A Theory of Prioritising Composition*

Emil Sekerinski† Kaisa Sere‡

Abstract

An operator for the composition of two processes, where one process has priority over the other process, is studied. Processes are described by action systems, and data refinement is used for transforming processes. The operator is shown to be compositional, i.e. monotonic with respect to refinement. It is argued that this operator is adequate for modelling priorities as found in programming languages and operating systems. Rules for introducing priorities and for raising and lowering priorities of processes are given. Dynamic priorities are modelled with special priority variables which can be freely mixed with other variables and the prioritising operator in program development. A number of applications show the use of prioritising composition for modelling and specification in general.

1 Introduction

Priorities in concurrent computations is a concept found in various programming languages and operating systems. We develop a theory of priorities, in which priorities have a "logical" meaning: a process of a certain priority cannot be interrupted by a process of a lower priority, but will be interrupted if a process of a higher priority becomes ready. For example, interrupt priorities and priorities of Modula-2 modules (monitors) belong to this class. By contrast, priorities of UNIX processes affect only the proportional allocation of processor time. Also, in occam assigning priorities belongs to "configuration" which comes after the logical design [17]. In these situations priorities are only hints to the scheduler, and the programmer is not supposed to rely on any logical effect.

*Published in The Computer Journal, Volume 39, Issue 8, pp. 701-712.

†Department of Computer Science, Åbo Akademi University, Lemminkäisenkatu 14, 20520 Turku, Finland, Emil.Sekerinski@abo.fi

‡Deptartment of Computer Science and Applied Mathematics, University of Kuopio, P.O.Box 6711, 70211 Kuopio, Finland, Kaisa.Sere@uku.fi

The core of our theory is an operator for prioritising composition of processes described as action systems, and an operator for prioritising composition of individual actions. An action system is a set of guarded commands, or actions, sharing local variables [3]. Action systems model parallelism by interleaving of atomic actions.

Basic algebraic properties of both forms of prioritising composition are presented (Sec. 3). The main theoretical result is that prioritising composition of action systems is compositional in the sense that it is monotonic with respect to refinement of its operands under the same circumstances that parallel composition is (Sec. 4). A specification may already contain priorities, or priorities may be introduced by a refinement step using a rule for prioritising decomposition (Sec. 5). Furthermore, rules are given for raising and lowering the priority of processes (Sec. 6). The use of prioritising composition is illustrated for resource sharing, overriding behaviour, modelling control systems, timed action systems, and various forms of scheduling by dynamic priorities (Sec. 7). These applications show the usefulness of prioritising composition for modelling and specification in general, not only as a programming language construct.

The prioritising composition of action systems is similar to prioritising parallel composition *PRI PAR* of occam, and the prioritising composition of actions is similar to the *PRI ALT* construct. As in the prioritising composition of action systems no parallelism is possible between the action systems of different priorities, we simply call it prioritising composition rather than prioritising parallel composition.

Our treatment of (static) priorities is purely based on the enabledness of actions: if one of two (or more) enabled actions is to be chosen for execution, the one with the higher priority is given preference. For this, we assume only that the behaviour of actions is characterised by the weakest precondition predicate transformer, which yields the enabledness of actions. Hence, we expect that the results are generalisable to related models of parallelism, like Unity [9] or the Temporal Logic of Actions [15], which also define enabledness of actions, but additionally consider fairness. Since fairness only influences nondeterministic choice, but not a choice based on priorities, fairness is independent of priorities, and is not considered here.

Priorities have been included in various process algebras [7, 10, 20]. The approaches vary slightly and some authors introduce prioritising operators between actions while others give priorities to actions themselves. The approaches that come closest to ours are those of Fidge [12] and Lowe [16] who define prioritising operators for CSP. The work by Fidge is directed towards giving a formal basis for the prioritising constructs in Ada and occam, whereas Lowe describes a deterministic model for timed CSP extended with prioritising operators on actions. It has also been noted in the context of process algebras that a prioritising composition

can be understood as the extreme case of a probabilistic choice [23, 24]. Here we study prioritising composition in a simple model without introducing probabilities.

Our treatment of priorities has its foundations in a state based rather than event based formalism: the basic communication mechanism between action systems are shared variables. This leads to a style of reasoning as for imperative programs. In particular, it allows for data refinement of action systems, a useful technique for distributing action systems [2], and the main subject of the study here. The works on process algebras mentioned above focus on extending the semantic basis for dealing with the language constructs, and do not treat methodological aspects of the introduced constructs or refinement. In our case, the underlying action systems theory is already rich enough to support these new operators.

2 Action Systems

An action system $\mathcal{A}$ is a set of actions operating on local and global variables:

$$|[\ \mathbf{var}\ a \mid A_0 \bullet \mathbf{do}\ A_1 \ [\!]\ \ldots\ [\!]\ A_n\ \mathbf{od}\ ]|$$

Action system $\mathcal{A}$ describes a computation, in which local variables a are first created and initialised such that predicate A_0 holds. 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 global variables of $\mathcal{A}$ are those mentioned in $A_0, A_1, \ldots, A_n$ but not declared locally.

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 [11]: $wp(A, p)$ is the weakest precondition such that action A terminates in a state satisfying predicate p. In addition to the statements considered by Dijkstra, we allow assumptions $[q]$, where q is a predicate, and nondeterministic choice $A\ [\!]\ B$ between actions A, B. The assumption $[q]$ can be thought of as stopping execution if q does not hold.

$$\begin{array}{lcllcl} wp(abort, p) & = & false & wp([q], p) & = & q \Rightarrow p \\ wp(skip, p) & = & p & wp((A\ [\!]\ B), p) & = & wp(A, p) \wedge wp(B, p) \\ wp(v := e, p) & = & p[v := e] & wp((A\,;\,B), p) & = & wp(A, wp(B, p)) \end{array}$$

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

$$wp(A, p \wedge q) \quad = \quad wp(A, p) \wedge wp(A, q) \tag{1}$$

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

$$(p \Rightarrow q) \quad \Rightarrow \quad (wp(A,p) \Rightarrow wp(A,q)) \tag{2}$$

Action A preserves predicate p if, under the assumption that p holds initially and A does terminate (does not abort), A establishes p. The predicate $wp(A, true)$ is true in those states in which A does not abort:

$$A \text{ preserves } p \qquad \text{iff} \qquad p \wedge wp(A, true) \quad \Rightarrow \quad wp(A,p)$$

A variable is said to be assigned by action A if it occurs on the left hand side of an assignment in A. Another property we require all actions to satisfy is following noninterference condition.

$$A \text{ preserves } r \qquad \text{if free variables of } r \text{ are not assigned in } A \tag{3}$$

All of the above operators satisfy this noninterference property or preserve it.

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:

$$A = B \qquad \text{iff} \qquad \text{for all } p: \quad wp(A,p) = wp(B,p)$$

An action that establishes any postcondition is said to be miraculous. We take the view that an action is only enabled in those initial states in which it behaves nonmiraculously. The guard of an action characterises those states for which the action is enabled:

$$gd\,A \quad = \quad \neg\, wp(A, false)$$

For example, we have that *abort*, *skip*, and $v := e$ are always enabled. The nondeterministic choice $A \,[\!]\, B$ is enabled when either A or B is enabled:

$$gd\,(A \,[\!]\, B) \quad = \quad gd\,A \vee gd\,B \tag{4}$$

The guarded action $g \rightarrow A$ is defined using assumption as follows:

$$g \rightarrow A \quad = \quad [g]\,;A$$

The guarded action $g \rightarrow A$ is only enabled when g holds and A is enabled:

$$gd\,(g \rightarrow A) \quad = \quad g \wedge gd\,A \tag{5}$$

As the nondeterministic choice $A \,[\!]\, B$ is included as an operator on actions, we can confine ourselves to action systems with only a single action. Furthermore, declared variables are either private or can be made public by marking them with a

star ($*$). Public variables are initialised and possibly modified in the action system they are declared in, but can also be read by other action systems. Hence an action system $\mathcal{A}$ is in general of the form:

$$|[\ \mathbf{var}\ a, x* \mid A_0 \bullet \mathbf{do}\ A\ \mathbf{od}\ ]|$$

An action system is formally a pair with the first component describing the initial values of the local and public variables and the action A as the second component.

An action system can also be associated with a set of behaviours. Let u be the list of global variables of $\mathcal{A}$, which also includes the public variables. A behaviour is a sequence of states with components for both private and global variables, i.e. of the form $\langle (a_0, u_0), (a_1, u_1), \ldots \rangle$. The first element of each behaviour has to satisfy the initialisation predicate A_0 and all consecutive pairs of elements have to be according to A in the sense that:

$$\begin{array}{lcl} ((a,u) = (a_0,u_0)) & \Rightarrow & A_0 \\ ((a,u) = (a_i,u_i)) & \Rightarrow & \neg wp(A, \neg((a,u) = (a_{i+1},u_{i+1}))) \end{array}$$

Note that $wp(A,p)$ characterises those states in which A is guaranteed to establish p, whereas $\neg wp(A,\neg p)$ characterises those states in which A may establish p. Behaviours are either finite or infinite. Finite ones are either *terminating* or *aborting*. They are terminating if they end with an element (a_n,u_n) satisfying $((a,u) = (a_n,u_n)) \Rightarrow \neg\, gd\, A$. Finite ones are aborting if they end with an element satisfying $((a,u) = (a_n,u_n)) \Rightarrow \neg\, wp(A, true)$.

Both views of action systems are described by Back and von Wright [6]. Our use of the behavioural view is to justify data refinement of action systems (Sec. 4).

3 Prioritising and Parallel Composition

We start by defining the prioritising composition of actions, and then considering the parallel and prioritising composition of action systems.

Let A, B, C be actions. The prioritising composition $A \mathbin{//} B$ selects the first operand if it is enabled, otherwise the second, the choice being deterministic.

$$A \mathbin{//} B \quad = \quad A \mathbin{[\!]} (\neg gd\, A \rightarrow B)$$

Since $A = gd\, A \rightarrow A$, the above definition can be equivalently stated as:

$$A \mathbin{//} B \quad = \quad (gd\, A \rightarrow A) \mathbin{[\!]} (\neg gd\, A \rightarrow B) \tag{6}$$

The prioritising composition of two actions is enabled if either operand is.

$$gd(A \mathbin{//} B) \quad = \quad gd\, A \vee gd\, B \tag{7}$$

Prioritising composition of actions is associative, allowing parentheses to be omitted in repeated applications.

$$(A \mathbin{/\!/} B) \mathbin{/\!/} C \quad = \quad A \mathbin{/\!/} (B \mathbin{/\!/} C) \tag{8}$$

Proof. We transform the left hand side into the right hand side:

$$\begin{array}{ll} & (A \mathbin{/\!/} B) \mathbin{/\!/} C \\ = & \{\text{definition of } \mathbin{/\!/}\} \\ & (A \mathbin{/\!/} B) \mathbin{[\!]} \neg gd(A \mathbin{/\!/} B) \to C \\ = & \{\text{definition of } \mathbin{/\!/}, (7)\} \\ & A \mathbin{[\!]} \neg gd\, A \to B \mathbin{[\!]} \neg gd\, A \wedge \neg gd\, B \to C \\ = & \{\to \text{ distributes over } \mathbin{[\!]} \text{ to the right}\} \\ & A \mathbin{[\!]} \neg gd\, A \to (B \mathbin{[\!]} \neg gd\, B \to C) \\ = & \{\text{definition of } \mathbin{/\!/}, \text{twice}\} \\ & A \mathbin{/\!/} (B \mathbin{/\!/} C) \end{array}$$ □

Prioritising composition of actions distributes over choice to the right.

$$A \mathbin{/\!/} (B \mathbin{[\!]} C) \quad = \quad (A \mathbin{/\!/} B) \mathbin{[\!]} (A \mathbin{/\!/} C) \tag{9}$$

Proof.

$$\begin{array}{ll} & A \mathbin{/\!/} (B \mathbin{[\!]} C) \\ = & \{\text{definition of } \mathbin{/\!/}, \to \text{ distributes over } \mathbin{[\!]} \text{ to the right}\} \\ & A \mathbin{[\!]} \neg gd\, A \to B \mathbin{[\!]} \neg gd\, A \to C \\ = & \{\ \mathbin{[\!]} \text{ idempotent, definition of } \mathbin{/\!/}\ \} \\ & (A \mathbin{/\!/} B) \mathbin{[\!]} (A \mathbin{/\!/} C) \end{array}$$ □

Prioritising composition does not distribute over choice to the left in general. To see this, consider the enabledness of the action C in $(A \mathbin{[\!]} B) \mathbin{/\!/} C$ and $(A \mathbin{/\!/} C) \mathbin{[\!]} (B \mathbin{/\!/} C)$. In the first case, C can be selected only if both A and B are disabled. In the second case, C can be selected if either A or B is disabled.

Let $\mathcal{A}$ and $\mathcal{B}$ be action systems given by:

$$\begin{array}{lcl} \mathcal{A} & = & |[\ \mathbf{var}\ a, x* \mid A_0 \bullet \mathbf{do}\ A\ \mathbf{od}\ ]| \\ \mathcal{B} & = & |[\ \mathbf{var}\ b, y* \mid B_0 \bullet \mathbf{do}\ B\ \mathbf{od}\ ]| \end{array}$$

We assume that the variable lists a and b are disjoint, which can always be achieved by renaming. The parallel composition $\mathcal{A} \parallel \mathcal{B}$ joins the local variables and their

initialisation and merges the actions by nondeterministic choice, without any assumption of fairness in case both are enabled [2].

$$\mathcal{A} \parallel \mathcal{B} \quad = \quad |[\ \textbf{var}\ a, b, x*, y* \mid A_0 \wedge B_0 \bullet \textbf{do}\ A \ [\!]\ B\ \textbf{od}\ ]|$$

The prioritising composition $\mathcal{A} /\!\!/ \mathcal{B}$ combines $\mathcal{A}$ and $\mathcal{B}$ in a way that preference is given to the action of $\mathcal{A}$. The choice between the action of $\mathcal{A}$ and $\mathcal{B}$ is deterministic in the sense that when both are enabled, the action of $\mathcal{A}$ is taken.

$$\mathcal{A} /\!\!/ \mathcal{B} \quad = \quad |[\ \textbf{var}\ a, b, x*, y* \mid A_0 \wedge B_0 \bullet \textbf{do}\ A /\!\!/ B\ \textbf{od}\ ]|$$

Next we investigate the algebraic properties of prioritising composition. Let $\mathcal{C}$ be an action system, given by:

$$\mathcal{C} \quad = \quad |[\ \textbf{var}\ c, z* \mid C_0 \bullet \textbf{do}\ C\ \textbf{od}\ ]|$$

Prioritising composition of action systems is associative (like parallel composition), allowing us to omit parentheses in repeated applications.

$$(\mathcal{A} /\!\!/ \mathcal{B}) /\!\!/ \mathcal{C} \quad = \quad \mathcal{A} /\!\!/ (\mathcal{B} /\!\!/ \mathcal{C}) \tag{10}$$

Proof. The theorem follows from the definition of $/\!\!/$ and (8). □

We use indexed action systems to denote sets of action systems. In the action system

$$\mathcal{A} \quad = \quad \mathcal{B}_1 /\!\!/ \mathcal{B}_2 /\!\!/ \mathcal{B}_3 /\!\!/ \ \ldots$$

$\mathcal{B}_1$ has priority over $\mathcal{B}_2$, which in turn has priority over $\mathcal{B}_3$, and so on. Hence this models a hierarchy of priorities. In a system with several processes, there may be classes of processes running with the same priority. This corresponds to action systems $\mathcal{B}_i$ being of the form:

$$\mathcal{B}_i \quad = \quad \mathcal{C}_{i,1} \parallel \mathcal{C}_{i,2} \parallel \mathcal{C}_{i,3} \parallel \ \ldots$$

Here $\mathcal{C}_{i,j}$ correspond to the individual processes and $\mathcal{B}_i$ correspond to the priority classes. In a typical programming environment the priority of a process would be given by a number, rather than by constructing a large expression of the above form. This number determines the class $\mathcal{B}_i$ to which the process belongs. This translation shows that the numbers themselves are irrelevant, only the order they induce is relevant. Indeed, our theory of priorities is based solely on expressions of the above form, but is applicable to programming environments where priorities are determined by numbers.

Let $\mathcal{G}$ be an action system without local variables, i.e. of the form:

$$\mathcal{G} \quad = \quad |[\ \textbf{do}\ G\ \textbf{od}\ ]|$$

Prioritising composition with an action system without local variables distributes over parallel composition to the right:

$$\mathcal{G} \mathbin{/\!\!/} (\mathcal{A} \parallel \mathcal{B}) \quad = \quad (\mathcal{G} \mathbin{/\!\!/} \mathcal{A}) \parallel (\mathcal{G} \mathbin{/\!\!/} \mathcal{B}) \tag{11}$$

Proof. The theorem follows from the definitions of $/\!\!/$ and $\parallel$ and (9). □

This theorem would not hold if $\mathcal{G}$ had local variables, since they would get duplicated on the right hand side. Prioritising composition does not distribute over parallel composition to the left in general, for the same reason that prioritising composition of actions does not distribute over choice to the left.

4 Refinement

In this section we study different notions of refinement for action systems. We adapt the work of Back and von Wright [6] to handle the prioritising composition operators on actions and action systems.

Action A is refined by action C, written $A \leq C$, if, whenever A establishes a certain postcondition, so does C:

$$A \leq C \quad \text{iff} \quad \text{for all } p: \quad wp(A,p) \Rightarrow wp(C,p)$$

Together with the monotonicity of *wp* this implies that for a certain precondition, C might establish a stronger postcondition than A (reduce nondeterminism of A) or even establish postcondition *false* (behave miraculously). Choice ($[\!]$) and sequential composition (;) are both monotonic with respect to refinement in both operands. Prioritising composition is monotonic in its left operand only if its guard is not strengthened by the refinement:

$$A \leq C \quad \Rightarrow \quad A \mathbin{/\!\!/} B \leq C \mathbin{/\!\!/} B \qquad \text{if} \quad gd\,A \Rightarrow gd\,C \tag{12}$$

Proof. We transform the right hand side of the implication:

$$\begin{array}{ll} & A \mathbin{/\!\!/} B \leq C \mathbin{/\!\!/} B \\ \Leftrightarrow & \{\text{definition of } /\!\!/ \text{ and } \rightarrow\} \\ & A \mathbin{[\!]} [\neg gd\,A]\,;\, B \leq C \mathbin{[\!]} [\neg gd\,C]\,;\, B \\ \Leftarrow & \{\text{monotonicity of } [\!] \text{ and } ;\text{, assumption } A \leq C\text{, reflexivity of } \leq\} \\ & [\neg gd\,A] \leq [\neg gd\,C] \\ \Leftrightarrow & \{\text{for } p,q: [p] \leq [q] \text{ iff } q \Rightarrow p\text{, assumption } gd\,A \Rightarrow gd\,C\} \\ & \mathit{true} \end{array}$$

□

However, prioritising composition is always monotonic in its right operand:

$$A \leq C \quad \Rightarrow \quad B \mathbin{//} A \leq B \mathbin{//} C \tag{13}$$

Proof. The theorem follows from the definitions of $//$ and $\rightarrow$ and the monotonicity of $[]$ and $;$. □

A variation of refinement is if A is (data-) refined by C via a relation R, written $A \leq_R C$. For this, assume A operates on variables a, u and C operates on variables c, u. Let R be a predicate over a, c, u:

$$A \leq_R C \qquad \text{iff} \qquad \text{for all } p: \quad R \wedge wp(A, p) \Rightarrow wp(C, (\exists\, a \bullet R \wedge p))$$

When using the notation $A \leq_R C$, the variables a and c are assumed to be understood from the context. Data refinement of choice and sequential composition can be carried out piecewise: if $A \leq_R C$ and $B \leq_R D$, then $A \,[]\, B \leq_R C \,[]\, D$ and $A ; B \leq_R C ; D$. Data refinement of prioritising composition additionally requires that the guard of the left operand is not strengthened under the refinement relation:

$$(A \leq_R C) \wedge (B \leq_R D) \quad \Rightarrow \quad A \mathbin{//} B \leq_R C \mathbin{//} D \quad \text{if} \quad R \wedge gd\,A \Rightarrow gd\,C \tag{14}$$

Proof. We transform the right hand side of the implication:

$$\begin{array}{ll} & A \mathbin{//} B \leq_R C \mathbin{//} D \\ \Leftrightarrow & \{\text{definition of } // \text{ and } \rightarrow\} \\ & A \,[]\, [\neg gd\,A] ; B \leq_R C \,[]\, [\neg gd\,C] ; D \\ \Leftarrow & \{\text{piecewise refinement of } [] \text{ and } ;\} \\ & (A \leq_R C) \wedge ([\neg gd\,A] \leq_R [\neg gd\,C]) \wedge (B \leq_R D) \\ \Leftrightarrow & \{\text{assumptions } A \leq_R C \text{ and } B \leq_R D, \text{ for } p, q : [p] \leq_R [q] \text{ iff } R \wedge q \Rightarrow p\} \\ & R \wedge \neg gd\,C \Rightarrow \neg gd\,A \\ \Leftrightarrow & \{\text{assumption } R \wedge gd\,A \Rightarrow gd\,C\} \\ & \mathit{true} \end{array}$$

□

Data refinement laws for assignments and other operators, especially calculational aspects of data refinement, have been studied by Back [1], Morgan [18] and Morris [19]. The following law gives the condition under which an action B is refined via R by itself. It is applicable when B does not assign any variables that are refined using R.

$$B \text{ preserves } R \quad \Rightarrow \quad B \leq_R B \tag{15}$$

Proof. For any p, we have for the right hand side:

$$
\begin{array}{ll}
 & B \leq_R B \\
\Leftrightarrow & \{\text{definition of } \leq_R\} \\
 & R \wedge wp(B,p) \Rightarrow wp(B,(\exists\, a \bullet R \wedge p)) \\
\Leftarrow & \{\text{for any } q\text{: } q \Rightarrow (\exists\, a \bullet q)\text{, monotonicity}\} \\
 & R \wedge wp(B,p) \Rightarrow wp(B,R \wedge p) \\
\Leftarrow & \{\text{conjunctivity, logic}\} \\
 & R \wedge wp(B,p) \Rightarrow wp(B,R) \\
\Leftarrow & \{\text{monotonicity, logic}\} \\
 & R \wedge wp(B,\mathit{true}) \Rightarrow wp(B,R) \\
\Leftrightarrow & \{B \text{ preserves } R\} \\
 & \mathit{true}
\end{array}
$$

□

Data refinement allows the local variables of an action system to be replaced. Assume that action systems $\mathcal{A}$ and $\mathcal{C}$ are of the form

$$
\begin{array}{lcl}
\mathcal{A} & = & |[\ \mathbf{var}\ a,w,x* \mid A_0 \bullet \mathbf{do}\ A\ \mathbf{od}\ ]| \\
\mathcal{C} & = & |[\ \mathbf{var}\ c,w,x* \mid C_0 \bullet \mathbf{do}\ C\ \mathbf{od}\ ]|
\end{array}
$$

where lists a and c are disjoint. Let R be a predicate over a,c,w,x. Action system $\mathcal{A}$ is (data-) refined by $\mathcal{C}$ via R, written $\mathcal{A} \leq_R \mathcal{C}$, if

(a) *initialisation*: $C_0 \Rightarrow (\exists\, a \bullet R \wedge A_0)$,

(b) *main action*: $A \leq_R C$,

(c) *exit condition*: $R \wedge gd\,A \Rightarrow gd\,C$.

Again, when using the notation $\mathcal{A} \leq_R \mathcal{C}$, the variables a and c are assumed to be understood from the context. The existence of the relation R ensures that the observable behaviour of $\mathcal{C}$, in terms of its traces, satisfies that of $\mathcal{A}$: A trace $tr\,s$ is obtained from a behaviour s by deleting the local component of each element of s. Trace refinement $\mathcal{A} \sqsubseteq \mathcal{C}$ means that every trace $tr\,t$ of $\mathcal{C}$ approximates some trace $tr\,s$ of $\mathcal{A}$ in the sense that either $tr\,t = tr\,s$ or s is aborting and $tr\,t$ is a prefix of $tr\,s$. As shown by Back and von Wright [6], data refinement $\mathcal{A} \leq_R \mathcal{C}$ implies trace refinement $\mathcal{A} \sqsubseteq \mathcal{C}$.

On action systems, an operator $F(\mathcal{A})$ is said to be compositional if when refining $\mathcal{A}$ to $\mathcal{C}$, then $F(\mathcal{A})$ is refined by $F(\mathcal{C})$ as well. Under the above restriction that

the data refinement relation is only over the variables a, c, w, x, parallel composition is compositional in the sense that:

$$\mathcal{A} \leq_R \mathcal{C} \quad \Rightarrow \quad \mathcal{A} \parallel \mathcal{B} \leq_R \mathcal{C} \parallel \mathcal{B} \tag{16}$$

Proof. The right hand side expands to:

$$\begin{array}{l} |[\ \textbf{var}\ a, w, x*, b, y* \mid A_0 \wedge B_0 \bullet \textbf{do}\ A \,[\!]\, B\ \textbf{od}\ ]| \quad \leq_R \\ \quad |[\ \textbf{var}\ c, w, x*, b, y* \mid C_0 \wedge B_0 \bullet \textbf{do}\ C \,[\!]\, B\ \textbf{od}\ ]| \end{array}$$

According to the definition of $\leq_R$ on action systems, three conditions have to be checked. For *initialisation* we have:

$$\begin{array}{ll} & C_0 \wedge B_0 \Rightarrow (\exists\, a \bullet R \wedge A_0 \wedge B_0) \\ \Leftrightarrow & \{a \text{ not free in } B_0\} \\ & C_0 \wedge B_0 \Rightarrow (\exists\, a \bullet R \wedge A_0) \wedge B_0 \\ \Leftarrow & \{\text{condition (a) of } \mathcal{A} \leq_R \mathcal{C}\} \\ & \mathit{true} \end{array}$$

For *main action* we have:

$$\begin{array}{ll} & A \,[\!]\, B \leq_R C \,[\!]\, B \\ \Leftarrow & \{\text{piecewise refinement of } [\!]\ \} \\ & (A \leq_R C) \wedge (B \leq_R B) \\ \Leftrightarrow & \{\text{condition (b) of } \mathcal{A} \leq_R \mathcal{C}\} \\ & (B \leq_R B) \\ \Leftarrow & \{(15)\} \\ & B \text{ preserves } R \\ \Leftrightarrow & \{a, w, x \text{ not assigned in } B, (3)\}\ (*) \\ & \mathit{true} \end{array}$$

For *exit condition* we have:

$$\begin{array}{ll} & R \wedge (gd\, A \vee gd\, B) \Rightarrow (gd\, C \vee gd\, B) \\ \Leftarrow & \{\text{condition (c) of } \mathcal{A} \leq_R \mathcal{C}\} \\ & \mathit{true} \end{array}$$

□

If R also ranged over global variables assigned in B, then the step (*) in the proof would be only valid under the additional noninterference condition that B preserves R. This restriction of the refinement relation is similar to the strong simulation of

Back [2], except that we allow R to depend on (but not change) the public variables (which are considered there) as well. This implies that if a variable is written by several processes, it has to be declared global to them and cannot be used in any local refinement of a process. If a variable is assigned by just one process, it can be made a public variable of that process and a refinement relation of that process may depend on it.

Because of its symmetry, parallel composition is monotonic in both arguments. Prioritising composition is also monotonic in both arguments:

$$\mathcal{A} \leq_R \mathcal{C} \quad \Rightarrow \quad \mathcal{A} /\!\!/ \mathcal{B} \leq_R \mathcal{C} /\!\!/ \mathcal{B} \tag{17}$$

$$\mathcal{A} \leq_R \mathcal{C} \quad \Rightarrow \quad \mathcal{B} /\!\!/ \mathcal{A} \leq_R \mathcal{B} /\!\!/ \mathcal{C} \tag{18}$$

Proof. The right hand side of (17) expands to:

$$\begin{array}{l} |[\ \mathbf{var}\ a, w, x*, b, y* \mid A_0 \wedge B_0 \bullet \mathbf{do}\ A /\!\!/ B\ \mathbf{od}\ ]| \quad \leq_R \\ \quad |[\ \mathbf{var}\ c, w, x*, b, y* \mid C_0 \wedge B_0 \bullet \mathbf{do}\ C /\!\!/ B\ \mathbf{od}\ ]| \end{array}$$

Three conditions have to be checked. The *initialisation* condition and its proof are identical to the previously given one. For *main action* we have:

$$\begin{array}{ll} & A /\!\!/ B \leq_R C /\!\!/ B \\ \Leftrightarrow & \{(14)\} \\ & (A \leq_R C) \wedge (B \leq_R B) \wedge (R \wedge gd\,A \Rightarrow C) \\ \Leftrightarrow & \{\text{conditions (b) and (c) of } \mathcal{A} \leq_R \mathcal{C}\} \\ & (B \leq_R B) \\ \Leftarrow & \{(15)\} \\ & B \text{ preserves } R \\ \Leftrightarrow & \{a, w, x \text{ are not assigned in } B, (3)\} \\ & \mathit{true} \end{array}$$

By (7), the *exit condition* and its proof are identical to the previously given one.

The right hand side of (18) expands to:

$$\begin{array}{l} |[\ \mathbf{var}\ a, w, x*, b, y* \mid B_0 \wedge A_0 \bullet \mathbf{do}\ B /\!\!/ A\ \mathbf{od}\ ]| \quad \leq_R \\ \quad |[\ \mathbf{var}\ c, w, x*, b, y* \mid B_0 \wedge C_0 \bullet \mathbf{do}\ B /\!\!/ C\ \mathbf{od}\ ]| \end{array}$$

The *initialisation* condition and its proof are identical to the previously given one.

For *main action* we have:

$$
\begin{array}{ll}
 & B \mathbin{/\!/} A \leq_R B \mathbin{/\!/} C \\
\Leftrightarrow & \{(14)\} \\
 & (B \leq_R B) \wedge (A \leq_R C) \wedge (R \wedge gd\,B \Rightarrow gd\,B) \\
\Leftarrow & \{\text{condition (b) of } \mathcal{A} \leq_R \mathcal{C}\text{, logic}\} \\
 & (B \leq_R B) \\
\Leftarrow & \{(15)\} \\
 & B \text{ preserves } R \\
\Leftrightarrow & \{a, w, x \text{ are not assigned in } B\text{, (3)}\} \\
 & \mathit{true}
\end{array}
$$

By (7), the *exit condition* and its proof are identical to the previously given one. □

If an action system is refined in several steps, then the refinement relation between the variables of the original and final action system can be calculated from the individual refinement relations. Let m be those variables of $\mathcal{B}$ which are refined under R in $\mathcal{A} \leq_R \mathcal{B}$ and are further refined under S in $\mathcal{B} \leq_S \mathcal{C}$. Let $R\,;\,S$ stand for $(\exists\, m \bullet R \wedge S)$.

$$(\mathcal{A} \leq_R \mathcal{B}) \wedge (\mathcal{B} \leq_S \mathcal{C}) \quad \Rightarrow \quad (\mathcal{A} \leq_{R\,;\,S} \mathcal{C}) \tag{19}$$

The rule is applied when both operands of a prioritising composition are being refined independently. Let $\mathcal{D}$ also be an action system.

$$(\mathcal{A} \leq_R \mathcal{C}) \wedge (\mathcal{B} \leq_S \mathcal{D}) \quad \Rightarrow \quad (\mathcal{A} \mathbin{/\!/} \mathcal{B} \leq_{R \wedge S} \mathcal{C} \mathbin{/\!/} \mathcal{D}) \tag{20}$$

Proof. We calculate:

$$
\begin{array}{ll}
 & (\mathcal{A} \leq_R \mathcal{C}) \wedge (\mathcal{B} \leq_S \mathcal{D}) \\
\Rightarrow & \{(18), (17)\} \\
 & (\mathcal{A} \mathbin{/\!/} \mathcal{B} \leq_R \mathcal{C} \mathbin{/\!/} \mathcal{B}) \wedge (\mathcal{C} \mathbin{/\!/} \mathcal{B} \leq_S \mathcal{C} \mathbin{/\!/} \mathcal{D}) \\
\Rightarrow & \{(19)\} \\
 & (\mathcal{A} \mathbin{/\!/} \mathcal{B} \leq_{R\,;\,S} \mathcal{C} \mathbin{/\!/} \mathcal{D}) \\
\Leftrightarrow & \{\text{disjoint variables are refined under } R \text{ and } S\} \\
 & (\mathcal{A} \mathbin{/\!/} \mathcal{B} \leq_{R \wedge S} \mathcal{C} \mathbin{/\!/} \mathcal{D})
\end{array}
$$

□

An analogous rule holds for parallel composition. Jointly, these rules allow processes in a prioritised process system to be refined individually and the refinement relation between the initial and final process system to be calculated.

5 Decomposition

One method of developing a parallel program is to first specify it without consideration of parallelism, and then add parallelism in subsequent development steps. If the specification is given as an action system

$$\mathcal{A} \;=\; |[\ \mathbf{var}\ a, x* \mid A_0 \bullet \mathbf{do}\ A\ \mathbf{od}\ ]|\ ,$$

the definition of parallel composition directly gives a way of doing so [4]:

Theorem (*parallel decomposition*). If action system $\mathcal{A}$ is of the form

$$\mathcal{A} \;=\; |[\ \mathbf{var}\ b, c, y*, z* \mid B_0 \wedge C_0 \bullet \mathbf{do}\ B \,[\!]\, C\ \mathbf{od}\ ]|$$

where variables c, z do not occur in B_0, variables b, y do not occur in C_0, and furthermore variables c do not occur in B, variables b do not occur in C, then

$$\mathcal{A} \;=\; \mathcal{B} \parallel \mathcal{C}$$

where:

$$\begin{array}{lcl} \mathcal{B} & = & |[\ \mathbf{var}\ b, y* \mid B_0 \bullet \mathbf{do}\ B\ \mathbf{od}\ ]| \\ \mathcal{C} & = & |[\ \mathbf{var}\ c, z* \mid C_0 \bullet \mathbf{do}\ C\ \mathbf{od}\ ]| \end{array}$$ □

This development method for action systems is now extended to prioritised programs. We introduce a shorthand for the case when action system $\mathcal{A}$ is refined by action system $\mathcal{A}'$ with same local and public variables as $\mathcal{A}$ via some refinement relation:

$$\mathcal{A} \leq \mathcal{A}' \qquad \text{iff for some } R: \quad \mathcal{A} \leq_R \mathcal{A}'$$

This special case of refinement may only reduce nondeterminism of the initialisation and the action and refine aborting behaviour of the action to nonaborting behaviour. Mutual refinement $\mathcal{A} \leq \mathcal{A}'$ and $\mathcal{A}' \leq \mathcal{A}$ does not imply equality of $\mathcal{A}$ and $\mathcal{A}'$, since the action of $\mathcal{A}$ and $\mathcal{A}'$ may behave arbitrarily in unreachable states.

Theorem (*prioritising decomposition*). If action system $\mathcal{A}$ is of the form

$$\mathcal{A} \;=\; |[\ \mathbf{var}\ b, c, y*, z* \mid B_0 \wedge C_0 \bullet \mathbf{do}\ B \,[\!]\, g \rightarrow C\ \mathbf{od}\ ]|$$

where variables c, z do not occur in B_0, variables b, y do not occur in C_0, and furthermore variables c do not occur in B, variables b do not occur in C, and for some predicate I,

(a) *initialisation*: $B_0 \wedge C_0 \Rightarrow I$,

(b) *preservation*: $(B \text{ preserves } I) \wedge (C \text{ preserves } I)$,

(c) *exit condition*: $I \wedge \neg gd\,B \wedge gd\,C \Rightarrow g$

then

$$\mathcal{A} \;\leq\; \mathcal{B} \mathbin{/\!/} \mathcal{C}$$

where:

$$\begin{array}{lcl} \mathcal{B} & = & |[\ \mathbf{var}\ b, y* \mid B_0 \bullet \mathbf{do}\ B\ \mathbf{od}\ ]| \\ \mathcal{C} & = & |[\ \mathbf{var}\ c, z* \mid C_0 \bullet \mathbf{do}\ C\ \mathbf{od}\ ]| \end{array}$$ □

The role of the *exit condition* is to ensure that when eliminating g, action C does not become enabled in $\mathcal{B} \mathbin{/\!/} \mathcal{C}$ when it was not in $\mathcal{A}$, because then $\mathcal{B} \mathbin{/\!/} \mathcal{C}$ would not terminate when $\mathcal{A}$ would. However, eliminating g may decrease nondeterminism.

Proof. Taking I for the refinement relation, we have to show:

$$\begin{array}{ll} |[\ \mathbf{var}\ b, c, y*, z* \mid B_0 \wedge C_0 \bullet \mathbf{do}\ B \mathbin{[\!]} g \rightarrow C\ \mathbf{od}\ ]| & \leq_I \\ \quad |[\ \mathbf{var}\ b, c, y*, z* \mid B_0 \wedge C_0 \bullet \mathbf{do}\ B \mathbin{/\!/} C\ \mathbf{od}\ ]| & \end{array}$$

Assuming (a), (b), and (c) of the theorem, three conditions have to be checked. For *initialisation* we have:

$$\begin{array}{ll} & B_0 \wedge C_0 \Rightarrow I \wedge B_0 \wedge C_0 \\ \Leftrightarrow & \{\text{logic, (a)}\} \\ & \mathit{true} \end{array}$$

For *main action* we have:

$$\begin{array}{ll} & B \mathbin{[\!]} g \rightarrow C \leq_I B \mathbin{/\!/} C \\ \Leftrightarrow & \{\text{definition of } \mathbin{/\!/},\ C = gd\,C \rightarrow C\} \\ & B \mathbin{[\!]} g \rightarrow C \leq_I B \mathbin{[\!]} \neg gd\,B \rightarrow (gd\,C \rightarrow C) \\ \Leftrightarrow & \{\text{definition of } \rightarrow, \text{ for any } p, q : [p]\,;\,[q] = [p \wedge q]\} \\ & B \mathbin{[\!]} [g]\,;\,C \leq_I B \mathbin{[\!]} [\neg gd\,B \wedge gd\,C]\,;\,C \\ \Leftarrow & \{\text{piecewise refinement of } \mathbin{[\!]} \text{ and } ;\} \\ & (B \leq_I B) \wedge ([g] \leq_I [\neg gd\,B \wedge gd\,C]) \wedge (C \leq_I C) \\ \Leftrightarrow & \{\text{assumption (b), (15), for any } p, q, R : [p] \leq_R [q] \text{ iff } R \wedge q \Rightarrow p\} \\ & I \wedge \neg gd\,B \wedge gd\,C \Rightarrow g \\ \Leftrightarrow & \{\text{assumption (c)}\} \\ & \mathit{true} \end{array}$$

For *exit condition* we have:

$$
\begin{array}{ll}
 & I \wedge gd(B \,[\!]\, g \to C) \Rightarrow gd(B \mathbin{/\!\!/} C) \\
\Leftrightarrow & \{(4),(7)\} \\
 & I \wedge (gd\,B \vee (g \wedge gd\,C)) \Rightarrow gd\,B \vee gd\,C \\
\Leftrightarrow & \{\text{logic}\} \\
 & \mathit{true}
\end{array}
$$

□

The idea of prioritising decomposition is similar to that of transforming a guarded conditional to a form with else-branches. It gives a more concise description by eliminating possibly huge guards (as the applications in Sec. 7 suggest), it shows the intentions explicitly, and it allows an efficient implementation: On a single processor machine, lists with processes of each priority need to be kept, and only if no processes of a certain priority are enabled, does the list with the next lower priority processes need to be consulted.

6 Changing Priorities

In this section we investigate under which conditions the priority of a process can be lowered and raised. The typical use of this is to improve the responsiveness of one process in a way that the correctness of the process system is preserved. We start by priority changing laws for actions.

If we have the choice between two actions, we can always give preference to one:

$$A \,[\!]\, B \;\leq\; A \mathbin{/\!\!/} B \tag{21}$$

The reverse is true only under an additional condition of exclusion. An action A excludes an action B if their guards are disjoint.

$$A \text{ excludes } B \qquad \text{iff} \qquad gd\,A \Rightarrow \neg gd\,B$$

Exclusion of actions is symmetric: if A excludes B, then B also excludes A. Under the assumption of exclusion, choice and prioritising composition are equal:

$$A \,[\!]\, B \;=\; A \mathbin{/\!\!/} B \qquad \text{if } A \text{ excludes } B \tag{22}$$

We give some laws when three actions with only two priorities are involved. When two actions are of high priority, one of them can be assigned low priority in the case when it excludes the other action of low priority.

$$(A \,[\!]\, B) \mathbin{/\!\!/} C \;\leq\; A \mathbin{/\!\!/} (B \,[\!]\, C) \qquad \text{if } B \text{ excludes } C \tag{23}$$

Proof. This follows from applying (21) and then (22). □

Likewise, if two actions are of low priority, one of them can be assigned high priority in the case when it excludes the other action of high priority.

$$A \mathbin{/\!/} (B \mathbin{[\!]} C) \quad \leq \quad (A \mathbin{[\!]} B) \mathbin{/\!/} C \qquad \text{if } A \text{ excludes } B \tag{24}$$

Proof. This follows from applying (21) and then (22). □

Finally, combining the last two laws gives the conditions under which an action can be moved between high and low priority.

$$(A \mathbin{[\!]} B) \mathbin{/\!/} C \quad = \quad A \mathbin{/\!/} (B \mathbin{[\!]} C) \qquad \text{if } A \text{ excludes } B \text{ and } B \text{ excludes } C \tag{25}$$

We continue with priority changing laws for action systems. If we have two processes running in parallel, we can always give preference to one of them:

$$\mathcal{A} \parallel \mathcal{B} \quad \leq \quad \mathcal{A} \mathbin{/\!/} \mathcal{B} \tag{26}$$

Proof. Taking the refinement relation R to be *true*, we have to show that:

$$\begin{array}{l} |[\ \mathbf{var}\ a, b \mid A_0 \wedge B_0 \bullet \mathbf{do}\ A \mathbin{[\!]} B\ \mathbf{od}\ ]| \quad \leq_{true} \\ \quad |[\ \mathbf{var}\ a, b \mid A_0 \wedge B_0 \bullet \mathbf{do}\ A \mathbin{/\!/} B\ \mathbf{od}\ ]| \end{array}$$

The three conditions are easily checked. □

The reverse is only true under an additional condition of exclusion. Action system $\mathcal{A}$ excludes action system $\mathcal{B}$, if for some predicate I,

(a) *initialisation*: $A_0 \wedge B_0 \Rightarrow I$,

(b) *preservation*: (A preserves I) $\wedge$ (B preserves I),

(c) *exclusion*: $I \Rightarrow$ (A excludes B) .

Exclusion of action systems is symmetric: if $\mathcal{A}$ excludes $\mathcal{B}$, then $\mathcal{B}$ also excludes $\mathcal{A}$. Under the assumption of exclusion, prioritising composition can be transformed to parallel composition.

$$\mathcal{A} \mathbin{/\!/} \mathcal{B} \quad \leq \quad \mathcal{A} \parallel \mathcal{B} \qquad \text{if } \mathcal{A} \text{ excludes } \mathcal{B} \tag{27}$$

Proof. As $\mathcal{A}$ excludes $\mathcal{B}$, we have that the corresponding conditions (a) to (c) hold for some I. Taking the I for the refinement relation, we have to show:

$$\begin{array}{l} |[\ \mathbf{var}\ a, b \mid A_0 \wedge B_0 \bullet \mathbf{do}\ A \mathbin{/\!/} B\ \mathbf{od}\ ]| \quad \leq_I \\ \quad |[\ \mathbf{var}\ a, b \mid A_0 \wedge B_0 \bullet \mathbf{do}\ A \mathbin{[\!]} B\ \mathbf{od}\ ]| \end{array}$$

Three conditions have to be checked. The *initialisation* and the *exit condition* follow easily. For *main action* we have:

$$
\begin{array}{ll}
 & A \mathbin{/\!\!/} B \leq_I A \mathbin{[\!]} B \\
\Leftrightarrow & \{\text{as } B = gd\,B \to B\text{, definition of } \mathbin{/\!\!/} \text{ and } \to\} \\
 & A \mathbin{[\!]} [\neg gd\,A]\,;\, B \leq_I A \mathbin{[\!]} [gd\,B]\,;\, B \\
\Leftarrow & \{\text{piecewise refinement of } \mathbin{[\!]} \text{ and } ;\} \\
 & (A \leq_I A) \wedge ([\neg gd\,A] \leq_I [gd\,B]) \wedge (B \leq_I B) \\
\Leftrightarrow & \{(15)\text{, for } p,q,R : [p] \leq_R [q] \text{ if } q \wedge R \Rightarrow q\} \\
 & (A \text{ preserves } I) \wedge (gd\,B \wedge I \Rightarrow \neg gd\,A) \wedge (B \text{ preserves } I) \\
\Leftrightarrow & \{\text{conditions (b) and (c) of } \mathcal{A} \text{ excludes } \mathcal{B}\ \} \\
 & \mathit{true}
\end{array}
$$

□

We give some laws when three action systems with only two priorities are involved. When two action systems are of high priority, one of them can be assigned low priority in case it excludes the other action system of low priority.

$$(\mathcal{A} \parallel \mathcal{B}) \mathbin{/\!\!/} \mathcal{C} \quad \leq \quad \mathcal{A} \mathbin{/\!\!/} (\mathcal{B} \parallel \mathcal{C}) \qquad \text{if } \mathcal{B} \text{ excludes } \mathcal{C} \tag{28}$$

Proof. This follows from applying (26) and then (27). □

Likewise, if two action systems are of low priority, one of them can be assigned high priority in case it excludes the other action system of high priority.

$$\mathcal{A} \mathbin{/\!\!/} (\mathcal{B} \parallel \mathcal{C}) \quad \leq \quad (\mathcal{A} \parallel \mathcal{B}) \mathbin{/\!\!/} \mathcal{C} \qquad \text{if } \mathcal{A} \text{ excludes } \mathcal{B} \tag{29}$$

Proof. This follows from applying (26) and then (27). □

7 Applications

In this section we give a number of application domains for the concepts introduced. This section is not merely a collection of examples of the developed theory, but more an account of different areas where the operators can be used and sketches on their use.

7.1 A Shared Resource

Let us apply the concepts of prioritising decomposition, and lowering and raising priorities. We use a modification of an example by Lowe [16]. Consider the action

system

$$\begin{array}{lcl} \mathit{Print} & = & |[\ \textbf{var}\ \mathit{lp}_i{*} : \mathit{seq}\ T\ \ \textbf{for}\ i = 1..3 \mid \mathit{lp}_i = \langle\,\rangle\ \ \textbf{for}\ i = 1..3 \bullet \\ & & \quad \textbf{do}\ g_i \rightarrow \mathit{lp}_i := \mathit{lp}_i \circ \langle f_i \rangle\ \ \textbf{for}\ i = 1..3 \\ & & \quad [\!]\ \mathit{lp}_i \neq \langle\,\rangle \rightarrow \mathit{print\ hd\ lp}_i\ \ \textbf{for}\ i = 1..3 \\ & & \quad \textbf{od}\ ]| \end{array}$$

where the first action $g_i \rightarrow \mathit{lp}_i := \mathit{lp}_i \circ \langle f_i \rangle$ is supposed to place some file f_i into the printer queue lp_i whenever the condition g_i (not specified here) is true. The second action $\mathit{lp}_i \neq \langle\,\rangle \rightarrow \mathit{print\ hd\ lp}_i$ sends the first file of queue lp_i to the printer. (We ignore removing the file from the queue.) We can decompose *Print* into the following four components:

$$\begin{array}{lcl} \mathit{User}_i & = & |[\ \textbf{var}\ \mathit{lp}_i{*} : \mathit{seq}\ T \mid \mathit{lp}_i = \langle\,\rangle \bullet \\ & & \quad \textbf{do}\ g_i \rightarrow \mathit{lp}_i := \mathit{lp}_i \circ \langle f_i \rangle\ \textbf{od}\ ]| \\ \mathit{Lpr} & = & |[\ \textbf{do}\ \mathit{lp}_i \neq \langle\,\rangle \rightarrow \mathit{print\ hd\ lp}_i\ \ \textbf{for}\ i = 1..3\ \textbf{od}\ ]| \end{array}$$

Now we have that:

$$\mathit{Print} \ = \ \mathit{Lpr} \parallel \mathit{User}_1 \parallel \mathit{User}_2 \parallel \mathit{User}_3$$

If we want to give priority for the printer server over the users, we combine the system as:

$$\mathit{Print} \ \leq \ \mathit{Lpr} \mathbin{/\!/} (\mathit{User}_1 \parallel \mathit{User}_2 \parallel \mathit{User}_3)$$

We can also compose the system by giving a higher priority to one of the users.

$$\mathit{Lpr} \mathbin{/\!/} (\mathit{User}_1 \parallel \mathit{User}_2 \parallel \mathit{User}_3) \ \leq \ \mathit{Lpr} \mathbin{/\!/} \mathit{User}_1 \mathbin{/\!/} (\mathit{User}_2 \parallel \mathit{User}_3)$$

Moreover, some user can be given a priority higher than the printer server.

$$\mathit{Print} \ \leq \ \mathit{User}_1 \mathbin{/\!/} \mathit{Lpr} \mathbin{/\!/} (\mathit{User}_2 \parallel \mathit{User}_3)$$

Or all users can run on higher priority than the server.

$$\mathit{Print} \ \leq \ (\mathit{User}_1 \parallel \mathit{User}_2 \parallel \mathit{User}_3) \mathbin{/\!/} \mathit{Lpr}$$

We can also decompose the system so that every user has his or her own printer server.

$$\mathit{Lpr}_i \ = \ |[\ \textbf{do}\ \mathit{lp}_i \neq \langle\,\rangle \rightarrow \mathit{print\ hd\ lp}_i\ \textbf{od}\ ]|$$

Then we have that:

$$\mathit{Print} \ \leq \ (\mathit{Lpr}_1 \mathbin{/\!/} \mathit{User}_1) \parallel (\mathit{Lpr}_2 \mathbin{/\!/} \mathit{User}_2) \parallel (\mathit{Lpr}_3 \mathbin{/\!/} \mathit{User}_3)$$

The advantage of this over the decomposition $Lpr \mathbin{/\!/} (User_1 \parallel User_2 \parallel User_3)$ is that each user now has to wait only for his or her own printer server, previously all files had to be printed before another could be enqueued. Hence, starvation can be a problem when using prioritising composition in this way. In the action system

$$\begin{array}{lll} Print' & = & |[\ \mathbf{var}\ lp_i* : seq\ T\ \ \mathbf{for}\ i = 1..3 \mid lp_i = \langle\,\rangle\ \ \mathbf{for}\ i = 1..3 \bullet \\ & & \mathbf{do}\ g_i \rightarrow lp_i := lp_i \circ \langle f_i \rangle \\ & & [\!]\ \neg g_i \wedge lp_i \neq \langle\,\rangle \rightarrow print\ hd\ lp_i\ \ \mathbf{for}\ i = 1..3 \\ & & \mathbf{od}\ ]| \end{array}$$

we avoid part of starvation, because the printer takes over only when a user has nothing to do. Now applying our prioritising decomposition theorem of Sec. 5, the system $Print'$ can be refined as follows:

$$Print' \quad \leq \quad (User_1 \mathbin{/\!/} Lpr_1) \parallel (User_2 \mathbin{/\!/} Lpr_2) \parallel (User_3 \mathbin{/\!/} Lpr_3)$$

where $User_i$ and Lpr_i for $i = 1..3$ are as above. The refinement is correct, because $\neg g_i \wedge (\neg g_i \wedge lp_i \neq \langle\,\rangle) \ \Rightarrow\ lp_i \neq \langle\,\rangle$, which is condition (c) of the theorem with invariant I as true. The other conditions hold trivially.

7.2 Overriding Behaviour

Consider a process that copies input values to a buffer and outputs them as needed. Input and output values are stored in global variables i, o of type T, respectively. Availability of an input value and a request for an output is indicated by boolean variables ir, or, respectively. Furthermore the variable *reset* is set by the environment if the buffer needs to be emptied immediately.

$$\mathbf{var}\ i, o : T$$
$$\mathbf{var}\ ir, or, reset : Bool$$

In case there is both a request for input and output, either request can be handled first. However, if there is a request for resetting the buffer, that request has priority. This is expressed below by an action system with a body of the form $A_1 \mathbin{/\!/} (A_2 [\!] A_3)$. With the understanding that the operators $[\!]$ and $/\!/$ have same binding power and associate to the right, the parenthesis are omitted:

$$\begin{array}{lll} Buffer & = & |[\ \mathbf{var}\ buf : seq\ T \mid buf = \langle\,\rangle \bullet \\ & & \mathbf{do}\ reset \rightarrow buf, reset := \langle\,\rangle, false \\ & & \mathbin{/\!/}\ ir \rightarrow buf, ir := buf \circ \langle i \rangle, false \\ & & [\!]\ or \wedge buf \neq \langle\,\rangle \rightarrow o, or, buf := hd\ buf, false, tl\ buf \\ & & \mathbf{od}\ ]| \end{array}$$

This could be equivalently expressed by replacing the operator $/\!/$ by the operator $[\!]$ and conjoining $\neg reset$ to the guards of the last two actions.

Inheritance The above can be expressed more clearly by a form of inheritance on action systems. Suppose $\mathcal{A}$ and $\mathcal{B}$ are given by:

$$\begin{array}{lcl} \mathcal{A} & = & |[\ \textbf{var}\ a, x* \mid A_0 \bullet \textbf{do}\ A\ \textbf{od}\ ]| \\ \mathcal{B} & = & |[\ \textbf{inherit}\ \mathcal{A} \\ & & \textbf{var}\ b, y* \mid B_0 \bullet \textbf{do}\ B\ \textbf{od}\ ]| \end{array}$$

Inheritance is defined by the prioritising composition of the actions and joining the variable declarations, i.e.:

$$\mathcal{B} \ = \ |[\ \textbf{var}\ a, b, x*, y* \mid A_0 \wedge B_0 \bullet \textbf{do}\ B\ /\!/\ A\ \textbf{od}\ ]|$$

The previous example is now equivalently expressed by:

$$\begin{array}{lcl} \mathit{SimpleBuf} & = & |[\ \textbf{var}\ \mathit{buf} : \mathit{seq}\ T \mid \mathit{buf} = \langle\,\rangle \bullet \\ & & \textbf{do}\ \mathit{ir} \rightarrow \mathit{buf}, \mathit{ir} := \mathit{buf} \circ \langle i \rangle, \mathit{false} \\ & & \ [\!]\ \mathit{or} \wedge \mathit{buf} \neq \langle\,\rangle \rightarrow o, \mathit{or}, \mathit{buf} := \mathit{hd}\,\mathit{buf}, \mathit{false}, \mathit{tl}\,\mathit{buf} \\ & & \textbf{od}\ ]| \\ \mathit{Buffer} & = & |[\ \textbf{inherit}\ \mathit{SimpleBuf} \\ & & \textbf{do}\ \mathit{reset} \rightarrow \mathit{buf}, \mathit{reset} := \langle\,\rangle, \mathit{false}\ \textbf{od}\ ]| \end{array}$$

More generally, the effect of inheritance is as follows. Consider the action systems

$$\begin{array}{lcl} \mathcal{A} & = & |[\ \textbf{var}\ a, x* \mid A_0 \bullet \textbf{do}\ A\ [\!]\ A'\ \textbf{od}\ ]| \\ \mathcal{B} & = & |[\ \textbf{inherit}\ \mathcal{A} \\ & & \textbf{var}\ b, y* \mid B_0 \bullet \textbf{do}\ B\ \textbf{od}\ ]| \end{array}$$

where A and A' are such that and $gd\,A \Rightarrow gd\,B$ and A' excludes B. (Any action system $\mathcal{A}$ can be written in this form.). Then B overrides A and has the same priority as A':

$$\begin{array}{lcl} \mathcal{B} & = & |[\ \textbf{inherit}\ \mathcal{A} \\ & & \textbf{var}\ a, b, x*, y* \mid A_0 \wedge B_0 \bullet \textbf{do}\ A'\ [\!]\ B\ \textbf{od}\ ]| \end{array}$$

Superposition Inheritance models a form of superposition in the following way. Let B be an action with $gd\,B \Rightarrow gd\,A$. Then in

$$\mathcal{B} \;=\; |[\ \mathbf{var}\ a, x* \mid A_0 \bullet \mathbf{do}\ (B \mathbin{/\!/} A) \mathbin{[\!]} A'\ \mathbf{od}\ ]|$$

the action B is chosen whenever it and A are enabled at the same time. The action A is chosen whenever $gd\,A \wedge \neg gd\,B$ holds. Let $\mathcal{A}$ be as above. If A excludes A', then $\mathcal{B}$ can be equivalently expressed as:

$$\begin{array}{lcl} \mathcal{B} & = & |[\ \mathbf{inherit}\ \mathcal{A} \\ & & \mathbf{do}\ B\ \mathbf{od}\ ]| \end{array}$$

The underlying understanding is that the actions are identified by their guards (rather than their name). To model a stronger form of superposition, where A is never chosen in $\mathcal{B}$, the condition $gd\,A \Rightarrow gd\,B$ is needed. Superposition of action systems is more thoroughly discussed by Back and Sere [5].

7.3 Control Systems

A control system consist of a control program, the *controller* which maintains a continuous interaction with its environment, the *plant*. *Sensors* are used by the controller for inspecting the state of the plant and *actuators* for influencing it. We can describe both controller and plant by action systems. Let the values of the sensors and actuators be given by the variables s, a, respectively. A part of the state of the plant might not be visible to the controller, which is modelled by variables local to the plant. Likewise, the controller may also have local variables:

$$\begin{array}{lcl} \mathit{Plant} & = & |[\ \mathbf{var}\ p, s* \mid P_0 \bullet \mathbf{do}\ P\ \mathbf{od}\ ]| \\ \mathit{Controller} & = & |[\ \mathbf{var}\ c, a* \mid C_0 \bullet \mathbf{do}\ C\ \mathbf{od}\ ]| \end{array}$$

We consider the sensors to be exported variables of the plant and the actuators to be exported variables of the controller. The behaviour of the whole control system is then given by:

$$\mathit{System} \;=\; \mathit{Controller} \mathbin{/\!/} \mathit{Plant}$$

Prioritising composition is used to express that the evolution of the plant is interrupted by controller actions as soon as they become enabled: the controller actions harness the plant so that the plant is forced to behave in a correct, well-controlled way. Once the controller is enabled, the controller action is taken, possibly repeatedly, until the controller disables itself. This assumes that the controller is sufficiently fast such that a possible evolution of the plant during the controller reaction is negligible.

Because of the monotonicity of prioritising composition, both controller and plant can be refined independently. An application of this model to a discrete event control system is described by Rönkkö, Sekerinski, Sere [21]. The systematic derivation of control programs using this model is studied by Sekerinski [22], and this was also the initial motivation for introducing prioritising composition on action systems.

7.4 Timed Action Systems

Consider an action system $\mathcal{A}$ which accesses the global variable *now* that represents time. This variable is used in a read only manner in the system $\mathcal{A}$, typically in guards. Let further $\mathcal{T}$ be an action system that models the passage of time:

$$\mathcal{T} = |[\ \textbf{var}\ now{*} :\ int \bullet \textbf{do}\ true \rightarrow now := now + 1\ \textbf{od}\ ]|$$

Then the prioritising composition $\mathcal{A} \mathbin{//} \mathcal{T}$ models a real time action system where either time advances or an action of $\mathcal{A}$ is taken. However, the advancement of time may not disable an action that was enabled in $\mathcal{A}$, ensuring that no action of $\mathcal{A}$ misses its deadline. This way of modelling real time action systems was originally proposed by Fidge and Wellings [13]. It conforms with the maximal progress assumption for real-time systems which states that time is only allowed to advance if there is nothing else to do.

Observe that the prioritising operator hides a possibly large guard: without it, the guard of the tick action $true \rightarrow now := now + 1$ would be $\neg gd\,A$ which, in case $A = A_1 \mathbin{[\!]} \cdots \mathbin{[\!]} A_n$, can be a very large expression.

7.5 Dynamic Priorities

Dynamic priorities can be handled by storing the priority of a process or an action in a special variable and including the priority check as part of the enabledness condition. Below we give two examples of the use of this idea. First, we model a nonpreemptive scheduler implementing a round-robin order on actions. Thereafter we consider a preemptive scheduler that wakes up at a regular basis and reschedules the actions.

Nonpreemptive (cooperative) scheduler Consider the following action system which models a round-robin scheduler by setting the variable π circularly.

$$\mathcal{S} = |[\ \textbf{var}\ \pi{*} :\ int \mid \pi = 0 \bullet \textbf{do}\ true \rightarrow \pi := (\pi + 1)\ mod\ n\ \textbf{od}\ ]|$$

Let $\mathcal{A}$ be a system in which the actions A_i are initially guarded by different values π_i.

$$\begin{array}{lcl} \mathcal{A} & = & |[\ \textbf{var}\ \pi_i :\ \mathit{int}\ \ \textbf{for}\ i = 0..n-1 \mid \pi_i = i\ \ \textbf{for}\ i = 0..n-1 \bullet \\ & & \quad \textbf{do}\ \pi = \pi_i \rightarrow A_i\ \ \textbf{for}\ i = 0..n-1\ \textbf{od}\ ]| \end{array}$$

Now we have that $\mathcal{A} \mathbin{/\!\!/} \mathcal{S}$ models a system, where each of the actions $A_1, \ldots A_n$ is given a chance to be active based on the round-robin scheduler. In case the action A_i is not enabled when $\pi = \pi_i$, the action misses its turn and has to wait for the next round. By storing each action's turn in the variables π_i, actions may change their (or even anothers) turn.

Preemptive scheduler Let $\mathcal{A}$ be as follows

$$\begin{array}{lcl} \mathcal{A} & = & |[\ \textbf{var}\ \tau_i* :\ \mathit{int}\ \ \textbf{for}\ i = 1..n \mid \tau_i = 0\ \ \textbf{for}\ i = 1..n \bullet \\ & & \quad \textbf{do}\ \pi_i = \mathit{max}(\pi_1, \ldots, \pi_n) \rightarrow A_i\ \ \textbf{for}\ i = 1..n\ \textbf{od}\ ]| \end{array}$$

Here we assume that the variables τ_i model the execution time used by each action (or process). Hence, the variable τ_i is modified in action A_i. Let $\mathcal{S}$ be the action system:

$$\begin{array}{lcl} \mathcal{S} & = & |[\ \textbf{var}\ \pi_i* :\ \mathit{int}\ \ \textbf{for}\ i = 1..n \mid \pi_i = 0\ \ \textbf{for}\ i = 1..n \bullet \\ & & \quad \textbf{do}\ g(\tau_1, \ldots, \tau_n) \rightarrow \pi_i := f_i(\tau_1, \ldots, \tau_n)\ \ \textbf{for}\ i = 1..n\ \textbf{od}\ ]| \end{array}$$

Then the system $\mathcal{S} \mathbin{/\!\!/} \mathcal{A}$ models a preemptive scheduler where always when the condition $g(\tau_1, \ldots, \tau_n)$ holds in the system $\mathcal{A}$, the scheduler $\mathcal{S}$ will assign new priority classes π_i to the processes based on functions $f_i(\tau_1, \ldots, \tau_n)$ of the execution times.

Here the scheduler changes the priority variables, but the scheduler could be modified such that the processes can set their own priorities, as with the round-robin scheduler above.

8 Conclusions

We developed a theory of prioritising composition of processes and gave a wide range of applications of the theory. The theory is based on describing processes as action systems, but the idea can be applied more generally as only basic notions of actions and action refinement were used. Extensions of the theory in several directions are possible.

Other operators on action systems can be defined, including sequential composition (;), choice ($[\!]$), conditional (**if**), iteration (**do**), local variables (**var**), as done

by Back and Sere [4]. In that case, action systems model statements of a parallel programming language. The combination of prioritising composition with these operators has not been studied. In particular, having variable declarations would be useful for parallel and prioritising decomposition, as it allows the variables over which the action systems communicate to be hidden [2, 4].

The refinement relation used requires that there is a one-to-one correspondence between the abstract and concrete actions. More general notions of refinement are possible which allow stuttering, i.e. one abstract step may correspond to several concrete steps or several abstract steps may correspond to one concrete step. Refinement of action systems with stuttering is presented and justified with respect to a trace semantics by Back and von Wright [6].

The public variables of an action systems are restricted to read-only export and the refinement relation was only allowed to replace the private variables. This was done to ensure compositionality of both parallel and prioritising composition. More liberal notions of refinement are possible, but lead to either noncompositionality or additional proof obligations. Xu studies such refinements, based on the idea of rely-guarantee conditions [25].

The underlying semantics of action systems is not fully abstract in the sense that two action systems may behave identically in any context, but not be equal. For example, this is the case when $\mathcal{A} \leq \mathcal{A}'$ and $\mathcal{A}' \leq \mathcal{A}$. A fully abstract semantics for a shared variable parallel programming language has been given by Brookes [8]. It would be interesting to extend such a semantics to include prioritising composition.

Finally, our interpretation of dynamically changing priorities is to introduce special priority variables as we did with time also. (Observe that the examples on timed action systems and the nonpreemptive scheduler are very similar.) The access to these variables is restricted, and both static and dynamic priorities can be mixed in a specification as shown by our final example. Furthermore, our transformation and refinement rules are directly applicable to dynamic priorities as well. The idea of using special priority variables is studied by Henzinger et al. when modelling processes as timed transition systems [14] much in a similar way as we do here. They do not, however, consider refinement. Moreover, in their formalism one does not mix dynamic and static priorities as we do.

Acknowledgements We are grateful for discussions on the topic to R. Back, M. Butler, J. Grundy, and J. von Wright.

References

[1] R.J.R. Back. Data refinement in the refinement calculus. In *22nd Hawaii International Conference of System Sciences*, Kailua-Kona, 1989.

[2] R.J.R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness*, Lecture Notes in Computer Science 430, pages 67–93, Mook, The Netherlands, 1989. Springer Verlag.

[3] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[4] R.J.R. Back and K. Sere. From action systems to modular systems. In M. Naftalin, T. Denvir, and M. Bertran, editors, *Formal Methods Europe '94*, pages 1–25, Barcelona, Spain, 1994. Lecture Notes in Compter Science 873, Springer-Verlag.

[5] R.J.R. Back and K. Sere. Superposition refinement of action systems. *Formal Aspects of Computing*, 8(3):324–346, 1996.

[6] R.J.R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR '94: Concurrency Theory*, Lecture Notes in Computer Science 836. Springer-Verlag, 1994.

[7] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, 9:127–168, 1986.

[8] S. Brookes. Full abstraction for a shared variable parallel language. In *8th IEEE International Symposium on Logic in Computer Science*, 1993.

[9] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[10] R. Cleaveland and M.C.P. Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2), 1990.

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

[12] C.J. Fidge. A formal definition of priority in CSP. *ACM Transactions on Programming Languages and Systems*, 15(4):681–705, 1993.

[13] C.J. Fidge and A.J. Wellings. An action-based formal model for concurrent, real-time systems. Technical Report 95-1, Software Verification Research Centre, January 1995.

[14] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112(2):273–337, 1994.

[15] L. Lamport. A temporal logic of actions. Research Report 57, DEC System Research Center, April 1990.

[16] G. Lowe. Probabilistic and prioritized models of timed CSP. *Theoretical Computer Science*, 128(2):315–352, 1995.

[17] INMOS Ltd. *occam2 Reference Manual*. Prentice-Hall, 1988.

[18] C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[19] J.M. Morris. Laws of data refinement. *Acta Informatica*, 9(3):287–306, 1989.

[20] V. Natarajan, L. Christoff, and R. Cleaveland. Priorities and abstraction in process algebra. In P.S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 880, Madras, India, 1994. Springer-Verlag.

[21] M. Rönkkö, E. Sekerinski, and K. Sere. Control systems as action systems. In R. Smedinga, M.P. Spathopoulos, and P. Kozák, editors, *WODES 96 — Workshop on Discrete Event Systems*, Edinburgh, 1996. IEEE Press.

[22] E. Sekerinski. Deriving control programs by weakest preconditions. Research report, TUCS, April 1996.

[23] S.A. Smolka and B. Steffen. Priority as extremal probability. In *Concur '90*, Lecture Notes in Computer Science 458. Springer-Verlag, 1990.

[24] C. Tofts. A synchronous calculus of relative frequency. In *Concur '90*, Lecture Notes in Computer Science 458. Springer-Verlag, 1990.

[25] Q.-W. Xu. On compositionality in refining concurrent systems. In J.-F. He, editor, *BCS FACS 7th Refinement Workshop*, Bath, U.K., 1996. Springer-Verlag.