order to permit arbitrary parallel decomposition of action systems, we need to make the scope rules of blocks more permissive. In the next section we show how to do this in a way that permits any partitioning of an action system to be described as a parallel composition of component action systems.

9. Variables and Scopes

We relax the rigid scope rules of blocks by permitting identifiers to be exported from and imported to blocks, thus making these more like *modules* in the Modula-2 or Oberon sense. Besides partitioning local variables among processes, we also partition global variables and constants such as procedures among these. The first allows us to model *persistency* of data, while the second allows modelling *encapsulation* of data and *remote procedure calls* between processes.

The block notation *begin var* $w := w_0$; *S end* wraps a number of different things into a single construct:

(i) It *associates* the variables w with the statement S.

(ii) It *creates* new variables w on entry to the block.

(iii) It *initializes* the variables w to w_0.

(iv) It *hides* the variables w from the environment.

(v) It *destroys* the variables w on exiting the block.

To gain more flexibility in parallel composition, and in order to be able to model distributed systems in a realistic fashion, we need to take these different aspects apart. We do this by generalizing the block construct as described below.

Hiding and Revealing

For parallel composition, it is useful to permit an identifier declared inside one action system to be visible to other action systems in a parallel composition. The other action systems may be given read/write/execute access to these idenifiers.

An identifier x is made visible to the outside by marking it at declaration time: *var* x^* := e indicates that the identifier x can be accessed by the environment. (We could indicate more restricted access if needed, but we consider only general access here for simplicity.) We extend the syntax of declarations as follows:

D ::= ... | D^* {revealing a declaration}

(If more than one identifier is declared in D, then each identifier is understood to be marked.)

Created and Existing Variables

Another distinction that we want to make is that between a variable that is created by an action system and a variable that already exists when the action system starts up. A created variable is preceeded by the key word *var*, while an existing variable does not have this keyword. In the same way as each created variable is associated with a specific action system, we may also associate existing variables with an action system. This permits us to model how variables that exist at startup of an action system are distributed among the parallel components of the action system. We extend the syntax as follows:

D ::= ... | w {existing variables}

where w is a list of variables.

Accessed Variables

We may also need to keep track of the identifiers that are used inside an action system but which are not associated with the action system (i.e., the identifiers that have to be *imported* from other action systems in a parallel composition). We write

$\mathcal{A} : v$

when we want to indicate explicitly the global identifiers v that are used in $\mathcal{A}$ but not associated with $\mathcal{A}$. The list v may contain both variable and constant identifiers.

Generalized Blocks

With these extensions, the block construct itself only means that certain variables are associated with certain statements. Whether these variables are new or existing ones, or whether they are visible or hidden from the environment is indicated separately. For simplicity, in the sequel we assume that the variables that are created at block entry are destroyed at block exit. If needed, we could have a separate notation by which we could create variables that are not destroyed at block exit (or, dually, indicate that some global variables should be destroyed at block exit).

10. Program Modules

The above extensions to the block construct give us a *module description language* where the action systems are the modules and parallel composition is the composition operator for modules. Any partitioning of an action system can be described as a parallel composition of action systems because identifiers that are defined in one module and needed in another module can be made accessible by exporting.

We complete the picture by extending two other useful composition operators, *sequential composition* and *hiding*, to action systems:

```
𝒜 ::= ...
    | $\mathcal{A}_1; \mathcal{A}_2$            {sequential composition}
    | begin $D$; $\mathcal{A}_1$ end    {hiding}
```

Thus action systems can be built out of primitive action systems by using parallel composition, sequential composition and hiding. Again we define the meaning of action system described in this way by syntactic reduction.

Sequential Composition

Sequential composition of action systems corresponds to executing these in *phases*, where each action system describes one phase, to be started only when the previous phase has been completed. We define the meaning of sequential composition by the following reduction rule:

```
  begin $D_1^*$; $E_1$; do $A_1$ od end;
  begin $D_1^*$; $E_2$; do $A_2$ od end
= begin $D_1^*$; $E_1$; $E_2$; var $a$ := 1;
  do $a = 1 \rightarrow A_1$ [] $a = 1 \wedge \neg gA_1 \rightarrow a := 2$
  [] $a = 2 \rightarrow A_2$
  od end.
```

We require that the revealed declarations be the same in both components, whereas the hidden components can be different. The reduction introduces a new local variable a that keeps track of which phase is being executed in the reduced system. Note that $g \rightarrow (h \rightarrow S) = g \wedge h \rightarrow S$ holds in general. In particular, this means that for $A_i = g_i \rightarrow S_i$, the first action in the reduced system is equivalent to the action $a = 1 \wedge g_1 \rightarrow S_1$ and the last action is equivalent to $a = 2 \wedge g_2 \rightarrow S_2$.

Hiding

Hiding corresponds to nested partitioning of the access graph. We define the meaning of hiding by the following reduction rule:

  begin D_1*; E*; F_1;
  begin D_2*; E*; F_2; do A od end end
= begin D_1*; E*; F_1; D_2; F_2; do A od end.

A nested action system is thus defined to be equivalent to the flattened system. The variables that are revealed in the inner block are not revealed by the outer block unless explicitly so required. To re-export a variable to an outer block, we need to repeat the declaration in the outer block. Repeated declarations (E* above) are merged into a single declaration on the syntactic reduction.

Hiding Versus Creating

Hiding/revealing a variable or procedure only makes sense for parallel composition of action systems and provides an abstraction mechanism for modules. Hiding for parallel composition corresponds to creating a new variable for sequential composition: creating a variable on block entry and destroying it on exit means that the variable is not accessible to preceeding or succeeding action systems. A variable that is to be accesible for successive action systems must therefore exist before and after the execution of the action system; it cannot be created inside the action system.

Open and Closed Systems

The scoping rules allow us to distinguish between *open* and *closed* action systems. A closed action system does not reveal any identifiers, nor does it import any, while an open action system either reveals or imports some identifiers.

The sorting program *SORT*0 described previously is an open system where the variables to be sorted are assumed to be associated with the environment (i.e., another action system that is composed in parallel with the sorting action system). It would now be described with explicit mentioning of the imported variables, which are required to be natural numbers:

*SORT*0 :: begin do *EX*1 [] ... [] *EX*4 od end:
  $x1$, $x2$, $x3$, $x4$, $x5$: nat.

The following is also an open version of this system, but now the variables to be sorted are considered part of the action system rather than part of the environment.

*SORT*0' :: begin $x1$*, $x2$*, $x3$*, $x4$*, $x5$* : nat;
  do *EX*1 [] ... [] *EX*4 od end.

Here the variables $x1$, ..., $x5$ could also be accessed by other action systems in a parallel composition.

The sorting program as a closed system hides the variables to be sorted inside the system:

*SORT*1 :: begin $x1$, $x2$, $x3$, $x4$, $x5$: nat;
  do *EX*1 [] ... [] *EX*4 od end.

Note that even if the variables $x1$, ..., $x5$ are hidden, they are assumed to have well-defined values before the action system starts execution, and these variables will remain in the state space after the action system has terminated. Hence we can talk about these variables in pre- and postcondition specifiations, e.g., to state that they work correctly with respect to a given specification.

Nesting of Parallel and Sequential Components

A closed system cannot communicate with its environment during execution, but it can change the state of the system. Hence a closed system behaves as an ordinary sequential statement. We can therefore permit a closed action system to be used as a building block in constructing sequential statements. In particular, a closed action system can be used to construct an action that is part of an action system at a higher level, which in turn may be part of a parallel composition. This gives us nesting of parallel and sequential program constructs to any depth. The nesting is permitted by the following extension of the language:

S ::= ... | $\mathcal{A}$,   {action system}

where $\mathcal{A}$ is a closed action system.

11. Shared Variable Communication

We have shown how to decompose action systems into parallel components, but we have not described how these components communicate with each other. In the following three sections, we show that communication is already built into the approach and is a consequence of the way the action system is partitoned. No new facilities need to be added for communication between processes. We start here by showing how action systems can communicate using shared variables.

Let us partition action system *SORT*1 into parallel components by dividing up the actions. We let process *LOW*2 contain actions *EX*1, *EX*2, and process *HIGH*2 contain the actions *EX*3, *EX*4. We also partition the global variables such that variables $x1$, $x2$ belong to *LOW*2 because they are only accessed by actions in this process, and variables $x4$, $x5$ belong to *HIGH*2 for

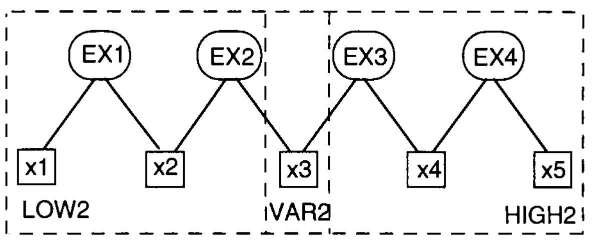

Figure 2. Partitioning of *SORT*2.

similar reasons. The variable $x3$ is shared between the two processes and forms a partition of its own that we call *VAR*2.

The partitioned action system *SORT*2 is then expressed as

*SORT*2 :: begin *LOW*2 ∥ *VAR*2 ∥ *HIGH*2 end,

where

*LOW*2 :: begin $x1, x2$; do *EX*1 [] *EX*2 od end : $x3$
*VAR*2 :: begin $x3$* end
*HIGH*2 :: begin $x4, x5$; do *EX*3 [] *EX*4 od end : $x3$.

Merging these three action systems again gives us the original action system *SORT*1. Note that we omit the empty loop in *VAR*2 for brevity. Figure 2 illustrates the partitioning of *SORT*2.

This partitioning of the actions gives us a *shared variable model* for concurrency. The action systems *LOW*2 and *HIGH*2, considered as processes, can execute in parallel. They communicate by the shared variable $x3$. The atomicity restriction is enforced if we require that variable $x3$ be accessed under mutual exclusion by the two processes. (In general, a correct implementation must also ensure that no deadlock occurs because of the way in which shared variables are reserved).

12. Shared Action Communication

Starting from the variables rather than from the actions gives us a *shared action* model of communication. We place variables $x1, x2, x3$ in action system *LOW*3 and variables $x4, x5$ are in action system *HIGH*3. The actions *EX*1, *EX*2 only access variables in LOW3 and are hence put into this part. Action *EX*4 only accesses variables in *HIGH*3 and is therefore put into this part. Action *EX*3 is shared between the two processes and forms a part of its own. This gives us the following sorting program:

*SORT*3 :: begin *LOW*3 ∥ *ACT*3 ∥ *HIGH*3 end,

where

*LOW*3 :: begin $x1, x2, x3$*; do *EX*1 [] *EX*2 od end
*ACT*3 :: begin do *EX*3 od end : $x3, x4$
*HIGH*3 :: begin $x4$*, $x5$; do *EX*4 od end

Figure3 illustrates the partitioning of *SORT*3.

By expanding the parallel composition, we can easily see that this gives us the same action system as before; i.e., *SORT*3 = *SORT*1.

The action *EX*3 is *shared* in this case because it involves both process. All other actions are *private* to either process. In this partition, the processes thus have disjoint sets of variables, and they communicate by carrying out *shared actions*. The shared action is only executed if each process is ready for it. A shared action can update variables in one or more processes in a way that may depend on the values of variables in other processes. In this way, information is communicated between processes. Note that there is nothing to prevent more than two action systems from participating in a shared action, thus in general providing for synchronized n-way communication [5, 14, 15].

The notion of a shared action is a considerable abstraction here because it is obvious that neither process can determine for itself by only looking at its local variables whether the shared action is enabled or not. Hence some kind of prior communication or centralized scheduling would be needed in order to implement the shared action. Additional restrictions on the action system can be enforced to make the communication mechanism efficiently implementable in a distributed environment [5]. The communication mechanisms of CSP and Occam are special cases of the shared action mechanism described here.

13. Remote Procedure Calls

The shared action model achieves synchronous communication in a parallel composition. However, the variables associated with a certain process are accessed by shared actions, which means that more information about the process is revealed to the environment than we might want. We present here a third communication model which also gives us

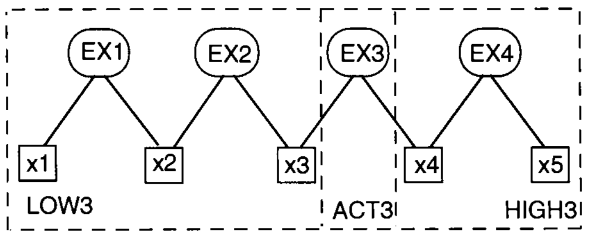

Figure 3. Partitioning of *SORT*3.

synchronous communication but has better locality properties. This method is based on the use of procedures.

An action system $\mathcal{A}$ communicates with another action system $\mathcal{B}$ by calling a procedure exported by $\mathcal{B}$. Information is passed between the processes via the parameters: value parameters send information from $\mathcal{A}$ to $\mathcal{B}$, while result parameters return information in the other direction. The procedure mechanism thus corresponds to *remote procedure call* communication.

The atomicity of an action also includes all procedure invocations that may occur during the action execution. This guarantees that the proof rules for simple iteration statements are still valid when reasoning about the total correctness of action systems with procedure calls.

The body of a procedure is a statement which is always executed whenever the procedure is invoked. Hence the remote procedure mechanim is rather weak, as there is no way in which an action system can refuse to accept a procedure call from another action system. There is, however, a rather simple fix to this: use actions as statements.

The procedure then has an enabledness condition, just as an action in a loop or conditional statement has. This enabledness condition must be satisfied when the procedure is called. If it is not satisfied, then the action from which the procedure has been called is considered not to have been enabled in the first place. A correct implementation of this mechanism has to guarantee that in such a situation the effect of the action is unmade and some other action is executed instead, if possible. Alternatively, we could have a mechanism which can determine in advance whether the action is enabled and prevent execution of an action that would turn out not to be enabled.

Hence an action with procedure calls (within a single action system or between action systems) is either executed to completion without any interference from other actions, or it is not executed at all. A parallel implementation has to respect this atomicity in order to be considered correct.

Example

We illustrate this communication mechanism, again with the sorting example. We partition the action system as follows:

$SORT4$ :: begin $LOW4 \parallel HIGH4$ end,

where

$LOW4$ :: begin $x1, x2, x3$;
proc $swap^*$(upd a) =
$(a > x3 \rightarrow a, x3 := x3, a)$
do $EX1$ [] $EX2$ od end

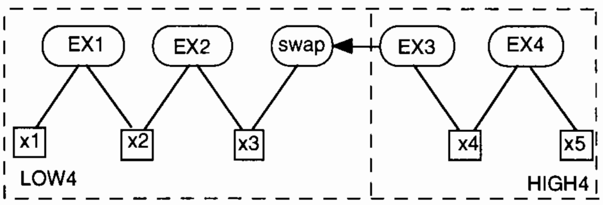

Figure 4. Partitioning of $SORT4$.

$HIGH4$ :: begin $x4, x5$
do $true \rightarrow swap(x4)$ $\{EX3'\}$
[] $EX4$
od end: $swap$.

Thus the action system $LOW4$ reveals the procedure swap that the action system $HIGH4$ calls with the parameter $x4$. The procedure is enabled if $x4 > x3$, in which case the two values are swapped. Note that the enabledness of the action that calls swap is only determined at the calling end (which has the required information). The variables $x1, ..., x5$ are now all hidden. The only way of accessing $x3$ is by using the swap procedure. The outermost brackets in $SORT4$ hide the swap procedure from the environment.

The access relation and partitioning of $SORT4$ is described in Figure 4.

Expanding Parallel Composition

We illustrate the above definitions by simplifying $SORT4$:

$SORT4$
= {reducing parallel composition}
begin begin $x1, x2, x3, x4, x5$;
proc $swap^*$(upd a) = $(a > x3 \rightarrow a, x3 := x3, a)$;
do $EX1$ [] $EX2$ [] $true \rightarrow swap(x4)$ [] $EX4$ od end end
= {expanding procedure call, performing adaption}
begin begin $x1, x2, x3, x4, x5$;
proc $swap^*$(upd) = $(a > x3 \rightarrow a, x3 := x3, a)$;
do $EX1$ [] $EX2$
[] true $\rightarrow$ begin $var\ a := x4 := a > x3 \rightarrow a, x3 := x3, a; x4 := a$ end
[] $EX4$
od end end
= {simplifying expanded call to $EX3$}
begin begin $x1, x2, x3, x4, x5$;
proc $swap^*$(upd a) = $(a > x3 \rightarrow a, x3 := x3, a)$
do $EX1$ [] $EX2$ [] $EX3$ [] $EX4$ od end end
= {flattening nested action system}
begin $x1, x2, x3, x4, x5$;
proc $swap$(upd a) = $(a > x3 \rightarrow a, x3 := x3, a)$;
do $EX1$ [] $EX2$ [] $EX3$ [] $EX4$ od end
= {omitting redundant procedure call}

begin $x1, x2, x3, x4, x5$
do $EX1$ [] $EX2$ [] $EX3$ [] $EX4$ od end
= {definition}
$SORT1$.

The simplification step is justified by the following derivation:

$true \rightarrow$ begin var a := $x4$; $a > x3 \rightarrow a, x3 := x3, a$; $x4 := a$ end
= $true \rightarrow (x4 > x3 \rightarrow x3, x4 := x4, x3)$
= $true \wedge x4 > x3 \rightarrow x3, x4 := x4, x3$
= $x4 > x3 \rightarrow x3, x4 := x4, x3$
= $EX3$.

14. Module Description

Finally, we show how an ordinary modular programming language could be adapted to the action system framework as described above. We choose the programming language Oberon (or Oberon-2) as our base language because it is simple and does not have any features for parallel programming. However, other languages could do as well, such as Modula-2 or Modula-3, C++, Object Pascal, Ada and so on.

We are interested in an actual language implementation of action systems because then we gain a uniform language in which both specifications and implementations of reactive systems can be expressed. Action systems are mainly used for specification of a parallel system. However, having an implemenation of action systems provides us with a tool for observing the behavior of the specified system at an early stage, to reveal inefficiencies in a real parallel implementation and observe other features of interest.

Implementing action systems in Oberon means that a subclass of action systems are efficiently executable in the Oberon system using the Oberon compiler. We want to extend the Oberon language so that more general action systems can be executed as well. Then it is not important that the more general mechanisms are efficiently implemented. We can be satisfied with a considerably less efficient implementation because we are executing just a specification, which has to be turned into a more efficient program before final delivery anyway.

Action Execution

First let us look at simple action systems without parallel or sequential composition, hiding or procedures. The action system

begin $var\ w = w_0$; do A_1 [] ... [] A_m od end : v. (1)

cannot be described directly as an Oberon module because the iteration statement should be executed nondeterministically, but Oberon only has deterministic execution. We could implement the iteration as a deterministic iteration statement, but then this would not be a good model of parallel execution.

The Oberon system does, in fact, already have a mechanism that is very similar to an action system: the *Oberon loop*. The Oberon loop is a circular list of *tasks*, which are parameterless procedures. Two special tasks are standard: listening to mouse and keyboard input and performing garbage collection. The user can also insert own tasks into the loop. The system runs by cycling through the Oberon loop forever. This basic event loop controls interaction with the user.

We can use this mechanism to simulate action execution. We consider an action to be just a parameterless procedure that is inserted into the Oberon loop. As the Oberon system is single-threaded, this means that an action will be executed to completion without any interference from any other action. Hence this mechanism guarantees atomicity of action execution.

However, there are two problems with this implementation. First, the Oberon tasks are always executed; i.e., there is no enabling condition. An action, on the other hand, need not be enabled. We can solve this problem by extending the notion of a procedure so that it can have an enabeling condition. Thus we get the following syntax for actions in an Oberon extension:

ACTION $EX1$;
WHEN $x1 > x2$
BEGIN $x1, x2 := x2, x1$ END $EX1$.

The scheduler needs the explicit guard in order to determine when an action system has terminated. We introduce a new key word for actions (rather than calling these procedures) because actions behave differently from procedures: an action is executed spontaneously by the system, whereas a procedure is only executed if it is explicitly called.

Secondly, scheduling in the Oberon loop is deterministic, whereas we need to have nondeterministic scheduling in order to model parallellism. However, the scheduler for the Oberon loop can be changed so that it cycles in a nondeterministic fashion through the actions in the loop.

Initialization

There are no local variables in our example, so the variable declaration and initialization part of the module are both empty. In addition to assigning initial values for the variables, an implementation of the initialization part of the action systems in Oberon would also have to install the actions of the module in the Oberon loop (if this is not done directly by the compiler). For added flexibility, we may, in fact, permit the initializations in the modules to be arbitrary statements.

Parallel Composition of Action Systems

An Oberon program is just a collection of modules that are initialized one by one. After the initializations, these modules continue to be available, so that they can be manipulated by the tasks in the Oberon loop, and the actions in the modules are executed in parallel, autonomously. Thus parallel composition of an action system is implicit in the collection of modules. The Oberon loop will take care of the execution of the actions of the modules in a nondeterministic fashion which simulates atomicity respecting parallel execution.

Variables and Hiding

The Oberon language already knows about importing and exporting variables, so these aspects do not provide any difficulties. What the language lacks is a facility to distribute existing variables among modules. This means that existing variables have to be assumed to be globally visible and have to be explicitly imported into a module in order to be used.

A facility for taking existing variables into use would correspond to having persistent data in the system. Standard Oberon lacks this feature, but it is present in some extensions of the Oberon system. A restricted form of persistency is available in most programming languages as a facility for binding internal file variabes to external files in a file systems.

The Oberon language does not permit nested module declarations, but other similar languages do, e.g., Modula-2.

Procedures

Procedures as we need them are already present in the Oberon langugage, except that the procedures are always enabled. We can solve this problem syntactically by adding an enabling condition to procedures also, in the same way as we have enabling conditions for actions.

However, this gives us an implementation problem instead. What should we do when we are about to execute a procedure that turns out not to be enabled? The semantics requires us to backtrack to the action from which the procedure originally was called and choose some othe action instead. Restoring the system state to what it was before the action execution can be a very costly operation. However, we may use this as the general strategy, and then assume that the compiler is smart enough to detect the situations when more efficient implementations can be done.

Example
The following is the Oberon equivalent of action system *SORT*4 (the *USE* clause models the placement of existing variables in a module):

```
MODULE LOW4;
   USE x1, x2, x3 : INTEGER;
PROC swap*(VAR a: INTEGER);
   WHEN a > x3 BEGIN a, x3 := x3, a END swap;
ACTION EX1;
   WHEN x1 > x2 BEGIN x1, x2 := x2, x1
                                        END EX1;
ACTION EX2;
   WHEN x2 > x3 BEGIN x2, x3 := x3, x2
                                        END EX2;
BEGIN END LOW4.

MODULE HIGH4;
   IMPORT LOW4;
   USE x4, x5: INTEGER;
ACTION EX3;
   BEGIN LOW4.swap(x4) END EX3;
ACTION EX4;
   WHEN x4 > x5 BEGIN x4, x5 := x5, x4
                                        END EX4;
BEGIN END HIGH4.
```

15. Conclusions

We have shown how to extend a simple guarded command language with modularization features that are considered useful and necessary in the construction of large programs. The meaning of these modularization constructs is defined by syntactic reduction, showing how each construct can be removed by syntactic transformations that reduces a program containing the construct to one that does not contain it. Ultimately, we can reduce a large modularized program to a simple guarded command.

Another purpose of the treatment here has been to show that the conceptual basis required for the main modularization facilities is actually rather small. Most properties needed are already present in the basic language of guarded commands with blocks. The modularization facilities introduced here are not restricted to programming languages, but work equally well for specification languages. In fact, many of the constructs that we have defined are such that they cannot be implemented efficiently in their full generality. Hence what we really have is a specification language with modularization facilities. A programming language should be seen as a restricted subset of the specification language where the restictions are such that they facilitate efficient execution of programs written in the language.

The reduction rules introduced in this paper are intended to provide justification for module refinement rules. For each kind of module that we have defined (procedure, process, data module), we can give conditions under which the replacement of this module with another module is correct in the sense that it preserves the correctness of whole program. Justifying these refinement rules requires the reduction rules.

The logical basis for program refinement that we use is the *refinement calculus*, originally described by Back [1, 2] as a formal framework for stepwise refinement of sequential programs. This calculus extends Dijkstra's weakest precondition semantics [13] for total correctness of programs with a relation of refinement between program statements. A good overview of how to apply the refinement calculus in practical program derivations is given by Morgan [19].

Because action systems can be seen as special kinds of sequential systems, the refinement calculus framework carries over to the refinement of parallel systems in [8]. Reactive system refinement is handled by existing techniques for data refinement of sequential programs within the refinement calculus [3].

We have not described the methods by which modularity is used in program refinement, nor have we given specific refinement rules for modular components. These issues have been treated in other papers, and the interested reader can consult them for more details [9, 10, 11, 7]. The features of our modular programming language have been chosen so that refinement of program modules is simple and straightforward.

Acknowledgments: The work reported here was supported by the Academy of Finland. We want to thank Michael Butler, Eric Hedman, Xu Qiwen, Marina Waldén and Joakim von Wright for comments and discussions on the modularization issues described here.

References

1. Back R (1980) Correctness Preserving Program Refinements: Proof Theory and Applications. Mathematical Center Tracts, Vol 131, Mathematical Centre, Amsterdam
2. Back R (1988) A Calculus of Refinements for Program Derivations. Acta Informatica, 25:593–624
3. Back R (1989) Refinement Calculus, Part II: Parallel and Reactive Programs. In REX Workshop for Refinement of Distributed Systems, Lecture Notes in Computer Science, Vol. 430, Nijmegen, The Netherlands, Springer-Verlag
4. Back R (June 1989) Refining Atomicity in Parallel Algorithms. In PARLE Conference on Parallel Architectures and Languages, Europe, Eindhoven, the Netherlands, Springer-Verlag
5. Back R, Kurki-Suonio R (1983) Decentralization of Process Nets with Centralized Control. In 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, 131–142, ACM
6. Back R, Kurki-Suonio R (October 1988) Distributed Cooperation with Action Systems. ACM Transactions on Programming Languages and Systems, 10:513–554
7. Back RJR, Martin AJ, Sere K (1995) An Action System Specification of the Caltech Asynchronous Microprocessor. In Mathematics of Program Construction 95, Lecture Notes in Computer Science, Vol. 947, Kloster Irsee, Germany, Springer–Verlag
8. Back R, Sere K (1991) Stepwise Refinement of Action Systems. Structured Programming, 12:17–30, Springer-Verlag
9. Back RJR, Sere K (1994) Action Systems with Synchronous Communication. In Olderog ER, Editor, IFIP TC 2 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94), 107-126. Elsevier
10. Back R, von Wright J (1989) Refinement Calculus, Part I: Sequential Programs. In REX Workshop for Refinement of Distributed Systems, Lecture Notes in Computer Science, Vol. 430, Nijmegen, The Netherlands, Springer-Verlag
11. Back R, von Wright J (August 1994) Trace Refinement of Action Systems. In Proc. of CONCUR '94, Uppsala, Sweden
12. Chandy K, Misra J (1988) Parallel Program Design: A Foundation. Addison-Wesley
13. Dijkstra E (1976) A Discipline of Programming. Prentice Hall International
14. Francez N (September 1989) Cooperating Proofs for Distributed Programs with Multiparty Interaction. Information Processing Letters, 32:235–242, North-Holland
15. Francez N, Hailpern BT, Taubenfeld G (1986) SCRIPT—A Communication Abstraction Mechanism and Its Verification. Science of Computer Programming, 6:35–88
16. Hoare CAR (August 1978) Communicating Sequential Processes. Communications of the ACM, 21(8):666–677
17. INMOS Ltd. (1984) Occam Programming Manual. Prentice Hall International
18. Morgan CC (January 1988) Data Refinement by Miracles. Information Processing Letters 26:243–246
19. Morgan C (1990) Programming from Specifications. Prentice-Hall
20. Morris JM (1987) A Theoretical Basis for Stepwise Refinement and the Programming Calculus. Science of Computer Programming, 9:287–306
21. Nelson G (October 1989) A Generalization of Dijkstra's Calculus. ACM Transactions on Programming Languages and Systems, 11(4):517–562
22. United States Department of Defence. Reference Manual for the Ada Programming Language. ANSI/MIL–STD–1815A–1983, American National Standards Institute, New York
23. Wirth N (1988) The Programming Language Oberon. Software—Practice and Experience, 18(7)