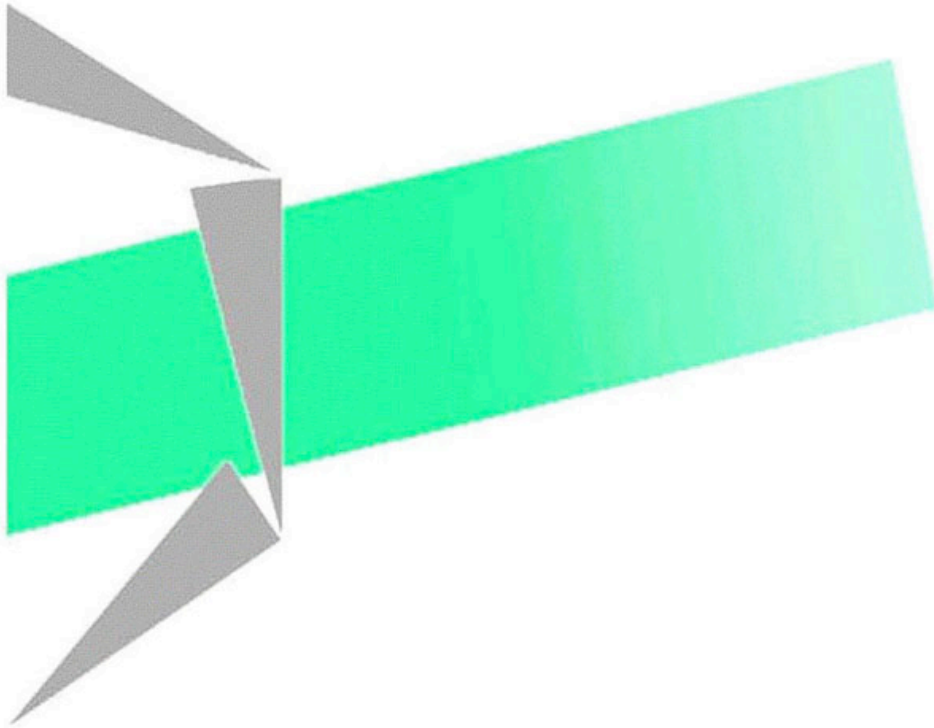


# Les cahiers du laboratoire Leibniz



## A Computation Model for Concurrent Declarative Programming

Rachid Echahed, Wendelin Serwe

Laboratoire Leibniz-IMAG, 46 av. Félix Viallet, 38000 GRENOBLE, France -  
ISSN : 1298-020X

n° 79  
Mars. 2003

Site internet : <http://www-leibniz.imag.fr/LesCahiers/>



# A Computation Model for Concurrent Declarative Programming

Rachid ECHAHED                      Wendelin SERWE  
Laboratoire LEIBNIZ – Institut IMAG, CNRS  
46, avenue Felix Viallet, F-38031 Grenoble, FRANCE  
Tel: (+33) 4 76 57 48 91; Fax: (+33) 4 76 57 46 02  
Rachid.Echahed@imag.fr    Wendelin.Serwe@imag.fr

30th August 2002

## Abstract

We propose a general computation model combining mobile processes and declarative programming languages, e.g., functional, logic or functional-logic languages. In contrast to most existing concurrent extensions of declarative languages, we distinguish clearly between the notion of processes and those underlying declarative programming languages, *e.g.*, functions and predicates. Thus, our computation is generic and may be applied to extend several kinds of declarative languages. It also extends PA process algebra in order to deal with parameter passing, mobile processes and interactive declarative programming. In our setting, declarative programs are dynamic, that is to say they may be modified thanks to the actions performed by processes. We also propose a new formal definition of a component.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Computation Model: Overview</b>	<b>3</b>
2.1	Stores . . . . .	4
2.2	Actions . . . . .	4
2.3	Processes . . . . .	6
2.4	Operational Semantics . . . . .	7
2.5	Example of the Dining Philosophers . . . . .	8
<b>3</b>	<b>Related Work</b>	<b>10</b>
3.1	Concurrent Declarative Programming . . . . .	10
3.2	Concurrent Programming . . . . .	14
<b>4</b>	<b>Computation Model: Syntax</b>	<b>16</b>
4.1	Stores . . . . .	16
4.2	Component Signatures . . . . .	19
4.3	Interactions . . . . .	24

4.4	Processes . . . . .	25
4.5	Components and Systems . . . . .	29
<b>5</b>	<b>Computation Model: Operational Semantics</b>	<b>31</b>
5.1	Execution of Guarded Actions . . . . .	32
5.2	Execution of Process Terms . . . . .	33
5.3	Combined Operational Semantics of a Component . . . . .	35
5.4	Operational Semantics of a System . . . . .	36
<b>6</b>	<b>Conclusion</b>	<b>37</b>

## 1 Introduction

Classical declarative languages, *i.e.*, functional, logic and functional-logic languages, aim to provide high-level descriptions of systems. These languages have well-known nice features (*e.g.*, abstraction, readability, compilation techniques, proof methods *etc.*) since they are based on the notions of functions and predicates, *i.e.*, well mastered mathematical concepts which have been successfully used in describing algorithms even before the invention of computers. In such a declarative language, functions are classically described by means of equations (rewrite rules) or  $\lambda$ -abstractions, whereas Horn clauses (with constraints) are a classical means for the description of predicates.

However, the concepts of functions and predicates are not sufficient to capture the whole complexity of real-world applications, where interactivity, concurrency and distributivity are needed [46, 64]<sup>1</sup>. A recent example of applications that require interactivity and concurrency are window systems. The usefulness of concurrency in the concise design of window systems has been pointed out by a number of researchers, as for instance in [52] and some concurrent extensions of declarative languages were even motivated by the design of an adequate window system, written in these languages, as for example eXene [31] for CML [54] and Haggis [26] for Concurrent Haskell [50].

The notion of *processes* has been introduced as abstraction for the expression of concurrency. Processes have been well investigated in form of *process algebras* and *process calculi*, *e.g.*, [3, 28, 37, 45, 47]. These formalisms allow the description of a system as a set of (mobile) processes that can be executed on a single computer or distributed over a network. Informally, a process is characterised by the actions that it can execute, or the interactions it may have with its environment, *i.e.*, other processes that are executing concurrently. Thus, a process is obviously different from a function or a predicate.

However, similar to classical declarative programming languages, programming models purely based on process calculi need to be extended in order to provide the notions of functions and predicates without encoding them in terms of processes.

A well-defined combination of the different programming styles mentioned above, namely declarative and concurrent programming, would allow to express most parts of a system using the most appropriate concepts. Hence there would be no need to

---

<sup>1</sup>Notice in this context, that A. Turing admits the existence of computing machines more powerful than “Turing Machines”, when he states in [63]: “We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine.”

*encode* one concept by another, and the different aspects of the problem could be expressed directly and, in consequence, more clearly. Furthermore, a clear separation and structuring of these different concepts, abstractions or notions, namely functions, predicates and processes, should lead to well structured, readable and understandable programs which are consequently easy to maintain.

Numerous proposals to the integration of declarative and concurrent programming exist. Common to most of them is that they do not distinguish clearly between processes and the concepts underlying the declarative language, but rather try to *encode* processes in terms of the latter [2, 5, 10, 11, 12, 13, 16, 17, 25, 30, 35, 39, 40, 49, 50, 54, 56, 57, 61, 62]. Thus each of these approaches seems to be tailored for a specific language rendering the extension to a general computation model not straightforward.

This paper aims at a new computation model, or a new general framework of programming languages, providing a component based approach for constructing systems. These programming languages are a combination of declarative, *i.e.*, either functional, logic or functional-logic, programming with concurrency, expressed by means of mobile processes. We suggest to model a system by a set of interacting components, and provide a clear, formal definition of a component for our model.

A main feature of the suggested computation model is the clear distinction between, on the one hand, concepts which are definable in classical declarative languages, such as functions, predicates or constraints and (mobile) processes on the other hand. Thus the merit of our contribution is to propose a new computation model where each part of a mobile, concurrent, functional and/or logic application can be described by the most appropriate known theoretical concept, instead of encoding all these different concepts in a sole framework. Furthermore, our computation model is a conservative extension of, on the one hand, declarative programming languages and, on the other hand, process calculi or process algebra.

Theoretically, our computation model can be characterised as a new (modal) theory whose models are Kripke-structures. Practically, we provide a new full and rigorous combination of programming paradigms, providing the respective advantages of functional, logic, functional-logic, concurrent, mobile and component-based programming in addition to the advantages proper to the combination, in the same way as it was already the case for the integration of functional and logic programming.

The rest of the paper is organised as follows. In the following section, we present the broad outlines of our computation model. In section 3 we compare our model with some related work. The formal presentation of our computation model by its syntax and semantics is subject of sections 4 and 5. Finally, section 6 concludes with some perspectives and enhancements of the model.

## 2 Computation Model: Overview

We suggest to model a system as a set of interacting *components*, which may be distributed over a network or reside on a single computer [21, 22]. These components capture the different, clearly distinguishable entities of a system and interact with each other by exchanging messages. Each component is identified by a component-name (also called storename) *sn* which can be seen as the address of the component. Inter-

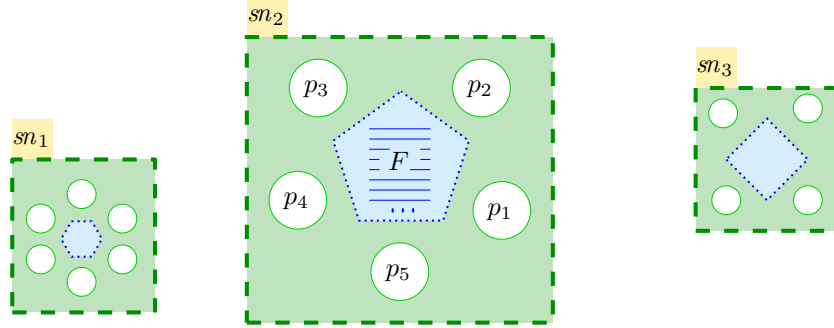


Figure 1: Execution Model of a System

nally, a component is organised as a set of *processes*  $p_i$ , *i.e.*, a concurrent program, and a *store*  $F$ , *i.e.*, a traditional declarative program, which can be seen as a pair of a *signature* and a set of formulæ or *rules* describing a (static) theory. The processes  $p_i$  communicate by modifying the stores, *i.e.*, by altering, in a possibly non-monotonic way, the current theories described by the stores, for example by simply redefining constants (*e.g.*, adding a message in a queue) or more generally by adding or deleting formulæ in  $F$ . We call these modifications of the stores *actions* and allow their definition by the programmer [23]. Interaction between components is based on asynchronous message passing, where the messages correspond to actions to be executed on the remote store. Figure 1 shows the execution of a system of three components with storenames  $sn_1$ ,  $sn_2$  and  $sn_3$ .

In the remainder of this section we give an overview of our model for a single component. The full model, taking into account the interactions between components is presented by its syntax and semantics in sections 4 and 5.

## 2.1 Stores

Our approach to combine declarative programming and concurrency is generic in the sense that it is independent from the actual declarative language used for the description of the store. In fact, we only require that programs in a declarative language can be seen as a pair of a *signature*  $\Sigma$  and a set of formulæ or *rules*  $\mathcal{R}$ , *i.e.*,  $F = \langle \Sigma, \mathcal{R} \rangle$ , and that the operational semantics (of the declarative language) allows the test for *validity* of a term (of the special sort **Truth**). Notice that most (declarative) programming languages fit into this framework.

## 2.2 Actions

Actions are the principal constituents of processes, as it is witnessed by the fact that the semantics of a process is, loosely speaking, defined by the sequences (in linear time semantics) or graph (in branching time semantics) of actions that the process might execute. Most classical process algebras consider *abstract* actions, in the sense that actions are simply elements of a vocabulary (of actions) without further specification, besides a relation which associates pairs of actions in order to model synchronisation

and communication. In our model, the execution of an action has the observable effect of modifying the store of the component. Thus we need to specify the notion of actions further to take into account these modifications.

Since an elementary action transforms a store into another one, we are led to define elementary actions as total recursive functions from (well-formed) stores to (well-formed) stores. Thus, actions are placed on a meta-level with respect to the store, and we suppose that we dispose of all the necessary (abstract) data types for expressing actions. We require an elementary action to be a *recursive* function in order to ensure the termination of its execution. The requirement of *totality* guarantees that the elementary action can be applied to any store. We require further, that all stores in the range of an elementary action (which is applied to a well-formed store) should be well-defined stores, in particular, they should be well-typed. These conditions together ensure that the execution of an elementary action will never produce an error during the runtime of the system. That is to say, all actions can be executed and terminate, as well as that the store of a component corresponds (at any moment during the execution) to a well-formed (declarative) program.

A similar view of (elementary) actions can be found for instance in Concurrent Haskell (CH), where the monadic I/O operations, *i.e.*, functions of type  $\text{IO } \mathbf{t}$ , are considered as *state transformers* and are called *actions* [50, section 2.1]. A noteworthy difference<sup>2</sup> is that actions in CH are functions, and as such they return a value, in addition to the implicit “world” parameter. On the other hand, our (elementary) actions are the basic elements of processes and describe only the effect on the store without returning any value.

**Examples of Elementary Actions.** For the rest of this paper, we suppose that we are given a set  $A$  of predefined (elementary) actions, namely `tell`, `del`, `:=` and `new`. The *definition* of actions by the programmer is discussed in more detail in [23, 59].

Intuitively, `tell( $f$ )` (respectively, `del( $f$ )`) add (respectively, remove) a rule (or more generally, a formula)  $f$  to (respectively, from) the store (or program). Thus the profile of `tell` (respectively, `del`) is  $\text{tell} : \text{rule} \rightarrow \text{store} \rightarrow \text{store}$ , where *store* (respectively, *rule*) is the type of meta-representations of stores (respectively, rules). Furthermore, we abbreviate for the rest of this paper `tell( $t \rightarrow \text{TRUE}$ )` (where  $t$  is a (meta-representation of a term of sort `Truth`) to `tell( $t$ )`.

Certainly the most common elementary action is assignment `:=( $c, v$ )`, also written more familiarly in infix-notation as  $c := v$ . This action, which is the only action considered in classical imperative programming languages, changes the definition of the constant<sup>3</sup>  $c$  to the value  $v$ , *i.e.*, a (meta-representation of a) term. A reasonable requirement on the assignment action is to normalise the new value of the constant with respect to the current store before adding the new definition.

Besides actions modifying the rules or formulæ of a store, we need also actions for the modification of the *signature*. The creation of new operator (or function) symbols is handled by the elementary action `new`. The intuitive meaning of an elementary action

---

<sup>2</sup>Besides the fact that the (sort) *state* used in CH is not further specified, whereas in our model a state is partly defined as a store, *i.e.*, a declarative program (for more details, see the definition of the operational semantics in section 5).

<sup>3</sup>In imperative languages,  $c$  is traditionally considered as a *variable*.

$\text{new}(f, s)$  is to enrich the signature of the store with the operator symbol  $f$  of profile  $s$ . Besides the  $\text{new}$ -action creating new function symbols, we may need similar actions for the other kinds of symbols in a signature, *i.e.*, sorts, constructors, *etc.* Obviously, the exact set of needed  $\text{new}$ -actions depends on the (declarative) language used for the description of the store.

## 2.3 Processes

The processes of a component are specified in the style of a process algebra, see for instance [3, 28]. We introduce first the notion of a guarded action and present in a second step the process terms of our computation model. Finally, we present the definition of (recursive) processes.

**Guarded Actions.** As we have already mentioned before, actions are essential for the definition of processes. In fact, the basic processes in our computation model are *guarded actions*. A guarded action  $\alpha$  is a pair, consisting of a *guard* and a sequence of (calls to) *elementary actions* (with parameters):

$$[g \Rightarrow \mathbf{a}_1(t_{1,1}, \dots, t_{1,l_1}); \dots; \mathbf{a}_k(t_{k,1}, \dots, t_{k,l_k})]$$

such that the guard  $g$  is a (conjunction of) expression(s) of sort **Truth** and, for all  $i \in \{1; \dots; k\}$ ,  $\mathbf{a}_i(t_{i,1}, \dots, t_{i,l_i})$  is a well formed call to the elementary action  $\mathbf{a}_i$ .

To shorten the notation, we sometimes omit the parameters of the (elementary) actions and abbreviate a guarded action to  $[g \Rightarrow \mathbf{a}_i]$ . Similarly, we call in the sequel both, guarded and elementary actions, just “actions”, whenever there is no risk of confusion.

Similar to the “guarded commands” of [20], the execution of the sequence of (elementary) actions  $\mathbf{a}_i$  in a guarded action  $[g \Rightarrow \mathbf{a}_i]$  is only possible if the current theory described by the store allows to prove that the guard  $g$  is valid.

As the execution of actions modifies the store, the order in which the actions are executed may influence the resulting store. To obtain a *deterministic* execution of a guarded action, we follow the tradition in imperative programming and require the sequential execution (from left to right) of the elementary actions of a guarded action.

**Process Terms.** Elementary (or basic) process terms are the execution of (guarded) actions and process calls. The predefined process **success** represents the process which (immediately, *i.e.*, without execution of an action) terminates successfully. As usual in process algebra (see, *e.g.*, [28]), we provide some operators for combining processes: parallel ( $\parallel$ ) and sequential ( $;$ ) composition, nondeterministic choice ( $+$ ) and choice with priority ( $\oplus$ ). Hence we define a *process term*  $p$  as a “well-formed” expression according to the following grammar:

$$p ::= \text{success} \mid [g \Rightarrow \mathbf{a}_i] \mid q(t_1, \dots, t_m) \mid p; p \mid p \parallel p \mid p + p \mid p \oplus p$$

In the case of a process call  $q(t_1, \dots, t_m)$ , we require that the parameters  $t_1, \dots, t_m$  of the process  $q$  are of sorts corresponding to the profile of  $q$  (and that the arity of  $q$  is  $m$ ). The operator of choice with priority  $\oplus$  is not very common, but we found

it necessary to model critical applications where nondeterminism is not acceptable [1]. The intended meaning of the process term  $q_1 \oplus q_2$  is: “execute the process  $q_2$  only if the process  $q_1$  cannot be executed”, *i.e.*, the process  $q_1$  has a higher priority than the process  $q_2$ .

**Process Definitions.** The behaviour of processes is defined by means of *process definitions*. However, the recursive definition of the behaviour of a process requires some care in order to avoid pathological cases, as for example processes with an infinite branching degree. To avoid this kind of problems, process definitions are usually required to be *guarded*<sup>4</sup>, that is to say, a recursive call to a process has to be preceded by the execution of an action.

A process is defined by a set of rules, clauses<sup>5</sup> or *guarded commands* ordered by priority. Each command consists of a guarded action and a restricted process term. Using the previously introduced notations, a process can thus be defined by a phrase of the following form:

$$q(x_1, \dots, x_m) \Leftarrow \bigoplus_{i=1}^n ([g^i \Rightarrow a_{j_i}^i]; p_i)$$

Intuitively, the operational behaviour of a process call  $q(t_1, \dots, t_m)$  is similar to the alternative construct of the guarded command language of [20]. That is to say, we have to evaluate which of the guards of the commands defining  $q$  are valid, and then to choose among them the one with the highest priority. Choosing a command means to atomically execute the sequence of elementary actions associated with the guard and afterwards to behave like the associated restricted process term.

## 2.4 Operational Semantics

The operational semantics of a component integrates two orthogonal aspects, namely the use of the store as a classical declarative program (*i.e.*, evaluation of expressions and goal solving) and the execution of processes. Informally, the operational semantics of a component is defined by means of a transition system the states of which consist of a store and a process term. Execution of actions corresponds to state transitions, and in every state the (current) store can be used as a classical declarative program.

Hence, our computation model is a conservative extension of declarative programming, since a component without any processes corresponds to a declarative program in the classical sense. On the other hand, when abstracting from the effects of the actions, *i.e.*, the modifications of the stores, and considering only the names of actions that are executed, our computation model is a classical process algebra.

---

<sup>4</sup>Notice, that this notion of guard is to be distinguished from the guards in guarded actions, since the guard in a guarded process is an action, whereas the guard in a guarded action is a term (of sort **Truth**) of the store.

<sup>5</sup>In analogy with the definition of predicates by a set of clauses in logic programming; for the same reasons we use the symbol  $\Leftarrow$  in process definitions.

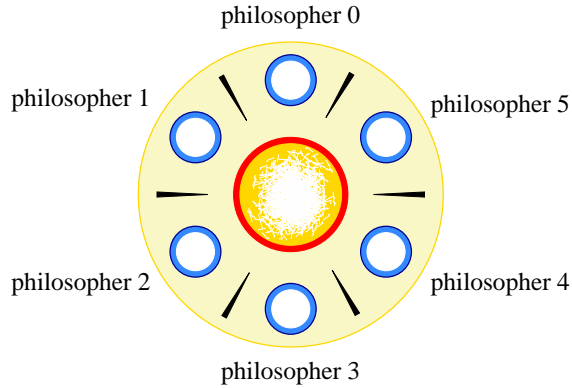


Figure 2: Dining Table for Six Philosophers

## 2.5 Example of the Dining Philosophers

We conclude this brief overview of our computation model by the presentation of a solution to the “Dining Philosophers” problem [19]. The problem concerns the life of some (Chinese) philosophers, which alternate between thinking and eating. These philosophers live in a same room, and are seated around a round table, in the middle of which stands a large bowl of rice (with the enjoyable property of containing always a sufficient amount of well-prepared rice). Figure 2 shows such a table for six philosophers. As usual, a philosopher needs two chop sticks to eat. Unfortunately, there are only as many chop sticks as philosophers, so that the philosophers have to share the sticks with their neighbours. Furthermore, the philosophers are not allowed to exchange sticks across the table.

We model the situation with two predicates on natural numbers, *i.e.*, functions with the profile  $Nat \rightarrow \mathbf{Truth}$ , which have to be added to a signature of natural numbers, namely  $stick(x)$  and  $is\_eating(y)$ . The former represents the fact that stick  $x$  is lying on the table, and the latter is true whenever philosopher  $y$  is eating. Hence the initial store is an extension of a theory for natural numbers with  $n$  rules stating the presence of the  $n$  sticks on the table, *e.g.*,  $stick(i) \rightarrow \mathbf{TRUE}$ , for  $i \in \{0; \dots; n - 1\}$ .

The behaviour of a philosopher can be modeled by a pair of two processes, shown in table 1. We use two processes to model the two states for a philosopher: either the philosophers thinks or eats. These processes take two arguments of sort  $Nat$ , corresponding to the number of the philosopher and to the total number of philosophers in the system. In order to start to eat, a thinking philosopher has to execute a guarded action. The guard  $stick(x) \wedge stick(((x+1) \bmod n))$  ensures that the needed sticks are both available, and the three elementary actions modify the theory by removing the two sticks, and by adding the eating philosopher. The guard  $\mathbf{TRUE}$  in the action of the process eats reflects that an eating philosopher can decide to stop eating at any moment.

The initial process term is the parallel composition of  $n$  thinking philosophers,

$\text{thinks}(x, n) \Leftarrow \left[ \begin{array}{l} \text{stick}(x) \quad \wedge \\ \text{stick}(((x+1) \bmod n)) \Rightarrow \text{del}(\text{stick}(((x+1) \bmod n))); \\ \text{tell}(\text{is\_eating}(x)) \end{array} \right]; \text{eats}(x, n)$
$\text{eats}(x, n) \Leftarrow \left[ \begin{array}{l} \text{del}(\text{is\_eating}(x)); \\ \text{TRUE} \Rightarrow \text{tell}(\text{stick}(x)); \\ \text{tell}(\text{stick}(((x+1) \bmod n))) \end{array} \right]; \quad \text{thinks}(x, n)$

Table 1: Process Definitions for the Dining Philosophers

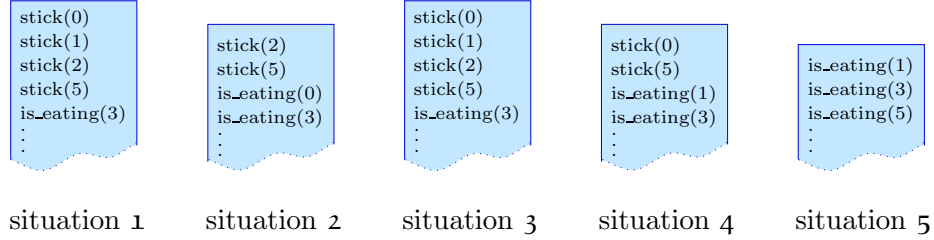


Figure 3: Possible Execution Sequence for Six Philosophers

numbered from 0 to  $n - 1$ , *i.e.*,

$$\text{thinks}(0, n) \parallel \dots \parallel \text{thinks}((n - 1), n)$$

Thus, at the beginning of the execution of this system, we observe the situation shown in figure 2. Thus, the solving of the goal  $\text{is\_eating}(x)$  is initially impossible: There exists no substitution (for  $x$ ) such that  $\text{is\_eating}(x)$  could be reduced to `TRUE`. However, during the execution, the philosophers start and stop eating (respectively, thinking), such that we might observe the sequence of theories (or stores) shown in figure 3.

First, philosopher number 3 starts eating (situation 1). In situation 2, philosopher number 0 joins him. Thus, in situation 2, the current theory has been changed such that the solving of the goal  $\text{is\_eating}(x)$  returns the two answer substitutions  $\{x \mapsto 0\}$  and  $\{x \mapsto 3\}$ . Philosopher number 0 has stopped eating in situation 3, whereas philosopher number 3 is still hungry and continues to eat. He is joined by philosopher number 1 in situation 4 and by philosopher number 5 in situation 5. Hence, the solving of the goal  $\text{stick}(y)$  in situation 5 is impossible, since all sticks are in use, and no stick is available on the table.

Notice that our solution of the Dining Philosophers is straightforward. Indeed, the store is a simple description of the situation, and the two processes are obtained immediately from the formalisation of the behaviour of a philosopher by an extended automaton. Notice further, that our solution does not need an additional semaphore in order to avoid dead-locks, and that we provide a generic description of a philosopher as a process.

## 3 Related Work

As already stated in the introduction, numerous proposals combining declarative and concurrent programming exist. Consequently, we cannot give a complete survey of all of them in this paper, but restrict ourselves to compare our computation model to some of the closely related proposals.

### 3.1 Concurrent Declarative Programming

The main difference with existing concurrent extensions of declarative programming languages is that we distinguish clearly between the notions underlying the declarative language and processes. Our motivation for this separation is to avoid the need to *encode* one concept by another. We strongly believe that this leads to more structured programs which in consequence are easier to write, read and understand. Furthermore, our approach is general in the sense that it can be applied to extend most declarative languages with concurrency, since we do not rely on particular concepts to be available in the declarative language, but provide processes as an *additional* notion to those which are already offered by the declarative language. In fact, our approach requires only the conditions mentioned in section 4.1.

However, besides this fundamental difference, there are some common points, which we point out for some selected examples of concurrent extensions of declarative programming languages.

#### 3.1.1 Concurrent Logic Programming

The commands defining our processes (see definition 13) are syntactically similar to the clauses defining predicates in logic programming, with the difference that we provide more than just conjunction to combine the process calls of the bodies of the “clauses” and that we explicitly order the clauses defining a process by priority. Therefore, definitions of processes are less “declarative” as definitions of predicates in logic programming. However, a process is a different concept which has to be distinguished from predicates, and the order of the execution of actions by a process matters, requiring a more imperative description of processes.

The built-in “predicates” `assert` and `retract` of Prolog [17] are similar to our actions. In fact, CIAO [11] and ESP [12] interpret these predicates as Linda [32] coordination primitives on a database of atoms. While this view does not improve our understanding of the semantics of this mix of predicates and actions, [11] presents nice implementation techniques for the execution of these particular actions.

The execution model of our components is closely related to the one of concurrent constraint programming (ccp) [56], where a (constraint) store is shared by a number of processes. The main difference between our model and basic ccp is that the store of ccp can only be modified in a *monotonic* manner.

Most of the semantics proposed for the family of ccp-languages consider the resulting, final store as the semantics of a process. Thus most of these semantics consider only *finite*, terminating executions. The only semantics for ccp considering explicitly infinite computations we are aware of are presented in [15] and [25], but even these approaches

are concerned with the final, resulting store, which is modeled as a least fix-point<sup>6</sup>. We prefer a trace based semantics in order to model non-terminating processes controlling external devices.

Several non-monotonic extensions of `ccp` have been suggested, either by providing new built-in actions [16, 13], the use of modifiable cells [61] or by using non-monotonic logics, as a logic with defaults [57] or linear logic [58, 5, 25]. The user-definable actions in our computation model allow a greater degree of flexibility than the specific new built-in primitives of [16] and [13]. Two of the solutions to the problem of the Dining Philosophers of [16, 13] use an additional arbiter process (second solution of [16] and the one of [13]). The first solution of [16] uses a specific constraint system which allows to model the atomic removal of two forks<sup>7</sup>. Similarly, the actions provided by the approaches based on linear logic are in our opinion less intuitive, since they rely on an *implicit* removal (of the constraints used for the proof of entailment of the guard), instead of specifying *explicitly* the constraints to be removed (as in our computation model). The semantics of a process in [5] is defined by a *history*, which can be seen informally as a graph representing *all* possible executions and taking into account the causal dependencies between occurrences of basic actions. In contrary to most semantics for process calculi, this semantics does not need to interleave all actions in a sequential manner and is thus a truly concurrent semantics. Although this semantics considers only *finite*, *i.e.*, either successful terminating or deadlocking, computations, [5] sketches a solution to the (non-terminating) problem of the Dining Philosophers (see section 2.5), using an additional semaphore<sup>8</sup>. Since [25] (similar to [58]) provides only an operator for prefixing a process with a guard, sequential composition, *i.e.*, imposing an order on the execution of tell operations, has to be encoded. As an example, consider the solution suggested for the Dining Philosophers in [25, section 2.2] which is defined as follows:

```
philosopher(I,N) =
  fork(I) ⊗ fork(I+1 mod N) →
    (tell(eat(I,N)) ||
     eat(I,N) → (tell(fork(I) ⊗ fork(I+1 mod N)) → philosopher(I,N)))
```

where the sequential composition “eat and then think” is encoded as

---

<sup>6</sup>Notice that this requires a monotone evolution of the store.

<sup>7</sup>Informally, the predicate  $use(x, leftfork)$  (respectively,  $use(x, rightfork)$ ) models the fact that philosopher  $x$  uses the fork on his left (respectively, right). The constraint system is such that

$$\begin{aligned} use(x_1, leftfork) \wedge use(x_2, rightfork) &= use(x_2, leftfork) \wedge use(x_3, rightfork) = \dots \\ &= use(x_n, leftfork) \wedge use(x_1, rightfork) = false \end{aligned}$$

Thus, using an atomic *atell*, a philosopher can only tell the use of his both forks, otherwise the constraint store would become *false*.

<sup>8</sup>[5] sketches two further solutions to the problem of the Dining Philosophers. The first one is a translation to `ccp` (with an *atomic atell*) of the one suggested in [60, page 1245], which is similar to ours, since a philosopher can take both forks (or sticks) in a single atomic step. The synchronisation is expressed by incrementally instantiating streams associated to the forks. The second solution is a translation to standard `ccp` of the one presented in [55] which is rather long, complicated (70 lines of code, which is still incomplete) and requires the use of an additional semaphore for ensuring that no deadlocks occur. Both solutions do not support interactive goal-solving.

$$\text{tell}(\text{eat}(\text{I}, \text{N})) \parallel \text{eat}(\text{I}, \text{N}) \longrightarrow \text{P}$$

Notice that this program is close to our solution (see section 2.5) since the atomic removal of both forks (or sticks) is possible, and that no additional semaphore is needed. Indeed, the main difference is the implicit removal of the predicates `fork(I)`.

The notion of *ports* as a many-to-one communication medium has been introduced in AKL [39]. It is argued that the introduced port primitives have a “logical reading” (as a special (port-) constraint) and preserve the monotonicity of the constraint store. In our non-monotonic setting, we can provide the behaviour of ports via appropriate (elementary) actions. For instance, we can model the streams as lists of messages. The reception of a messages corresponds to access the head of the list, and sending amounts to simply add a new message to the end of the list. Notice that we could easily specify other communication schemes, for example we might want to introduce priorities of the messages. In this case, we would just have to modify the definition of the action `send`.

Similar to logic programming, processes in Curry [36] are represented by constraints, using a concurrent interpretation of conjunction. As a means for communication, the notion of ports of [39] has been introduced in Curry and been extended to *named ports* which allow for distributed programming [35]. Our communication scheme allows to model the behaviour of ports (by appropriate actions) so that we do not need to introduce them into our computation model. Due to the use of monadic I/O, processes in Curry are not allowed to perform I/O actions, *i.e.*, to interact with the external world. This implies that Curry provides two different operators for sequential composition, namely `>>=` (which is used for the composition of action in the `I0` monad) and `&>` (which corresponds to the sequential interpretation of conjunction used for modelling the sequential composition of processes). The example program of the Dining Philosophers which comes with the distribution of PAKCS<sup>9</sup> uses an additional semaphore to ensure that only  $n - 1$  philosophers are seated around the table at the same time such that the system does not deadlock.

### 3.1.2 Concurrent Functional Programming

In most concurrent extensions of functional programming languages, processes are encoded by means of functions. Therefore, a process is required to return a value, even if the value is discarded (or simply does not matter) in most of the proposals we are aware of [53, 2, 50, 62, 54, 42].

The design of Concurrent Haskell (CH) [50] was guided by the research for a “minimal” set of primitive operations which would allow to provide concurrent programming in the functional language Haskell, such that, using the rich set of abstraction features of functional programming, more friendly abstractions can be defined. Thus, the confusion between the notions of functions and processes is one of the design principles of CH. In fact, only the type system allows to distinguish between a (pure) function and a state transformer (or process), *i.e.*, a function the result type of which is necessarily of the form `I0 t`<sup>10</sup>. However, the introduction of concurrency renders the interpre-

<sup>9</sup>PAKCS is the acronym for the Portland Aachen Kiel Curry System which is available for download at the URL <http://www.informatik.uni-kiel.de/~pakcs>.

<sup>10</sup>Since the value returned by a function representing a process is discarded, the type `t` is mostly the empty type `()`.

tation of monadic I/O as abstract descriptions of state transformers “untenable” [50, section 2.1] for CH, since the execution of the side-effects denoted by the actions cannot wait until the program has finished (and the description of the actions to be executed has been computed completely). Thus, the operational semantics of Haskell has to be extended.

The operational semantics of CH is stratified in two layers, namely the deterministic reduction of expressions (*i.e.*, the operational semantics of Haskell) and the concurrent reaction modeling the execution of processes, or reduction of functions of type  $\mathbf{IO} ()$ . While this separation in two levels is similar to our operational semantics as presented in section 5, there are several differences. First, CH does not provide operators for sequential composition and choice between processes, since these operators are not *primitive* in the sense that they can be simulated using the operators available in CH. We have included these operators, since we did not have the goal of designing a *minimal* extension to a particular language nor a minimal calculus allowing to model any problem, but rather searched to combine the best features of both, declarative programming and process calculi, where these operators are widely used. Second, communication between processes in CH uses (besides the implicit synchronisation on shared expressions due to the lazy evaluation strategy)  $\mathbf{MVars}$ , *i.e.*, mutable variables which are protected by semaphores. In the operational semantics,  $\mathbf{MVars}$  are modeled as a special kind of process. Thus this communication scheme is a particular case of the communication using a store, since the parallel composition of  $\mathbf{MVars}$  can be considered as a store (containing only atomic formulæ). Third, our operational semantics presents the execution of all actions or state-transformers at the level of the processes, whereas the operational semantics of CH integrates at least<sup>11</sup> the description of the operational behaviour of the operator of sequential composition, *i.e.*,  $\gg=$ , into the operational semantics of the declarative program. Last, but not least, CH does not provide a symmetrical operator for parallel composition of processes (as is usual in most process calculi), but a primitive action which has the side-effect of launching a process<sup>12</sup>.

Therefore, the main difference between our computation model and CH is that we provide operators taken from process calculi for the description of processes, instead of encoding processes as a particular kind of functions. Thus we allow the direct use of the appropriate description tools for each concept, without the need of encoding them. For instance, our solution to the example of the Dining Philosophers (see section 2.5) needs the atomic test of two guards, which is not directly provided in CH.

CML [54] differs from our model in two basic design choices. First, communication in CML is based on message passing, whereas our processes share a common store. Second, our programming model is asynchronous, while processes in CML synchronise on *events*, a new data type introduced in CML. Furthermore, since the behaviour is specified by functions which might be defined by means of processes the definition of which might use functions, CML clearly does not distinguish between processes and functions. Notice that the solution to the problem of the Dining Philosophers given in [54, page 186] as an example for the implementation of the Linda [32] coordination

---

<sup>11</sup>Due to lack of space, the other primitive actions are not considered in [50, section 6].

<sup>12</sup>It is argued in [50, section 2.2] that a symmetrical fork (as for instance the primitive `symFork` of [40]) would have forced the synchronisation on the termination of the forked process, that is to say, the introduction of a general sequential operator.

principles in CML needs an additional semaphore to ensure that at most  $n - 1$  philosophers are seated around the table, since there is no direct support for the atomic test and update of two tuples, as we used in section 2.5.

### 3.2 Concurrent Programming

The description of processes in our computation model is based on process calculi. Hence, by means of appropriate actions we can model most of the communication mechanisms of these calculi. However, our computation model distinguishes clearly between the notions of processes and those underlying declarative languages, such as functions and predicates. This distinction does not exist in process calculi, and functions or predicates have to be *encoded*. We consider therefore our computation model to be more convenient for programming, since these different notions can be expressed directly using an appropriate formalism. In this section we compare our computation model to some programming languages based on process calculi. Following the calculi they are based on, most of these languages require the *encoding* of the notions of functions and predicates by means of processes.

Our action `new` (which allows the dynamic creation of channels)<sup>13</sup> allows to model mobility in the same way as the (asynchronous)  $\pi$ -calculus [47], *i.e.*, by passing communication links. Since the encoding of functions by means of processes in the  $\pi$ -calculus is rather complicated and not very intuitive, an integration of the  $\lambda$ -calculus and the  $\pi$ -calculus has been proposed [7], by combining the operational behaviour of functions and processes, whereas in our model, functions and processes are clearly distinguished notions. Furthermore, our model is a conservative extension of declarative programming, such that interactive goal solving with respect to the current store is possible.

The programming language `Pict` [51] is based on the asynchronous  $\pi$ -calculus. Contrary to `Pict` (and the  $\pi$ -calculus), our guards allow the atomic reception on several channels, whereas in the  $\pi$ -calculus processes can only wait on a single channel. Extensions of the (asynchronous)  $\pi$ -calculus without this restriction are the join-calculus [30] and  $\mathcal{L}_\pi$  [10].

`jocaml` [29] is a language based on the join-calculus where processes can be seen as communicating via a multiset of messages: sending a message corresponds to place it in the multiset. The “joint reception” of several messages is blocking and removes the received messages from the multiset. Thus broadcast is not provided directly and has to be encoded as in any language based on the  $\pi$ -calculus. Since `jocaml` is implemented on top of `ocaml` [42], `jocaml`-programs can use the facilities of `ocaml` for the definition of data structures and functions<sup>14</sup>. However, we are not aware of a complete description of the theoretical foundations of this integration.

In  $\mathcal{L}_\pi$  [10], processes communicate (as in the join-calculus) via a multiset, but additionally may have guards, *i.e.*, a  $\mathcal{L}_\pi$ -process that, when executed in an encapsulated environment, has to terminate successfully. Using these guards, the solution to the

<sup>13</sup>Together with the parameterised sort `Name` introduced in section 4.1.2, which allows channel names to be passed.

<sup>14</sup>“To explore the expressive power of message-passing in `jocaml`, we now consider the encoding of some data structures. In practice however, one would use the state-of-the-art built-in data structures inherited from `ocaml`, rather than their `jocaml` internal encodings.” [29, introduction of chapter 1.6]

problem of the Dining Philosophers given in [9, section 2.1, pages 23 – 24] allows a philosophers to take two sticks in a single atomic action, similar to our solution (see section 2.5). However,  $\mathcal{L}_\pi$  does not distinguish between predicates and processes, and like all the other dialects of the  $\pi$ -calculus mentioned above does not support interactive goal solving.

Similar to our computation model, a system in KLAIM [49] is composed of several components (called “nets” in KLAIM) which themselves are structured by means of concurrent, parameterised processes. Therefore the actions executed by processes in KLAIM are located, *i.e.*, paired with the location (of the net) where they are to be executed. This is similar to our pairs of storenames and elementary actions. However, the stores of KLAIM are not declarative programs, but multisets of tuples which are accessed or selected (as in Linda [32]) by means of pattern matching. A further similarity is the separation of the operational semantics in two levels, corresponding to the locations (components) and the nets (system). However, while KLAIM provides a global transition system describing the semantics of a system, we prefer to view a system as a parallel composition of several transition system, in order to reflect that we cannot know the states of all components at the same moment. Furthermore, KLAIM does not distinguish between tuples and processes: on the one hand, KLAIM supports the notion of active tuples, *i.e.*, tuples representing processes<sup>15</sup>, and on the other hand, the operational semantics of KLAIM presented in [49] models tuples as processes. This way of modelling messages as processes can also be found in the asynchronous  $\pi$ -calculus or the join-calculus calculus where sending a message corresponds to spawn a parallel process which synchronises on the reception of the message and terminates. Finally, we have voluntarily restricted ourselves in this paper to the precise description of components and their interactions, and model system as a static set of components. KLAIM does not have this restriction and allows the dynamic creation of locations (*i.e.*, components) by means of particular builtin actions. In our opinion, the creation of components should be distinguished from actions, since the former modify the *set* of stores in the system, whereas the latter modify stores, *i.e.*, *members* of the before mentioned set.

The combination of algebraic specification with a process calculus similar to CCS [44] and CSP [37] provided by LOTOS [38] distinguishes between functions and processes. However, in contrary to our proposal, the definitions of the functions cannot be changed, and the store is mainly used to specify the types of the messages. The communication mechanism of LOTOS by synchronisation on ports has to be simulated in our computation model. On the other hand, a broadcast is natural in our model, but rather difficult to obtain in LOTOS.

Similar to our model, algebraic state machines [8] distinguish between the static and dynamic parts of a system, in order to enhance the readability of specifications. Thus, algebraic state machines also use different layers in the description of a system. Indeed, the states of an algebraic state machine are algebraic specifications, and the transitions between states are expressed by particular transition rules or axioms. Compared to our computation model, algebraic state machines are closer to *specifications*, *i.e.*,

---

<sup>15</sup>Notice that this leads to two different ways of introducing a concurrent process: Either by putting an active tuple into the tuple space by means of the tuple space operation **eval** or by the operator **||** of parallel composition of processes.

they are not necessarily executable. This is witnessed by the fact that the transition rules are considered as specifications of a logical relation between states, and arbitrary logical formulæ are allowed in the pre- and post-conditions of the transition rules. Furthermore, algebraic state machines do not use process algebra combinators for their composition, but a single composition operator which is based on the data flow using the input and output channels, considering single algebraic state machines as black boxes. Thus, notions of processes (or algebraic state machines) and components are not as clearly distinguished as in our model. Finally, the use of process algebraic operators allows us to express the dynamic creation of processes, since we allow arbitrary process terms in the rules of process definitions, whereas the transitions of algebraic state machines cannot change the number of processes.

## 4 Computation Model: Syntax

In this section we present the syntax of our component-based approach to concurrent declarative programming [21, 22]. As mentioned in section 2, we model a system as a set of *components*, where each component is internally composed of a *store*  $F$ , *i.e.*, a declarative program, and a set of *processes*  $p_i$ , interacting via the modification of the store by means of the execution of *actions*. To distinguish between the different components of a system, we attribute to each component of the system an identifier or name, called *component name* or *storename*. In the system of figure 1, the storenames of the three components shown are  $sn_1$ ,  $sn_2$  and  $sn_3$ .

In the rest of this section we present the different parts of a component one by one in more detail. First we consider the level of the stores separately, that is to say we present the requirements on a declarative language which is to be used for the description of stores. Then we introduce in section 4.2 the notion of a component signature which declares all the symbols necessary for the description of a component. Section 4.3 discusses the interactions between components. The definition of processes is presented in section 4.4. Finally, section 4.5 gives the complete definition of a component, and briefly presents the construction of systems from a set of components.

### 4.1 Stores

The store of a component corresponds to a classical declarative program, *i.e.*, a set of formulæ or rules which can be seen as a theory description. Thanks to this general view of stores, our approach to combine declarative programming and concurrency is generic in the sense that it is independent from the actual declarative language. Consequently, (almost) any (pure) declarative language can be used for the description of a store. In this section, we present the required general properties of stores.

#### 4.1.1 General Properties of Stores

In our computation model, a store is a theory description that is shared by the processes of the component. Hence, the store of a component can be considered as a “knowledge base”, modeling information about the state of the component and its environment, that is exploited by the processes. Consequently, the processes need to access the

information contained in the store, as well as to modify the store. Access to the information uses the operational semantics of the declarative language the store is written in. The modification of the store is performed by the execution of actions and is presented in the following section dedicated to actions. In order to define the processes in a language independent way, we make the simplifying assumption that all stores are constructed from a signature and a set of rules. Thus we get the following definition.

**Definition 1 (store).** *A store is a classical (declarative) program  $F = \langle \Sigma, \mathcal{R} \rangle$  (written in the language  $\mathcal{L}$ ), composed of a signature  $\Sigma$  and a set of rules (also called phrases or formulæ)  $\mathcal{R}$ . A signature  $\Sigma = \langle S, \Omega \rangle$  is a pair of a set of sorts  $S$  and a ( $\overline{S}$ -indexed) family of operator, function or predicate symbols, such that  $\Sigma$  contains at least the sort **Truth** with its constructor **TRUE**.  $\overline{S}$  denotes the set of all types that can be constructed from the set of basic sorts specified by the set  $S$ <sup>16</sup>. We note the ( $\overline{S}$ -indexed) family of sets of terms for a signature  $\Sigma$  and variables  $X$  as  $T^{\mathcal{L}}(\Sigma, X)$ . Furthermore, we have a decidable predicate  $eval_{\mathcal{L}}(F, t)$  (also written as  $F \vdash_{\mathcal{L}} t$ ), which holds if the term  $t$  of sort **Truth**, i.e.,  $t \in T^{\mathcal{L}}_{\text{Truth}}(\Sigma, X)$ , can be reduced to **TRUE** using the rules of the store  $F = \langle \Sigma, \mathcal{R} \rangle$ .*

The predicate  $eval_{\mathcal{L}}$  (or relation  $\vdash_{\mathcal{L}}$ ) corresponds to an evaluation of a boolean expression in classical functional languages, or to a test for validity in logic programming. In the sequel, whenever the (declarative) language  $\mathcal{L}$  is clear from the context, we omit the corresponding index. Similarly, we write, by abuse of notation,  $S$  instead of  $\overline{S}$ , if there is no risk of confusion.

**Example 2.** *For the functional programming language SML [48], the predicate  $eval_{\text{SML}}$  is defined by the standard operational semantics of SML, applied to terms of type `bool`.*

**Example 3.** *For the logic programming language Prolog [17] (respectively, the functional logic programming languages  $\mathcal{TOY}$  [43] and Curry [36]), the operation  $eval_{\text{Prolog}}$  (respectively,  $eval_{\text{TOY}}$  and  $eval_{\text{Curry}}$ ) is defined by the standard operational semantics (based on resolution respectively, narrowing) with the additional condition that the answer substitution should be the identity substitution. Notice that we model atoms (i.e., applications of predicates) as terms of sort **Truth** and that a Prolog (respectively,  $\mathcal{TOY}$  or Curry) program is a set of clauses (or rules) and corresponds thus obviously to definition 1.*

**Example 4.** *Besides declarative languages, classical imperative programming languages, such as ADA or C, can be used for the description of stores. Intuitively, the rules of an imperative store define the values stored in each of the cells of the memory, and terms are straightforwardly defined as expressions. However, we have to require that expressions use only functions<sup>17</sup> which do not have side-effects, i.e., which do not modify the memory. The check for validity is then the (side-effect free) evaluation of a boolean-valued expression.*

<sup>16</sup>For instance, functional types can be constructed using the type constructor  $\rightarrow$ , e.g., the sort  $s_1 \rightarrow s_2$  denotes the sort of functions of domain  $s_1$  and range  $s_2$ .

<sup>17</sup>Procedures are captured in our model by the notion of processes. In fact, both execute actions.

Notice that the definition of stores is similar to the notion of a *logical system* in the framework of institutions [33]. Informally, a logical system is characterised by a signature  $\Sigma$ , a collection of  $\Sigma$ -sentences, a collection of  $\Sigma$ -models and a  $\Sigma$ -satisfaction relation (of  $\Sigma$ -sentences by  $\Sigma$ -models) [33, page 96].

In the remainder of this paper, we consider (conditional) term rewriting systems [18, 41] as a formalism for the description of stores.

#### 4.1.2 Modeling Mobility: Names

We conclude this section with the description of an additional built-in sort which is necessary in order to model mobile processes in the sense of the  $\pi$ -calculus [47], that is to say by passing the names of communication links. Recall from section 2, that processes of a component modify the store or declarative program by the execution of actions. Furthermore, these actions are defined on a meta-level with respect to the store, since they manipulate stores or declarative programs as data. At the level of the actions, *i.e.*, at the meta-level with respect to the store or declarative program, we distinguish between the value denoted by a constant  $c$  (which is a meta-representation of a *term*) and the constant  $c$  itself (which is a function *symbol*).

This distinction is present in all languages providing assignment. In imperative programming languages, references or pointers allow to distinguish between the pointer or reference and the value which is referenced. Notice that the notion of variable in imperative languages denotes both, the value stored in a given place in the memory (when occurring at the right of  $:=$ ) and the address of the place in the memory where the value is stored (when occurring on the left of  $:=$ ). Similarly, in SML [48] a symbol of type (or sort) `ref t` is distinguished from a symbol of type `t`. The distinction between structural equality (*i.e.*, equality of values) and physical equality or object identity is another reflection of this difference.

As already mentioned in section 2.2, we need, besides actions modifying the rules of a store, also actions for the modification of the *signature*. For the creation of new (function) symbols we introduced in section 2.2 the elementary action `new`. In our setting, this action requires the introduction of a new parameterised sort representing the *names* or references to (function) symbols into the store itself. The following example illustrates this necessity.

**Example 5.** *Consider a process, say  $A$ , which creates a new communication channel, say  $c$ , such that another, concurrently executing process, say  $B$ , should send messages to  $A$  using  $c$ . For simplicity, we suppose that channels are represented as lists of messages. Notice that since the channel  $c$  is freshly created by the process  $A$ , it is in the local scope of  $A$  and the process  $B$  has no knowledge of  $c$ . Since processes in our computation model use the common store for their interaction and communication,  $A$  has to use the store to inform  $B$  about the new channel  $c$ . Thus suppose that  $A$  can send messages to  $B$  using a channel  $d$ . But passing the name of the channel  $c$  to  $B$  is to be distinguished from passing the current value of  $c$  (which is probably an empty list of messages). Therefore, the type of messages that can be passed through the channel  $d$  has to be names of channels.*

Therefore we introduce a new parameterised (built-in) sort to denote the sort of the name of a symbol of sort  $s$ .

**Definition 6 (Name).** *The sort  $\text{Name}(s)$  denotes the sort of symbol-names such that the sort of the symbols is  $s$ . If  $c$  is of sort  $\text{Name}(s)$ , we denote by  $c\uparrow$  the associated symbol of sort  $s$ .*

For the ease of programming, we require that the names of the symbols declared by the programmer are added implicitly. For a signature  $\Sigma = \langle S, \Omega \rangle$ , we call *name-signature*  $\Sigma^n$  the signature of all the names of the symbols of  $\Sigma$ . Thus we require that the signature  $\tilde{\Sigma}$  of a store is the (disjoint) union of the signature  $\Sigma$  as defined by the programmer and the associated name-signature:<sup>18</sup>

$$\begin{aligned}\Sigma^n &\stackrel{\text{def}}{=} \left\langle \left\{ \text{Name}(s) \mid s \in \overline{S} \right\}, \left\{ \hat{f} : \text{Name}(s) \mid f \in \Omega_s \right\} \right\rangle \\ \tilde{\Sigma} &\stackrel{\text{def}}{=} \Sigma \uplus \Sigma^n\end{aligned}$$

Accordingly, the execution of the (elementary) action  $\text{new}(x, s)$  introduces *two* new symbols in the (signature of the) store, namely  $x$  of sort  $\text{Name}(s)$  and  $x\uparrow$  of sort  $s$ .  $x$  stands for the name of (or a reference to) the symbol  $x\uparrow$ .

## 4.2 Component Signatures

A component is defined as a part of a system. Consequently, the description of a component that interacts with the rest of the system necessarily depends on the specification of the system. In our computation model, a system is modeled as a set of components, which are identified by means of storenames. Thus we can represent a system by the set  $\mathcal{SN}$  of all the storenames of the components forming the system, together with a bijective mapping from storenames to components. In the following we therefore define a component with respect to a set of storenames  $\mathcal{SN}$  which represents the system the component is designed for.

In this section we present the notion of a component signature, which introduces all the different symbols occurring in the description of a component. Notice that these symbols belong to all the different levels shown in figure 4, that is to say to the level of the store, to the meta-level of the store, as well as to the part describing the processes. Besides the symbols of the stores (and the associated meta-representations) of the components in the system, a component signature defines symbols for actions and processes. These symbols defined in a component signature allow therefore the construction of processes and actions. We comment on the different parts of a component signature after the definition.

**Definition 7 (component signature).** *Let  $\mathcal{SN}$  be a set of storenames. We define, for a storename  $\hat{s}n \in \mathcal{SN}$  and a declarative language  $\mathcal{L}$ , a component signature  $\mathbb{C}\Sigma$  as a six-tuple  $\mathbb{C}\Sigma = \langle \Sigma, \mathbb{M}\Sigma_{\mathcal{L}}, A, I, E, P \rangle$  where*

- $\Sigma = \langle S, \Omega \rangle$  is a signature of a store, i.e., of a  $\mathcal{L}$ -program,
- $\mathbb{M}\Sigma_{\mathcal{L}} = \langle M_{\mathcal{L}}, MO_{\mathcal{L}} \rangle$  is a meta-signature for  $\mathcal{L}$ , i.e., a pair of a set of meta-sorts  $M_{\mathcal{L}}$  and a ( $\overline{M}_{\mathcal{L}}$ <sup>19</sup>-indexed) family of meta-function symbols  $MO_{\mathcal{L}}$ , such that  $M_{\mathcal{L}}$

<sup>18</sup>The (disjoint) union of two signatures  $\Sigma_1 = \langle S_1, \Omega_1 \rangle$  and  $\Sigma_2 = \langle S_2, \Omega_2 \rangle$  is defined in the obvious way, e.g.,  $\Sigma_1 \uplus \Sigma_2 \stackrel{\text{def}}{=} \langle S_1 \uplus S_2, \Omega_1 \uplus \Omega_2 \rangle$ .

<sup>19</sup>As in definition 1 we note  $\overline{S}$  the set of types that can be constructed from the basic sorts  $S$ .

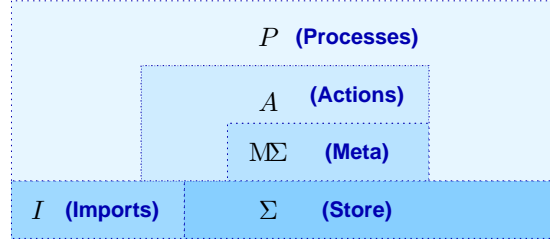


Figure 4: Levels of a System Description: Structure of a Component-Signature

contains all the sorts corresponding to the syntactic entities of the language  $\mathcal{L}$  (in particular the sort  $\mathbf{store}_{\mathcal{L}}$  representing well-formed stores written in  $\mathcal{L}$ ),

- $A$  is a  $(\overline{(S \uplus M_{\mathcal{L}})})^{19}$ -indexed family of action symbols, the sorts of which are of the form  $\mathbf{s}_1 \rightarrow \dots \rightarrow \mathbf{s}_n \rightarrow \mathbf{store}_{\mathcal{L}} \rightarrow \mathbf{store}_{\mathcal{L}}$  ( $n \geq 0$ ),
- $I = \{ \mathbb{I}_{sn} = \langle \Sigma_{sn}, \mathbb{M}\Sigma_{\mathcal{L}^{sn}}, A_{sn} \rangle \mid sn \in (\mathcal{SN} \setminus \{\widehat{sn}\}) \}$  is a  $(\mathcal{SN}$ -indexed) family of imported signatures  $\mathbb{I}_{sn}$ , i.e., triples of signatures  $\Sigma_{sn} = \langle S_{sn}, \Omega_{sn} \rangle$ , meta-signatures  $\mathbb{M}\Sigma_{\mathcal{L}^{sn}} = \langle M_{\mathcal{L}^{sn}}, MO_{\mathcal{L}^{sn}} \rangle$  ( $\mathcal{L}^{sn}$  is the (declarative) language used for the store of component  $sn$ ) and  $(\overline{(S \uplus M_{\mathcal{L}^{sn}})})^{19}$ -indexed families of actions symbols  $A_{sn}$  (the sorts of which are of the form  $\mathbf{s}_1 \rightarrow \dots \rightarrow \mathbf{s}_n \rightarrow \mathbf{store}_{\mathcal{L}^{sn}} \rightarrow \mathbf{store}_{\mathcal{L}^{sn}}$ ),
- $E = \langle E_{\Sigma}, E_{\mathbb{M}\Sigma}, E_A \rangle$  is a triple of a sub-signature<sup>20</sup>  $E_{\Sigma}$ , a sub-meta-signature  $E_{\mathbb{M}\Sigma}$  and a subset of the action symbols  $E_A$ , i.e.,  $E_{\Sigma} \subseteq \Sigma$ ,  $E_{\mathbb{M}\Sigma} \subseteq \mathbb{M}\Sigma$  and  $E_A \subseteq A$ ,
- $P$  is a  $(\overline{PS}^{*19,21})$ -indexed family of process symbols, containing at least the parameterless process  $\mathbf{success} \in P_{\varepsilon}^{21}$ , which always terminates successfully,

and where the set of sorts  $PS$  is defined as

$$PS \stackrel{\text{def}}{=} S \uplus M_{\mathcal{L}} \uplus \left( \bigoplus_{sn \in (\mathcal{SN} \setminus \{\widehat{sn}\})} (S_{sn} \uplus M_{\mathcal{L}^{sn}}) \right) \uplus \{\mathbf{storename}\} \quad (1a)$$

The following paragraphs motivate and comment on the different parts of a component signature.

$\Sigma$ : On the level of the store, a programmer has to specify the signature of the initial stores. Obviously, the component signature has to contain the signature of the store of the component itself, but in order to interact with other components, the component signature needs to incorporate also (parts of) the signatures of the initial stores of these other components (see the imported symbols  $I$ ).

<sup>20</sup>For two signatures  $\Sigma_1 = \langle S_1, \Omega_1 \rangle$  and  $\Sigma_2 = \langle S_2, \Omega_2 \rangle$  we say that  $\Sigma_1$  is a *sub-signature* of  $\Sigma_2$ , written as  $\Sigma_1 \subseteq \Sigma_2$ , if  $S_1 \subseteq S_2$  and  $\Omega_1 \subseteq \Omega_2$ . A similar relation is defined for all other kinds of signatures, in particular meta-signatures, in the obvious way.

<sup>21</sup>For a set of types  $S$ , we denote sequences of types by  $S^*$  and note the empty sequence as  $\varepsilon$ .

$M\Sigma$ ,  $A$ : Along with the signatures of the stores, a component signature contains the definitions of the meta-signatures or ADT’s corresponding to meta-representations of the stores. These symbols are necessary for the definition of actions and their parameters. In this paper, we restrict ourselves to the set of predefined elementary actions presented in section 2.2. User-defined actions are discussed in more detail in [23].

Notice that actions may take, besides meta-terms, also terms of the store as parameters. Thus actions are different from meta-operators ( $MO_{\mathcal{L}}$ ), since the sorts of the latter are constructed only from meta-sorts, whereas the former have sorts constructed from meta-sorts and sorts. The second kind of parameters is to be understood as parameters of the meta-sort corresponding to the syntactic entities of the corresponding sort. In particular, terms are “reified”, *i.e.*, transformed implicitly to meta-terms by means of an implicit application of the mapping *reify*. Roughly speaking, *reify* associates to a syntactic entity  $e$  of a declarative language  $\mathcal{L}$  its representation as a meta-term (of the meta-sort  $\mathbf{e}$  corresponding to this entity  $\mathbf{e} \in M_{\mathcal{L}}$ ). Besides offering a more convenient description of processes, the use of the sorts of the stores in the profiles of action symbols has the additional advantage of enabling static type-checking of the arguments of actions. For instance, considering the assignment action  $c := v$ , we can statically verify that the sorts of the function symbol  $c$  and the term  $v$  are the same. Thus it is no longer necessary to verify this condition at runtime.

$I$ ,  $E$ : The imported signatures allow the processes of a component to interact with other components of the system. Since processes modify stores, we need to import the signatures of the stores, the meta-signatures and the (elementary) actions defined on the remote component: If a process wants to execute an action on the store of another component, it necessarily needs to know the actions executable on the other store, as well as some information about the signature of the store of the other component, in order to construct the parameters of the action in a meaningful way.

The exported symbols of a component are those which other components are allowed to import. So it seems natural to require that these symbols are defined in the component signature and that not necessarily all symbols are exported.

$PS$ : The set of sorts  $PS$  defines the (basic) sorts that can be used in the definition of processes. Besides the (disjoint) union of the sorts and meta-sorts,  $PS$  contains the additional sort `storename`, which represents storenames and allows to pass storenames as parameters to processes.

$P$ : The last kind of symbols introduced by a component signature are *processes*, which are defined by *process definitions*. Their operational semantics is given by means of transition system as described in section 5.

The following convention abbreviates the description of processes. In fact, by considering the definitions of the local store as imported from the component with storename  $\widehat{sn}$ , we obtain a uniform view of all the signatures and meta-signatures.

**Notation 8.** To shorten the notation we integrate the signature  $\Sigma$ , meta-signature  $\mathbb{M}\Sigma$  and family of actions  $A$  of the component into the family of imported symbols  $I$  (which becomes a  $\mathcal{SN}$ -indexed family). We define thus:

$$\Sigma_{\widehat{sn}} \stackrel{\text{def}}{=} \Sigma, \quad \mathbb{M}\Sigma_{\widehat{sn}} \stackrel{\text{def}}{=} \mathbb{M}\Sigma \quad \text{and} \quad A_{\widehat{sn}} \stackrel{\text{def}}{=} A$$

Furthermore, we renew the convention introduced in section 4.1.1 and confound, by abuse of notation, whenever there is no risk of confusion, the set of (basic) sorts  $S$  and the set of sorts  $\overline{S}$  that can be constructed from  $S$ .

We conclude this section with the introduction of the notion of *component terms*, *i.e.*, terms constructed using the symbols defined in a component signature. Component terms are used for the description of the parameters of actions and processes. Informally, a component signature  $\mathbb{C}\Sigma = \langle \Sigma, \mathbb{M}\Sigma_{\mathcal{L}}, A, I, E, P \rangle$  can be seen as a signature, *i.e.*, a pair  $\langle PS, PO \rangle$ , where the set of sorts  $PS$  is defined as in equation (1a) and the ( $\overline{PS}$ -indexed) family of operators combine all the operators defined in a component signature, *i.e.*,

$$PO \stackrel{\text{def}}{=} \left( \biguplus_{sn \in \mathcal{SN}} (\Omega_{sn} \uplus MO_{\mathcal{L}_{sn}}) \right) \uplus \mathcal{SN} \quad (1b)$$

Notice that, according to equation (1b), we consider the storenames as the constructors of the sort `storename`, *i.e.*, we have for all  $sn \in \mathcal{SN}$  that  $sn \in PO_{\text{storename}}$ .

Viewing a component signature as a signature according to equations (1a) and (1b), we define for a ( $\overline{PS}$ -indexed) family of variables  $X$  the ( $\overline{PS}$ -indexed) family of *component terms*  $CT(\mathbb{C}\Sigma, X)$  as the terms over the signature  $\mathbb{C}\Sigma$  and the set of variables  $X$ , *i.e.*,

$$CT(\mathbb{C}\Sigma, X) \stackrel{\text{def}}{=} T(\mathbb{C}\Sigma, X)$$

#### 4.2.1 Example of the Multiple Counters

Consider a (simplistic) application inspired from [34] representing a system of multiple counters. The application starts by creating a window (as shown in figure 5) representing a counter which can be incremented manually by clicking on the button labeled `Increment`. The behaviour of the two other buttons in the counter window is as follows. The `Copy`-button creates an independent counter (with an associated new window) and initialises it with the current value of the counter being copied, whereas the `Link`-button creates a new view (*i.e.*, a new window) of the same counter. All links (or views) of a same counter should behave identically, *e.g.*, they increase the counter at the same time. Additionally we may want to use the current value of the counters for some calculations, in the same way as we would like to use any other constant in a classical declarative language. We use our solution for this problem as a running example for the illustration of the different definitions in the remainder of this section.

We suggest to separate the management of the windows from the counters. Hence we model the system using two components, a first one for the counters, say  $C$ , and a second one for the window system, say  $X$ . This is similar to real window systems, where applications may run on a machine connected via the network to the machine controlling the monitor. In this chapter, we focus on the component  $C$ . We start in this example with the presentation of the component signature for the component  $C$ .

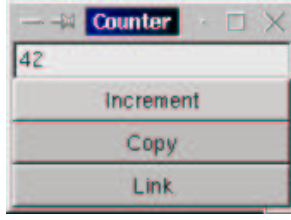


Figure 5: A Counter Window

$$\begin{aligned}
S &\stackrel{\text{def}}{=} \{Cnt; Evt; Wid; Evt\_List; Wid\_List\} \\
C &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} cnt : Nat \times Wid\_List \rightarrow Cnt; \\ increment, copy, link : Evt; \\ nil_{Evt} : Evt\_List; \quad cons_{Evt} : Evt \times Evt\_List \rightarrow Evt\_List \\ nil_{Wid} : Wid\_List; \quad cons_{Wid} : Wid \times Wid\_List \rightarrow Wid\_List \end{array} \right\} \\
D &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} get\_val : Cnt \rightarrow Nat; \quad get\_wins : Cnt \rightarrow Wid\_List \\ head_{Evt} : Evt\_List \rightarrow Evt; \quad tail_{Evt} : Evt\_List \rightarrow Evt\_List \\ head_{Wid} : Wid\_List \rightarrow Wid; \quad tail_{Wid} : Wid\_List \rightarrow Wid\_List \\ append_{Evt} : Evt\_List \times Evt\_List \rightarrow Evt\_List \end{array} \right\}
\end{aligned}$$

Table 2: Signature of the store for the Multiple Counters Example

We specify each of the eight parts of the component signature separately, using the following set of storenames  $SN = \{C; X\}$ .

The store of the component  $C$  describes a theory for counters. We model a *counter*  $c$  as a constant of type  $Cnt$ , *i.e.*, a pair  $\langle val, wins \rangle$  of the current *value*  $val$  of the counter  $c$  and a list of the *window identifiers*  $wins$  of the windows associated with  $c$ , *i.e.*, the windows displaying the value of  $c$ . The fields of a counter can be accessed by the functions  $get\_val$  and  $get\_wins$ . We represent the values of a counter by natural numbers. The window identifiers are represented by strings of characters, which we suppose to be a built-in sort. The processes of the counters have to react on *events* occurring in the windows. The only (high-level) events (occurring in a counter window) we consider are clicks on the different buttons. Thus we define the sort  $Evt$  by the set of the three constructors  $\{increment, copy, link\}$ . Obviously, we also need the sort of lists, which are classically represented by means of two constructors, namely  $cons$  which takes an element and a list and returns a list and  $nil$ , the empty list.

The signature  $\Sigma_C$  of the store of the component  $C$  is thus the enrichment of a signature of natural numbers with the sorts, constructors and functions (we omit the declaration of the equality predicate  $=$  for the new sorts) shown in table 2.

The actions we need for the modifications of the store  $C$  are assignment ( $:=$ ) and the creation of new symbols  $new$ . Recall from section 4.1.2, that  $new$  takes two arguments, the first corresponding to the new (function) symbol to be introduced in the signature

$$A_X \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \text{add-win} : \quad Nat \times \text{Name}(Cnt) \times \text{Name}(Evt\_List) \rightarrow (\text{store} \rightarrow \text{store}); \\ \text{refresh-wins} : Nat \times Wid\_List \quad \quad \quad \rightarrow (\text{store} \rightarrow \text{store}) \end{array} \right\}$$

Table 3: Imports from component the X by the component C

of the sort and the second to the sort of the new symbol. To simplify, we suppose that all these symbols are exported, *i.e.*, the component C does not hide any of its symbols.

Since there are only two components in the system, the family of imported symbols consists only of the imports from the component X which are shown in table 3. In order to display the counter windows, the component C imports two elementary actions from X, namely `add-win` and `refresh-wins`. The action `add-win`( $v, c, e$ ) creates a new counter window for the counter  $c$  displaying the value  $v$  such that clicks on the buttons in this window have the effect of adding a corresponding *Evt* at the end of the list of *Evt*s  $e$ . The process in the component X that actually handles the creation of the window is also in charge of sending an action to the store C which adds the *Wid* of the new window to the counter  $c$ . Thus, the action `add-win` takes the names of the counter  $c$  and the event-list  $e$  as arguments. The action `refresh-wins`( $v, l$ ) refreshes the displays of all the windows denoted by the *Wid*'s in the list  $l$  such that the new value  $v$  is displayed. Since the parameters of the actions of the component X are those imported from C, the imported signature  $\Sigma_X$  is empty.

We suggest to provide one control process per counter window, which we call `cnt_ctrl`. It needs two parameters: the name of the constant representing the counter the value of which is displayed in the window and the name of the list of events which occur in the window. A second process, called `create_win`, handles the creation of new counter windows. `create_win` takes the same arguments as `cnt_ctrl`. Thus, the set of processes of the component C is defined as follows

$$P \stackrel{\text{def}}{=} \{ \text{cnt\_ctrl}, \text{create\_win} : \text{Name}(Cnt) \times \text{Name}(Evt\_List) \rightarrow \text{process} \}$$

### 4.3 Interactions

In our computation model, processes of a same component use the common store for interaction. All processes of a component have access to all information in the store, *i.e.*, the signature and the rules. To communicate, processes modify the store by executing actions. Informally, interaction between components is based on the same scheme: Processes are allowed to modify the stores of other components, *i.e.*, they can execute actions on these stores. For instance, in the example of the multiple counters (see section 4.2.1), the component C needs to refresh the display of the windows associated to a counter whenever the value of the counter has changed. Since the display of the windows is handled by the component X, a corresponding action needs to be sent from C to X. In order to interact, components need to exchange the declarations or profiles of the symbols that are used for the interaction. We say that a component *exports* a declaration, when this declaration can be used by other components. The declarations of a remote component that a component uses for interaction are called *imported* declarations.

We distinguish different levels of imports (respectively, exports). For instance, the declarative program describing a store may itself be a collection of files or modules, and the access to symbols defined in other modules might be restricted. This is to be distinguished from the import (respectively, export) of declarations of a store or declarations of actions from one component to another. The former is a facility to structure the program or store, whereas the latter is necessary for interaction between components. For instance, a component must be able to construct the parameters of an action to be executed on a remote store. In this section, we describe the imports and exports related to the interaction between components.

Since interaction between components in our computation model is based upon the execution of actions on the stores of remote components, a component needs to import the *actions* which can be executed on the stores of other components. Since these actions take parameters that are related to the store of the remote component, the *signature* of the store, *e.g.*, sorts, functions and predicates, as well as its *meta-signature* have to be imported. This led us in the preceding section to define the signature imported from a component with storename  $sn$  as a triple of a signature  $\Sigma_{sn}$ , meta-signature  $M\Sigma_{\mathcal{L}_{sn}}$  (where  $\mathcal{L}_{sn}$  is the (declarative) language used for the store of component  $sn$ ) and a family of actions symbols  $A_{sn}$ .

To avoid name-clashes, *i.e.*, two symbols with the same identifier defined in different components, the identifiers of the imported symbols could be prefixed with the name of the component they are defined in, similar to the prefixing of the module name as in, for example, `ocaml` [42] or Curry [36].

The exported signature of a component is the part of the signature of the component which can be used, *i.e.*, imported, by other components. Thus, a component can export a sub-signature of its store, a sub-meta-signature of its stores and a subset of its action symbols.

**Example 9.** *As an example for imports, reconsider the example of the multiple counters (see section 4.2.1), in particular table 3 which gives signature imported by the component C from the component X.*

*The component C exports its complete signature, meta-signature and set of actions.*

## 4.4 Processes

The processes of a component are specified in the style of a process algebra, see for instance [3, 28]. Basic process terms, *e.g.*, guarded actions or process calls, are combined by means of operators for constructing processes. In this section, we present the definition of processes. We start with the basic process terms and go on with the definition of the operators on processes. Finally we give the rules for defining processes.

### 4.4.1 Guarded Actions

The basic process terms of our computation model are guarded actions. Informally, a guarded action is a pair of a *guard*, *i.e.*, a term of the store, and a sequence of pairs of a storename and a call to an elementary action.

**Definition 10 (guarded action).** *Let  $C\Sigma = \langle \Sigma, M\Sigma_{\mathcal{L}}, A, I, E, P \rangle$  be a component signature (for a declarative language  $\mathcal{L}$  and a storename  $\hat{sn}$ ) and  $X$  a (PS-indexed)*

family of (sets of) variables, where  $PS$  is defined according to equation (1a). We define a guarded action  $\alpha$  as a pair  $[g \Rightarrow \langle sn_1, \mathbf{a}_1(t_{1,1}, \dots, t_{1,l_1}) \rangle; \dots; \langle sn_k, \mathbf{a}_k(t_{k,1}, \dots, t_{k,l_k}) \rangle]$ , consisting of

- a guard  $g$ , i.e., a term of sort  $\mathbf{Truth}$  of the local store, i.e.,  $g \in T_{\mathbf{Truth}}(\Sigma, X)$ , and
- and a sequence of pairs  $\langle sn_i, \mathbf{a}_i(t_{i,1}, \dots, t_{i,l_i}) \rangle$  of a storename and a well-formed call to an elementary action, that is to say, we have  $(\forall i \in \{1; \dots; k\})$  that  $sn_i \in \mathcal{SN}$  is a storename,  $\mathbf{a}_i \in A$  is an elementary action and  $t_{i,j} \in CT(\mathbb{C}\Sigma_i, X)$  is a component term  $(\forall j \in \{1; \dots; l_i\})$ , where the signatures  $\Sigma_{sn}^i = \langle S_{sn}^i, \Omega_{sn}^i \rangle$  of the component signatures  $\mathbb{C}\Sigma_i$  are defined inductively as follows  $(\forall i \in \{1; \dots; k\})$ , and  $\forall sn \in \mathcal{SN}$ )<sup>22, 23</sup>

$$\Sigma_{sn}^1 \stackrel{\text{def}}{=} \Sigma_{sn} = \langle S_{sn}, \Omega_{sn} \rangle$$

$$\Sigma_{sn}^{i+1} \stackrel{\text{def}}{=} \begin{cases} \langle S, \Omega_{sn}^i \uplus \{c : \mathbf{Name}(s); c \uparrow : s\} \rangle & \text{if } sn_i = sn \text{ and} \\ \Sigma_{sn}^i & \mathbf{a}_i(t_{i,1}, \dots, t_{i,l_i}) = \mathbf{new}(c, s) \\ & \text{otherwise} \end{cases}$$

We note the set of guarded actions (for  $\mathbb{C}\Sigma$  and  $X$ ) as  $\mathcal{G}(\mathbb{C}\Sigma, X)$  and use  $\alpha$  to range over guarded actions.

Notice that definition 10 requires the guard of a guarded action to be a term in the store of the *local* component  $\widehat{sn}$ . This implies that while processes are allowed to execute actions on all stores in a system, they can only read, i.e., access information, from the local store (through the guards). The motivation for this restriction is that the check of the validity of the guard and the execution of the action expression are intended as a single (locally) *atomic* operation. By locally atomic we mean that the (sub-) sequences of actions for a given store have to be executed atomically, and that on the local store, the check of the validity of the guard, the evaluation of the action expression to normal form, the execution of the action and the sending of the sequences of actions to the remote stores form a single, atomic operation. If we would allow a guard to contain parts of different stores, than we would have to ensure *global* atomic execution, which is in our opinion not a natural abstraction when the stores (i.e., components) are explicitly distributed<sup>24</sup>.

#### 4.4.2 Process Terms

In this section we define the notion of process terms. Recall from section 2.3, that basic process terms are *guarded actions* (see definition 10) and *process calls*, i.e., applications of a process  $q \in P$ . Process terms can be composed using the operators of process algebras mentioned in section 2.3, namely parallel ( $\parallel$ ) and sequential ( $;$ ) composition, nondeterministic choice ( $+$ ) and choice with priority ( $\oplus$ ).

<sup>22</sup>The other parts of the component signatures  $\mathbb{C}\Sigma_i$  are the same as the corresponding parts of  $\mathbb{C}\Sigma$ .

<sup>23</sup>We will use the component signature  $\mathbb{C}\Sigma_k$  in definition 13.

<sup>24</sup>Global atomic execution of actions would also require the implementation of a “global consensus”, the implementation of which is difficult or even impossible in case of a single faulty process [27].

**Definition 11 (process terms).** Let  $\mathbb{C}\Sigma$  be a component signature and  $X$  an  $(\overline{PS}$ -indexed) family of (sets of) variables, where  $PS$  is defined according to equation (1a). We define the set of process terms  $\mathcal{P}(\mathbb{C}\Sigma, X)$  by the following grammar:

$$p ::= \text{success} \mid \alpha \mid q(t_1, \dots, t_m) \mid p; p \mid p \parallel p \mid p + p \mid p \oplus p$$

In order to get clearer and more readable programs by enforcing a “good programming style”, we introduce the notion of *restricted* process terms. Roughly speaking, a restricted process term is a process term syntactically restricted neither to contain any occurrence of  $\oplus$ , nor a guarded action. Thus, the set of *restricted process terms*  $\mathcal{P}(\mathbb{C}\Sigma, X)$  is defined according to the following grammar:

$$\bar{p} ::= \text{success} \mid q(t_1, \dots, t_m) \mid \bar{p}; \bar{p} \mid \bar{p} \parallel \bar{p} \mid \bar{p} + \bar{p}$$

**Example 12.** Considering the component signature of the example of the multiple counters presented in section 4.2.1, the following process term is not restricted, since it uses a guarded action directly:

$$\left[ \text{TRUE} \Rightarrow \left\langle \mathbb{C}, \text{new}(c, \text{Cnt}) \right\rangle; \left\langle \mathbb{C}, c := \text{zero} \right\rangle; \left\langle \mathbb{C}, \text{new}(e, \text{Evt\_List}) \right\rangle; \left\langle \mathbb{C}, e := \text{nil}_{\text{Evt}} \right\rangle \right]; \text{create\_win}(c, e)$$

#### 4.4.3 Process Definitions

A process is defined by a set (ordered by priority) of rules, clauses or *guarded commands*, each of which consists of a guarded action and a restricted process expression.

**Definition 13 (process definition).** A process definition for the process  $q$  is a phrase of the following form<sup>25</sup>:

$$q(x_1, \dots, x_m) \Leftarrow \bigoplus_{i=1}^n \left( [g^i \Rightarrow \langle sn_1^i, a_1^i(t_{1,1}^i, \dots, t_{1,l_1}^i) \rangle, \dots, \langle sn_{k^i}^i, a_{k^i}^i(t_{k^i,1}^i, \dots, t_{k^i,l_{k^i}^i}^i) \rangle]; \bar{p}_i \right)$$

where (for every  $i \in \{1; \dots; n\}$ , with  $n > 0$ ):

- $[g^i \Rightarrow \langle sn_1^i, a_1^i(t_{1,1}^i, \dots, t_{1,l_1}^i) \rangle, \dots, \langle sn_{k^i}^i, a_{k^i}^i(t_{k^i,1}^i, \dots, t_{k^i,l_{k^i}^i}^i) \rangle] \in \mathcal{G}(\mathbb{C}\Sigma, X)$  is a guarded action for the “local component”,
- $\bar{p}_i \in {}^r\mathcal{P}(\mathbb{C}\Sigma_{k^i}^i, X)$  is a restricted process term, where the component signature  $\mathbb{C}\Sigma_{k^i}^i$  is defined as in definition 10, and
- the free variables in the guarded commands (i.e., the sequential compositions of a guarded action and a restricted process term) are included in the parameters of the process  $q$  (i.e., the set  $\{x_1; \dots; x_m\}$ ).

<sup>25</sup>To ease the reading of the definition, the signification of the different indices is as follows: The process  $q$  has  $m$  parameters and is defined by a set of  $n$  guarded commands. The guarded action of guarded command number  $i$  contains a sequence of  $k^i$  elementary actions, the  $j$ -th of which takes  $l_j^i$  parameters.

$ \begin{aligned} & \text{cnt\_ctrl}(c, e) \Leftarrow \\ & \left[ \begin{array}{l} \text{head}(e\uparrow) = \text{increment} \Rightarrow \langle \mathbf{C}, e := \text{tail}(e\uparrow) \rangle; \\ \langle \mathbf{C}, c := \text{cnt}(\text{succ}(\text{get\_val}(c\uparrow)), \text{get\_wins}(c\uparrow)) \rangle; \\ \langle \mathbf{X}, \text{refresh-wins}(\text{succ}(\text{get\_val}(c\uparrow)), \text{get\_wins}(c\uparrow)) \rangle \end{array} \right]; \\ & \text{cnt\_ctrl}(c, e) \\ & \oplus \left[ \begin{array}{l} \text{head}(e\uparrow) = \text{copy} \Rightarrow \langle \mathbf{C}, e := \text{tail}(e\uparrow) \rangle; \\ \langle \mathbf{C}, \text{new}(e', \text{Cnt}) \rangle; \quad \langle \mathbf{C}, e' := \text{cnt}(\text{get\_val}(c\uparrow), \text{nil}_{\text{wid}}) \rangle; \\ \langle \mathbf{C}, \text{new}(e', \text{Evt\_List}) \rangle; \quad \langle \mathbf{C}, e' := \text{nil}_{\text{Evt}} \rangle \end{array} \right]; \\ & \text{cnt\_ctrl}(c, e) \parallel \text{create\_win}(c', e') \\ & \oplus \left[ \text{head}(e\uparrow) = \text{link} \Rightarrow \langle \mathbf{C}, e := \text{tail}(e\uparrow) \rangle; \langle \mathbf{C}, \text{new}(e', \text{Evt\_List}) \rangle; \langle \mathbf{C}, e' := \text{nil}_{\text{Evt}} \rangle \right]; \\ & \text{cnt\_ctrl}(c, e) \parallel \text{create\_win}(c, e') \\ & \text{create\_win}(c, e) \Leftarrow [\text{TRUE} \Rightarrow \langle \mathbf{X}, \text{add-win}(\text{get\_val}(c\uparrow), c, e) \rangle]; \text{cnt\_ctrl}(c, e) \end{aligned} $
--

Table 4: Process Definitions for the Component C

Intuitively, the operational behaviour of a process call  $q(t_1, \dots, t_m)$  is similar to the alternative construct of the guarded command language of [20]. That is to say, we have to evaluate which of the guards of the commands of the process definition for  $q$  are valid, and then to choose among them the command with the highest priority. Choosing a command means to atomically execute the sequence of elementary actions associated with the guard and afterwards to behave like the associated restricted process term.

**Example 14.** *The process definitions of the store of the component C for the example of the multiple counters (see section 4.2.1) are shown in table 4.*

*The process controlling a counter-window is `cnt_ctrl`, which takes two parameters: the name  $c$  of the associated counter and the name  $e$  of the event-queue to which the window system sends all the events occurring in the window being controlled (an example of a counter window is shown in figure 5). Intuitively, `cnt_ctrl` handles the events in the list  $e$  one by one. For instance, an event corresponding to a click on the **Increment**-button removes the event from the list  $e$ , increments the counter  $c$  (i.e., assigns to  $c$  the pair of the successor of the old value and the old list of windows) and triggers the redrawing of all windows associated to  $c$  (using the process function `refresh-windows`). Then the process continues to execute `cnt_ctrl` (for the same list  $e$  and counter  $c$ ). An event representing a click on the **Copy**-button removes the event from the list  $e$  and creates and initialises a new counter  $c'$  and a new event-list  $e'$ . Then the process continues in parallel with a new process which creates a new window for the new counter. The handling of clicks on the **Link**-button is similar to the handling of **Copy**.*

*The process `create_win` takes the names of a counter  $c$  and an event-list  $e$  as arguments and sends a call to the elementary action `add-win` to the component  $\mathbf{X}$  and subsequently behaves as the process `cnt_ctrl` controlling the new window for the counter  $c$ .*

In analogy to logic programming, where the free variables of a clause or rule have to be *renamed* each time the clause is used, we have to rename the new symbols introduced

by the elementary action `new` in the commands of a process definition. Similar to the renamed rules in logic programming, we call a renamed command of a process definition a *variant*.

**Definition 15 (p-variant).** Let  $\mathbf{c} = (\alpha; \bar{p})$  be a command of a process definition. A p-variant of  $\mathbf{c}$  is defined as the command obtained from  $\mathbf{c}$  by replacing consistently new symbols introduced by the execution of  $\alpha$  by fresh symbols. In the sequel, we denote “rename” the operation which returns a variant for a given command  $\mathbf{c}$ .

The following example illustrates definition 15.

**Example 16.** Consider the process definition of `cnt_ctrl` in the example of the multiple counters as shown in table 4. Every time the `Link`-button in a particular window is clicked, we have to create a new counter window. Therefore, the event-list associated to the new window has to get a fresh name, i.e., we have to rename  $e'$ . A possible renaming of the rule of `cnt_ctrl` handling clicks on `Link` is the following (where  $e'$  has been renamed to  $\tilde{e}$ ):

$$\begin{aligned} & [\text{head}(e\uparrow) = \text{link} \Rightarrow \langle C, e := \text{tail}(e\uparrow) \rangle; \langle C, \text{new}(\tilde{e}, \text{Evt\_List}) \rangle; \langle C, \tilde{e} := \text{nil}_{\text{Evt}} \rangle]; \\ & \text{cnt\_ctrl}(c, e) \parallel \text{create\_win}(c, \tilde{e}) \end{aligned}$$

## 4.5 Components and Systems

We conclude the presentation of our computation model with the complete definition of components and a brief presentation of their composition in order to construct systems.

### 4.5.1 Components

In the preceding sections we have presented how the different symbols occurring in a component signature are defined. All these different definitions together constitute a *component*. As already mentioned in section 4.2, a component is defined as part of a system. Thus the following definition depends on a set of storenames  $\mathcal{SN}$  representing the other parts of the system.

**Definition 17 (component).** Let  $\mathcal{SN}$  be a set of storenames. A component is defined as a five-tuple  $\mathcal{C} \stackrel{\text{def}}{=} \langle \hat{\mathit{sn}}, \mathcal{CS}, \mathcal{R}, \mathcal{R}^P, p^i \rangle$  where

- $\hat{\mathit{sn}} \in \mathcal{SN}$  is the storename (or component-name) of the component,
- $\mathcal{CS} = \langle \Sigma, \mathcal{M}\Sigma_{\mathcal{L}}, A, I, E, P \rangle$  is a component signature with respect to the set of storenames  $\mathcal{SN}$  and the storename  $\hat{\mathit{sn}}$  (see definition 7),
- $\mathcal{R}$  is a set of rules or formulæ such that  $F \stackrel{\text{def}}{=} \langle \Sigma, \mathcal{R} \rangle$  is a store, also called the initial store (see definition 1),
- $\mathcal{R}^P$  is a set of process definitions for the processes  $P$  (see definition 13) and
- $p^i \in {}^r\mathcal{P}^{\mathcal{N}}(\mathcal{CS}, \emptyset)$  is a closed restricted process term, called the initial process term.

$get\_val(cnt(v, ws)) \rightarrow v$		$(R_{get\_val})$
$get\_wins(cnt(v, ws)) \rightarrow ws$		$(R_{get\_wins})$
$head_{Evt}(cons_{Evt}(e, es)) \rightarrow e$	$head_{Wid}(cons_{Wid}(w, ws)) \rightarrow w$	$(R_{head})$
$tail_{Evt}(cons_{Evt}(e, es)) \rightarrow es$	$tail_{Wid}(cons_{Wid}(w, ws)) \rightarrow ws$	$(R_{tail})$

Table 5: Rules for the Store of the Component C

According to definition 17, a component is characterised by its component name or storename, its component signature with the corresponding definitions and its initial process term. The different symbols introduced in a component signature are defined by the store and the definitions of processes. The symbols defined in the imported signatures (in the component signature) are left without definition by the component, since they are imported from other components, which have to provide the necessary definitions. The evolution of the store of the component is defined by the execution of its initial process term on its initial store. We require the initial process term to be closed in order to have a concrete process to execute.

Different kinds of definitions can be distinguished among the definitions of a component, namely those, which are *static*, *i.e.*, which do not change during the execution, and those which are *dynamic*. The storename  $\hat{s}$ , the imported signatures  $I$ , exports  $E$ , as well as the actions  $A$  and processes  $P$  with their related definitions (*i.e.*,  $\mathcal{R}^P$ ) are static. The remaining definitions, *i.e.*, the signature  $\Sigma$  and the set of rules  $\mathcal{R}$ , are called dynamic, since they may change due to the execution of actions. The evolution of the dynamic part is described by the execution of the initial process term on the initial store. We also call the pair of the initial process term and the initial store the *initialisation* of a component.

**Example 18.** Consider the example of the multiple counters which has been presented together with the component signature of the component C in section 4.2.1. Table 5 gives the initial store, *i.e.*, the rewrite rules defining the functions of table 2. The first two rules of table 5 are straightforward definitions of access functions for the sort  $Cnt$ , and the remaining rules are classical definitions for the partial functions  $head$  and  $tail$ . The process definitions are shown in table 4.

It remains thus the specification of the initial process term. Consider the process expression of example 12, which is not in restricted form. In order to use this term as initial process term, we have to wrap it into a new (parameterless) process, say  $start$ . Hence, if we define  $start$  by the process definition

$$start \Leftarrow \left[ \text{TRUE} \Rightarrow \left[ \langle C, \text{new}(c, Cnt) \rangle; \langle C, c := zero \rangle; \langle C, \text{new}(e, Evt\_List) \rangle; \langle C, e := nil_{Evt} \rangle \right] \right]; \text{create\_win}(c, e)$$

we can define the initial process term of the component C as a call to the process  $start$ .

#### 4.5.2 Systems

We model a system simply as a (finite) set of components such that the exported and imported signatures match. Intuitively, a system is constructed by putting together the

components corresponding to the storenames with respect to which the components have been defined.

**Definition 19 (system).** Let  $\mathcal{S}$  be a set of components  $\mathcal{S} \stackrel{\text{def}}{=} \{C_{sn_1}; \dots; C_{sn_n}\}$  and consider the associated set of storenames  $\mathcal{SN}$ , i.e.,  $\mathcal{SN} \stackrel{\text{def}}{=} \{sn_1; \dots; sn_n\}$ .  $\mathcal{S}$  is called a system if for all storenames  $sn \in \mathcal{SN}$ , the component  $C_{sn}$  is defined (with respect to the set of storenames  $\mathcal{SN}$ ) as  $C_{sn} \stackrel{\text{def}}{=} \langle sn, \mathbb{C}\Sigma_{sn}, \mathcal{R}_{sn}, \mathcal{R}_{sn}^p, p_{sn}^i \rangle$  with component signatures  $\mathbb{C}\Sigma_{sn} \stackrel{\text{def}}{=} \langle \Sigma_{sn}, \mathbb{M}\Sigma_{sn}, A_{sn}, I_{sn}, E_{sn}, P_{sn} \rangle$  such that for all pairs of storenames  $sn_1, sn_2 \in \mathcal{SN}$  with  $sn_1 \neq sn_2$

$$(I_{sn_1})_{sn_2} \subseteq E_{sn_2} \quad (2)$$

Notice that a system is *complete* in the sense that all components of the system are specified, since all components are required to be defined with respect to the same set of storenames<sup>26</sup>. Thus, in principle, every component of a system knows about all other components. However, this does not imply that all components have necessarily to interact with each other, but only that such an interaction should be possible. The requirement of matching interfaces is expressed by condition (2) which implies that (for any pair of storenames  $sn_1$  and  $sn_2$ ) the component signature  $\mathbb{C}\Sigma_{sn_1}$  (of the component  $C_{sn_1}$ ) is only allowed to import a subset from the exported symbols  $E_{sn_2}$  of the component signature  $\mathbb{C}\Sigma_{sn_2}$  (of the component  $C_{sn_2}$ ).

**Example 20.** *The complete system for the example of the multiple counters is the composition of two components, namely C which models the control of the counters and X which models the display of the windows. The component C is presented in example 18. We do not specify the component X completely, but consider it as a predefined component (like the actual physical screen used for the display of the windows), the interface of which is presented in section 4.2.1 in form of the imported signature (see table 3). Intuitively, the store of X describes a theory of counter windows, describing all relevant properties of the different windows, as for instance their location on the screen, their name, and the event queue where the messages corresponding to clicks have to be sent to. A process running on the component X ensures that the events occurring on the display lead to the execution of the appropriate actions, i.e., the adding of a message corresponding to the button in the event queue associated to the window.*

## 5 Computation Model: Operational Semantics

The operational semantics of a component has to take into account two different aspects, namely the execution of processes (i.e., the execution of actions causing the modification of the store), and the classical operational semantics of the store, e.g., interactive goal-solving or evaluation of expressions. Thus, we present the operational semantics of a component in three steps. First, we define the execution of (closed) guarded actions. In a second step, we describe the execution of processes by a transition system  $\mathbb{T}_{\mathcal{C}}$ . Finally we combine the rules of  $\mathbb{T}_{\mathcal{C}}$  with rules describing the interactive use of the

---

<sup>26</sup>A straightforward generalisation might be to require that all components are defined with respect to a subset of a common set of storenames.

store, leading to a second transition system  $\mathcal{T}_C$  which defines the operational semantics of a component.

Throughout this section, we consider the operational semantics of a component  $C = \langle \widehat{sn}, \mathbb{C}\Sigma, \mathcal{R}, \mathcal{R}^p, p^i \rangle$  the storename of which is  $\widehat{sn}$ .

## 5.1 Execution of Guarded Actions

To execute a *closed* guarded action, *i.e.*, a guarded action containing no free variables, as for instance  $[g \Rightarrow \langle sn_1, \mathbf{a}_1(t_{1,1}, \dots, t_{1,l_1}) \rangle; \dots; \langle sn_k, \mathbf{a}_k(t_{k,1}, \dots, t_{k,l_k}) \rangle]$  in the store  $F$  (of a component  $C$  with storename  $sn$ ), we have first to test the validity of the guard  $g$  in the store  $F$ . If  $g$  holds in the store  $F$ , *i.e.*,  $F \vdash g$ , we have to execute the sequence  $\langle sn_1, \mathbf{a}_1(t_{1,1}, \dots, t_{1,l_1}) \rangle; \dots; \langle sn_k, \mathbf{a}_k(t_{k,1}, \dots, t_{k,l_k}) \rangle$  of storename/elementary action pairs. For each of these pairs, the storename  $sn_i$  determines the component where the elementary action is to be executed. As an elementary action (when supplied with arguments) corresponds to a function from stores to stores, the effect of executing an elementary action means to replace, *i.e.*, to “destructively update”, the store  $F$  (denoted by  $sn$ ) by the store the result or normal form of  $(\mathbf{a}_i(t_{i,1}, \dots, t_{i,m_i}))(F)$ , *i.e.*, the application of the action to  $F$ .

However, to take into account the difference between local and distributed computations, we distinguish between elementary actions intended for the local component (*i.e.*, associated with the storename  $\widehat{sn}$ ) and all the others. In the first case, we can directly update the local store, in the second we send a message containing the elementary action to the remote component. It is then up to the remote component to ensure that the elementary action is eventually correctly executed. In this section, we are only concerned with the actions to be executed on the local store.

To express the execution of a guarded action formally, we define two functions, namely *exec* and *sel*. These functions take as a parameter a sequence of pairs of a storename  $sn_i$  and an elementary action  $\mathbf{a}_i$ . We represent these sequences as lists. In the sequel, we use the data type of (polymorphic) lists  $Seq(E)$  with elements of sort  $E$ , which is defined by the constructors *nil* (without parameters) for the empty list and *cons* which takes an element  $e$  and a list  $l$  and constructs the list with the element  $e$  in front. The first element of a (non-empty) list  $l$  is denoted by *head*( $l$ ), and the remaining list by *tail*( $l$ ).

The function *sel* takes two arguments, a storename  $sn$  and an action expression  $l \in \mathcal{A}^N(\mathbb{C}\Sigma, X)$  in normal-form, *i.e.*, a sequence of pairs of storenames and elementary actions, which we represent by a list  $l$ , and returns the sub-list of  $l$  consisting of those pairs of storenames and elementary actions of  $l$ , the storename of which is  $sn$ .

$$sel(sn, l) = \begin{cases} nil & \text{if } l = nil, \\ cons(\langle sn', \mathbf{a}(t_1, \dots, t_m) \rangle, sel(sn, tail(l))) & \text{if } sn = sn' \text{ and} \\ & \text{head}(l) = \langle sn', \mathbf{a}(t_1, \dots, t_m) \rangle, \\ sel(sn, tail(l)) & \text{otherwise.} \end{cases}$$

The function  $exec_{sn}$  describes the execution on a store  $F$  (the storename of which

$$\begin{array}{ll}
\text{success}; p \equiv p & (Unit_{\equiv}) \\
\text{success} \parallel p \equiv p & \\
p_1 \parallel p_2 \equiv p_2 \parallel p_1 & (Comm_{\equiv}) \\
p_1 + p_2 \equiv p_2 + p_1 &
\end{array}$$

Table 6: Axiom Schemes Defining the Structural Congruence  $\equiv$  on Process Terms

is  $sn$ ) of a sequence of pairs of the storename  $sn$  and an elementary action.

$$exec_{sn}(l, F) = \begin{cases} F & \text{if } l = nil, \\
exec_{sn}(tail(l), (a(t_1, \dots, t_m))(F)) & \text{if } head(l) = \langle sn, a(t_1, \dots, t_m) \rangle \end{cases}$$

## 5.2 Execution of Process Terms

In this section, we describe the execution of process terms by means of the transition system  $\mathsf{T}_{\mathcal{C}} = \langle \mathsf{Q}, \longrightarrow, \langle F, p^{\ddagger} \rangle \rangle$ . The states of  $\mathsf{T}_{\mathcal{C}}$ , *i.e.*,  $\mathsf{Q}$ , are pairs, *e.g.*,  $\langle F, p \rangle$ , consisting of a store  $F$  and a process term  $p$ . The initial state of  $\mathsf{T}_{\mathcal{C}}$  is built from the initial store  $F$  and the initial process term  $p^{\ddagger}$ , which are both specified by the programmer as parts of the component  $\mathcal{C}$  (see definition 17).

As common in process calculi, we define the transition relation in the style of the Chemical Abstract Machine (CHAM) [4], *i.e.*, modulo a congruence relation, namely  $\equiv$ , on process terms. The congruence relation  $\equiv$  is defined by the axiom schemes shown in table 6; the extension to a congruence relation (reflexivity *etc.*) should be obvious.

Informally, the congruence relation  $\equiv$  states that the process term **success** is a (left) unit element for sequential ( $;$ ) and parallel ( $\parallel$ ) composition (see rules ( $Unit_{\equiv}$ )), and that the operators  $\parallel$  and  $+$  are commutative (see rule ( $Comm_{\equiv}$ )). Notice that **success** is not a neutral element for  $+$ . In fact, **success**  $+$   $p$  has the choice between (immediate) termination or the behaviour of  $p$ , whereas  $p$  cannot, in general, terminate immediately<sup>27</sup>. Obviously, *sequential* composition and choice with *priority* are not commutative by their very nature. Notice finally, that we do not need the axioms for associativity (of  $;$ ,  $\parallel$ ,  $+$  and  $\oplus$ ), since the inference rules shown in table 7 imply that the corresponding processes have the same behaviour.

The inference rules defining the transition relation  $\longrightarrow$  are given in table 7. These rules define exactly the set of correct transitions (with respect to  $\mathsf{T}_{\mathcal{C}}$ ), in the sense that all correct transitions can be inferred by these rules and all transitions that can be inferred by these rules are correct. We comment the rules for  $\longrightarrow$  of table 7 one by one. The combination of the structural congruence and the transition relation is described by rule ( $R_{\equiv}$ ). Intuitively, rule ( $R_{\equiv}$ ) allows to define the transition relation “modulo the congruence  $\equiv$ ”.

Using the auxiliary functions *sel* and *exec* defined in section 5.1, we describe by rule ( $R_{action}$ ) the effects of executing a closed guarded action. Under the premise of the validity of the guard in the current store ( $F \vdash g$ ), the local store is replaced by the application of the action expression in normal form, *i.e.*, a sequence of elementary actions, to the store. Since this sequence may contain elementary actions on several stores, we have to distinguish between elementary actions meant for the local store

<sup>27</sup>Consider the process term  $[TRUE \Rightarrow \text{skip}]$  which executes an action before successful termination.

$$\begin{array}{c}
\frac{p \equiv p' \quad \langle F, p' \rangle \longrightarrow \langle F', p'' \rangle \quad p'' \equiv p'''}{\langle F, p \rangle \longrightarrow \langle F', p''' \rangle} \quad (\mathbf{R}_{\equiv}) \\
\\
\frac{F \vdash g}{\langle F, [g \Rightarrow \langle sn_1, \mathbf{a}_1(t_{1,1}, \dots, t_{1,l_1}) \rangle; \dots; \langle sn_k, \mathbf{a}_k(t_{k,1}, \dots, t_{k,l_k}) \rangle] \rangle \longrightarrow} \\
\langle exec_{\widehat{sn}}(sel(\widehat{sn}, \langle sn_1, \mathbf{a}_1(t_{1,1}, \dots, t_{1,l_1}) \rangle; \dots; \langle sn_k, \mathbf{a}_k(t_{k,1}, \dots, t_{k,l_k}) \rangle), F), success \rangle \quad (\mathbf{R}_{action}) \\
\\
\frac{\langle q(x_1, \dots, x_n) \Leftarrow \bigoplus_{i=1}^m (\alpha_i; p_i) \rangle \in \mathcal{R}^p \quad \langle F, (\bigoplus_{i=1}^m rename(\alpha_i; p_i))[v_j/x_j] \rangle \longrightarrow \langle F', p' \rangle}{\langle F, q(v_1, \dots, v_n) \rangle \longrightarrow \langle F', p' \rangle} \quad (\mathbf{R}_{call}) \\
\\
\frac{\langle F, p_1 \rangle \longrightarrow \langle F', p'_1 \rangle}{\langle F, p_1; p_2 \rangle \longrightarrow \langle F', p'_1; p_2 \rangle} \quad (\mathbf{R}_{;}) \\
\\
\frac{\langle F, p_1 \rangle \longrightarrow \langle F', p'_1 \rangle}{\langle F, p_1 \parallel p_2 \rangle \longrightarrow \langle F', p'_1 \parallel p_2 \rangle} \quad (\mathbf{R}_{\parallel}) \\
\\
\frac{\langle F, p_1 \rangle \longrightarrow \langle F', p'_1 \rangle}{\langle F, p_1 + p_2 \rangle \longrightarrow \langle F', p'_1 \rangle} \quad (\mathbf{R}_{+}) \\
\\
\frac{\langle F, p_1 \rangle \longrightarrow \langle F', p'_1 \rangle}{\langle F, p_1 \oplus p_2 \rangle \longrightarrow \langle F', p'_1 \rangle} \quad (\mathbf{R}_{\oplus}) \\
\\
\frac{\langle F, p_2 \rangle \longrightarrow \langle F', p'_2 \rangle}{\langle F, p_1 \oplus p_2 \rangle \longrightarrow \langle F', p'_2 \rangle} \quad \text{if } \nexists p'_1, \nexists F'', \text{ such that } \langle F, p_1 \rangle \longrightarrow \langle F'', p'_1 \rangle \quad (\mathbf{R}'_{\oplus})
\end{array}$$

Table 7: Inference Rules Defining the Transition Relation  $\longrightarrow$  of  $\mathsf{T}_{\mathcal{C}}$

(recall that we suppose that the storename of the local component is  $\widehat{sn}$ ) and all the others. Notice that rule  $(\mathbf{R}_{action})$  describes only the execution of the “local” actions. The remaining actions are supposed to be sent to the corresponding components which are responsible for their execution.

Notice that the execution of an action is *locally atomic*, *i.e.*, all the elementary actions (for a same store) of an action are executed in a single transition step, so that actions executed by other processes cannot interfere. An example for the usefulness of the atomic execution of actions is the program for the dining philosophers (see section 2.5) where a philosopher needs to take the two necessary chop sticks in a single atomic action.

According to Rule  $(\mathbf{R}_{call})$ , a call to a process corresponds to the execution of an (instantiated) *variant* of the process definition. Recall from definition 15 that, a variant of a command of a process definition is obtained by renaming all new symbols introduced by new elementary actions, and that we note the renaming by the function *rename*. This is similar to the application of clauses in logic programming, where implicitly each variable is renamed by a fresh one, *i.e.*, a new and unused variable. However, in our computation model the programmer has explicitly to specify which symbols should be replaced by fresh ones via the elementary action *new*.

By the way, it can be seen easily, that there exists an  $i \in \{1; \dots; m\}$  such that  $p' \equiv \text{rename}(p_i)[v_j/x_j]$  ( $j \in \{1; \dots; n\}$ ), *i.e.*, the process term  $p'$  is (equivalent to) one of the process terms of the guarded commands defining the process  $q$ . The reason is that the transition in the premise of rule  $(R_{\text{call}})$  is necessarily an execution of a guarded action, due to the syntactical form of process definitions.

Rules  $(R_{;})$  to  $(R_{+})$  describe the classical semantics of the process algebraic operators  $;$ ,  $\parallel$  and  $+$ . According to rule  $(R_{;})$ , executing the process term  $p_1 ; p_2$  means to execute first  $p_1$ . When  $p_1$  has terminated its execution, that is to say when it has become the process term *success*, *i.e.*,  $p'_1 \equiv \text{success}$ , rule  $(R_{\equiv})$  together with the first axiom of  $(Unit_{\equiv})$  ensure that  $p_2$  can start its execution. Rule  $(R_{\parallel})$  specifies an *interleaving* semantics for the parallel composition  $p_1 \parallel p_2$ , *i.e.*, there is only one execution step (or transition) at a time such that the steps executed by concurrent processes are “interleaved” in a nondeterministic way. The execution of the process term  $p_1 + p_2$  as described by rule  $(R_{+})$  consists of the execution of  $p_1$ , discarding  $p_2$ . Since the operator  $+$  is commutative, see the axiom schemes  $(Comm_{\equiv})$ , the execution of  $p_1 + p_2$  can choose nondeterministically between  $p_1$  and  $p_2$ .

Since the operator  $\oplus$  is not commutative, we need two inference rules to define its operational behaviour<sup>28</sup>. When executing the process-term  $p_1 \oplus p_2$ , the process  $p_2$  will only be executed if (in the current store) an execution of  $p_1$  is impossible (see the side-condition of Rule  $(R'_{\oplus})$ ). In contrary,  $p_1$  can be executed independently from the executability of  $p_2$  (see Rule  $(R_{\oplus})$ ).

### 5.3 Combined Operational Semantics of a Component

Besides the execution of the processes modifying the store, described by the transition system  $\mathcal{T}_{\mathcal{C}}$ , the operational semantics of a component  $\mathcal{C}$  has another, orthogonal aspect, namely the classical operational semantics of the declarative programming language  $\mathcal{L}$  used for the description of the store, as for instance goal solving or evaluation of expressions. We suppose that the operational semantics of  $\mathcal{L}$  is described by a relation  $\curvearrowright$ , which we do not precise further. Intuitively,  $\curvearrowright$  describes the transformation steps of a *goal*, *i.e.*,  $\curvearrowright$  is a relation between goals. Examples for  $\curvearrowright$  are for instance rewriting or narrowing.

We describe the operational semantics of a component  $\mathcal{C}$  via a new transition system, namely  $\mathcal{T}_{\mathcal{C}} = \langle \mathcal{Q}, \mapsto, \langle F, p^{\mathbf{i}}, \mathbf{g}^{\mathbf{i}}, \mathbf{g}^{\mathbf{i}} \rangle \rangle$ , where  $\mathbf{g}^{\mathbf{i}}$  denotes a (possibly empty) initial goal the user wants to solve, or the expression which is to be reduced. The states  $\mathcal{Q}$  of the transition system  $\mathcal{T}_{\mathcal{C}}$  are *configurations*, *i.e.*, four-tuples  $\langle F, p, \mathbf{g}^{\mathbf{i}}, \mathbf{g} \rangle$ , where  $F$  is the current store,  $p$  is the current process term,  $\mathbf{g}^{\mathbf{i}}$  is the initial goal to solve and  $\mathbf{g}$  is the expression representing the current state of the evaluation of  $\mathbf{g}^{\mathbf{i}}$  (according to the operational semantics of the declarative language used for  $F$ ). We need the initial goal in a configuration, since we need to know the solving of which goal we should restart if a modification of the store invalidates the evaluation effectuated so far.

Classically, configurations of concurrent languages and process calculi, as *e.g.*, CSP [37], the  $\pi$ -calculus [47] or also our operational semantics for the execution of processes  $\mathcal{T}_{\mathcal{C}}$ , are described only by the first two parts of our configurations, namely  $\langle F, p \rangle$ . As for

---

<sup>28</sup>Notice that we need only *one* rule for the equally non commutative operator “ $;$ ” since in the process term  $p_1 ; p_2$  only  $p_1$  can be executed.

$$\frac{\langle F, \mathbf{g} \rangle \curvearrowright \langle F, \mathbf{g}' \rangle}{\langle F, p, \mathbf{g}^i, \mathbf{g} \rangle \mapsto \langle F, p, \mathbf{g}^i, \mathbf{g}' \rangle} \quad (\text{G})$$

$$\frac{\langle F, p \rangle \longrightarrow \langle F', p' \rangle}{\langle F, p, \mathbf{g}^i, \mathbf{g} \rangle \mapsto \langle F', p', \mathbf{g}^i, \mathbf{g}' \rangle} \quad \text{where } \mathbf{g}' \stackrel{\text{def}}{=} \begin{cases} \mathbf{g} & \text{if } F = F' \\ \mathbf{g}^i & \text{otherwise} \end{cases} \quad (\text{P})$$

Table 8: Inference Rules for the transition system  $\mathcal{T}$

declarative languages, a configuration classically uses only the first and the fourth parts  $\langle F, \mathbf{g} \rangle$  which allow to express the rules of the operational semantics  $\curvearrowright$  of the declarative language. Combining these two operational semantics adds the possibility to execute concurrent processes without loosing the characteristics of declarative languages. For instance, goals can be solved while processes are running. This feature is useful to allow to query, at any moment, the current store or state of an evolving system – without being limited to a fixed, predefined set of possible queries that has been established during the specification of the system. In particular, in a programming framework for rapid prototyping, where the prototype is a means to explore the set of interesting queries, it is interesting to dispose of a rich query language.

The transition relation of  $\mathcal{T}_C$ , *i.e.*,  $\mapsto$ , is defined by the two inference rules shown in table 8. Rule (G) concerns interactive goal-solving, *i.e.*, the use of the operational semantics of the declarative language, as for instance goal solving (in logic languages) or evaluation of expressions (in functional languages). In the example of the Dining Philosophers, rule (G) allows us to ask for the currently eating philosophers by solving the goal `is_eating(x)` (see section 2.5). Rule (P) describes the modifications of the store by the processes via the transition system  $\mathcal{T}$ . When a process modifies the store, we have to restart the goal-solving at the initial goal, *i.e.*,  $\mathbf{g}^i$ , as the modification may invalidate the already achieved derivation.

Obviously, rule (P) is only one of the many possibilities. A rather simple refinement would be to restart the goal-solving only if the execution of a process has altered some of the definitions used so far in the evaluation of the goal. Another, completely different option for rule (P) would be to solve the goal in an unmodified “private copy” of the store. More sophisticated techniques, namely rearranging of the search tree, *i.e.*, the order of the application of rules, have been investigated in [24]. These methods allow to reuse as much as possible of the search tree after the theory has been modified.

## 5.4 Operational Semantics of a System

In this section we present the outlines of the operational semantics of a system of several components. We do not want to model the semantics of such a system by a *single* transition system, since we find it unrealistic to assume that we might have total knowledge about the states of *all* the components of a distributed system at the same time. Indeed, if we were up to defining the states of the global transition system, we would be forced to the simplifying assumption that there exists a point during the execution of the system, such that all components are in a fixed state.

Therefore we do not believe that a single transition system as in the preceding sections is an appropriate model for a distributed system, since it gives a centralised view of a system. An example of such a semantics is the operational semantics of KLAIM [49], where a global transition system is used in order to specify the synchronisation between the different components (or nets) in a system.

We prefer to model a distributed system as a collection of concurrently executing transition systems. These transition systems interact by exchange of messages, where each message corresponds to a sequence of elementary actions which are to be executed atomically on the receiving component. To implement these ideas, it is sufficient to add to every component a *mailbox* which is used for the reception of messages from other components, and to set up a particular process which executes the actions arriving in the mailbox interleaved with the actions executed by the processes of the component.

## 6 Conclusion

In this paper, we have presented a new computational model for concurrent declarative programming, which clearly separates the concepts of processes and those underlying declarative programming languages, *e.g.*, functions and predicates. Thus our model does not force programmers to encode some concepts by means of others, but rather allows to express every part of a system by the most appropriate concept.

Following a component-based approach, we have defined the notion of a component. Informally, a component consists of a store, processes and actions. The store of a component is a declarative program and serves as a shared communication medium for the processes, which modify the store by the execution of actions. Actions are defined on a meta-level with respect to the store. Thus our model can be extended easily to allow programmers to define additional actions (using the metalanguage) [23, 59]. Another extension towards a more abstract description of processes consists in considering functional expressions on actions and processes [59].

In this paper, we have only presented the composition of a system from a set of components. Currently we are investigating the combination of components yielding itself a component.

Since our model is generic with respect to the declarative language being used for the description of the stores, it seems interesting to consider systems the components of which use different languages for the description of their stores. Notice this kind of “multi-language programming” requires, at least, the introduction of the concept of *translation* in order to enable communication across language borders [59].

The introduction of time is another issue to be investigated further, since time is mandatory for modeling most control systems. In such systems, some action has to be taken if a certain condition holds over a given time-interval. It is thus interesting to consider the integration of timed declarative programming as in [6] in our model.

Last, but not least, we have implemented a prototype of our computation model, which we have tested by means of several case-studies, including the Dining Philosophers (see section 2.5) and the multiple counters (see section 4.2.1), but also lift controllers and timetable generation for a hospital.

## References

- [1] J.-R. Abrial. *Formal Methods for Industrial Applications*, volume 1165 of *Lecture Notes in Computer Science*, chapter Steam-Boiler Control Specification Problem, pages 500 – 510. Springer Verlag, 1996.
- [2] J. Armstrong, R. Viriding, C. Wikstrom, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, second edition, 1996.
- [3] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
- [4] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217 – 248, 1992. selected papers of the 2<sup>nd</sup> Workshop on Concurrency and Compositionality, San Miniato, March 1990.
- [5] E. Best, F. de Boer, and C. Palamidessi. Partial order and SOS semantics for linear constraint programs. In D. Garlan and D. L. Métyer, editors, *Proceedings of the 2<sup>nd</sup> International Conference on Coordination: Languages and Models (Coordination '97)*, volume 1282 of *Lecture Notes in Computer Science*, pages 256 – 273, Berlin, Sept. 1997. Springer Verlag.
- [6] J. Blanc and R. Echahed. Adding time to functional logic programs. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 31 – 44, Kiel, Sept. 2001. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel. Bericht Nr. 2017.
- [7] G. Boudol. The  $\pi$ -calculus in direct style. In *Proceedings of the 24<sup>th</sup> ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 228 – 241, 1997.
- [8] M. Broy and M. Wirsing. Algebraic state machines. In T. Rus, editor, *Proceedings of the 8<sup>th</sup> International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 89 – 118, Iowa, May 2000. Springer Verlag. Invited talk.
- [9] L. Caires. *A Model for Declarative Programming and Specification with Concurrency and Mobility*. PhD thesis, Universidade Nova de Lisboa, July 1999.
- [10] L. Caires and L. Monteiro. Verifiable and executable logic specifications of concurrent objects in  $\mathcal{L}_\pi$ . In C. Hankin, editor, *Proceedings of the 7<sup>th</sup> European Symposium on Programming (ESOP '98)*, volume 1381 of *Lecture Notes in Computer Science*, pages 42 – 56, Lisbon, March – April 1998. Springer Verlag.
- [11] M. Carro and M. Hermenegildo. Concurrency in prolog using threads and a shared database. In *Proceedings of the 16<sup>th</sup> International Conference on Logic Programming (ICLP '99)*, Las Cruces, Nov. 1999. The MIT Press.
- [12] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251 – 284, 1994.

- [13] P. Codognet and F. Rossi. Nmcc programming: Constraint enforcement and retraction in cc programming. In *Proceedings of ICLP '95*. The MIT Press, 1995.
- [14] M. Davis, editor. *The Undecidable: Basic Papers On Undecidable Propositions, Unsolvability Problems And Uncomputable Functions*. Raven Press Books, Hewlett, New York, 1965.
- [15] F. S. de Boer and M. Gabbriellini. Infinite computations in concurrent constraint programming. In *Proceedings of the 13<sup>th</sup> Conference on Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, Pittsburgh, 1997.
- [16] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Non-monotonic concurrent constraint programming. In *Proceedings of the International Symposium on Logic Programming (ILPS '93)*, pages 315 – 334. The MIT Press, 1993.
- [17] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual*. Springer Verlag, 1996.
- [18] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 6, pages 243 – 320. Elsevier, Amsterdam, 1990.
- [19] E. W. Dijkstra. Hierarchical ordering of sequential processes. In C. A. R. Hoare and R. H. Perrott, editors, *Proceedings of a Seminar on Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 72 – 93, Belfast, 1971. Academic Press.
- [20] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453 – 457, 1975.
- [21] R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Proceedings of the 1<sup>st</sup> International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 300 – 314, London, July 2000. Springer Verlag.
- [22] R. Echahed and W. Serwe. A component-based approach to concurrent declarative programming. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 285 – 298, Kiel, Sept. 2001. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel. Bericht Nr. 2017.
- [23] R. Echahed and W. Serwe. Integrating action definitions into concurrent declarative programming. *Electronic Notes in Theoretical Computer Science*, 64:? – ?, ? 2002. special issue: selected papers of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001).

- [24] F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1 – 3):185 – 212, Oct. 1998.
- [25] F. Fages, P. Ruet, and S. Soliman. Phase semantics and verification of concurrent constraint programs. In *Proceedings of the 13<sup>th</sup> Annual IEEE Symposium on Logic in Computer Science (LICS '98)*, 1998.
- [26] S. Finne and S. L. Peyton Jones. Composing Haggis. In R. C. Veltkamp and E. H. Blake, editors, *Proceedings of the Eurographics Workshop on Programming Paradigms in Graphics*, pages 85 – 101, Maastricht, Sept. 1995. Springer Verlag.
- [27] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374 – 382, Apr. 1985.
- [28] W. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer Verlag, 2000.
- [29] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. *The JoCaml language (beta release): Documentation and User's Manual*. Institut National de Recherche en Informatique et en Automatique (INRIA), Jan. 2001. available at <http://pauillac.inria.fr/jocaml/htmlman/index.html>.
- [30] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23<sup>rd</sup> ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 372 – 385, St. Petersburg Beach, Florida, Jan. 1996.
- [31] E. R. Gansner and J. H. Reppy. A multithreaded higher-order user interface toolkit. In *User Interface Software*, volume 1 of *Software Trends*, pages 61 – 80. John Wiley & Sons, 1993.
- [32] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80 – 112, Jan. 1985.
- [33] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95 – 146, Jan. 1992.
- [34] GUI Fest '95 Post-Challenge: multiple counters. available at <http://www.cs.chalmers.se/~magnus/GuiFest-95/>, July 24 – July 28 1995. organized by Simon Peyton Jones and Phil Gray as a part of the Glasgow Research Festival.
- [35] M. Hanus. Distributed programming in a multi-paradigm declarative language. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP '99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 188 – 205, Paris, 1999. Springer Verlag.
- [36] M. Hanus [Ed.], S. Antoy, H. Kuchen, F. J. LópezFraguas, W. Lux, J. J. Moreno Navarro, and F. Steiner. Curry: An integrated functional logic language.

available at  
<http://www.informatik.uni-kiel.de/~mh/curry/report.html>,  
June 6, 2000. Version 0.7.1.

- [37] C. A. R. Hoare. *Communicating sequential processes*. International Series in Computer Science. Prentice Hall, 1985.
- [38] ISO JTC 1/SC 7. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, June 2000. ISO 8807:1989.
- [39] S. Janson, J. Montelius, and S. Haridi. Ports for objects in concurrent logic programs. In G. A. Agha, P. Wegner, and Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.
- [40] M. P. Jones and P. Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Yale University, Aug. 1993.
- [41] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1 – 112. Oxford University Press, 1992.
- [42] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system: Documentation and user’s manual*. Institut National de Recherche en Informatique et en Automatique (INRIA), July 2001. Release 3.02.
- [43] F. J. López-Fraguas and J. Sánchez-Hernández.  $\mathcal{TOY}$ : A multiparadigm declarative system. In *Proceedings of the 10<sup>th</sup> International Conference on Rewriting Techniques and Applications (RTA ’99)*, pages 244 – 247. Springer Verlag, LNCS 1631, 1999.
- [44] R. Milner. *A calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.
- [45] R. Milner, editor. *Communication and Concurrency*. Prentice Hall International, 1989.
- [46] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):78 – 89, Jan. 1993. Turing Award Lecture.
- [47] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.
- [48] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML – Revised*. The MIT Press, 1997.
- [49] R. D. Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315 – 330, May 1998.
- [50] S. L. Peyton Jones, A. D. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23<sup>rd</sup> ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL ’96)*, pages 295 – 308, St Petersburg Beach, Florida, Jan. 1996.

- [51] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.
- [52] R. Pike. A concurrent window system. *Computing Systems*, 2(2):133 – 153, Spring 1989.
- [53] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, SIGPLAN Notices, pages 293 – 305, Toronto, June 1991. ACM Press.
- [54] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
- [55] G. A. Ringwood. Parlog86 and the dining logicians. *Communications of the ACM*, 31(1):10 – 25, Jan. 1988.
- [56] V. A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. The MIT Press, 1993.
- [57] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5 – 6):475 – 520, November – December 1996.
- [58] V. A. Saraswat and P. Lincoln. Higher-order, linear, concurrent constraint programming. Technical report, Xerox PARC, 1992.
- [59] W. Serwe. *On Concurrent Functional-Logic Programming*. thèse de doctorat, Institut National Polytechnique de Grenoble, Mar. 2002.
- [60] E. Shapiro. Technical correspondence: Linda in context. *Communications of the ACM*, 32(10):1244 – 1249, Oct. 1989.
- [61] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324 – 343. Springer Verlag, 1995.
- [62] B. Thomsen, L. Leth, and T.-M. Kuo. FACILE — from toy to tool. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation and Application*, Monographs in Computer Science, chapter 5, pages 97 – 144. Springer Verlag, 1996.
- [63] A. M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, ser. 2, vol. 45:161 – 228, 1939. reprinted in [14, pages 154 – 222].
- [64] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315 – 351, Feb. 1998.

**Le laboratoire Leibniz est fortement pluridisciplinaire. Son activité scientifique couvre un large domaine qui comprend aussi bien des thèmes fondamentaux que des thèmes très liés aux applications, aussi bien en mathématiques qu'en informatique.**

**Les recherches sur les Environnements Informatiques d'Apprentissage Humain et la didactique des mathématiques ouvrent cette pluridisciplinarité sur les sciences humaines, elles jouent un rôle particulier en favorisant les coopérations entre différentes composantes du laboratoire.**

- \* mathématiques discrètes et recherche opérationnelle
- \* logique et mathématique pour l'informatique
- \* informatique de la connaissance
- \* EIAH et didactique des mathématiques

*Les cahiers du laboratoire Leibniz* ont pour vocation la diffusion des rapports de recherche, des séminaires ou des projets de publication réalisés par des membres du laboratoire. Au-delà, Les cahiers peuvent accueillir des textes de chercheurs qui ne sont pas membres du laboratoire Leibniz mais qui travaillent sur des thèmes proches et ne disposent pas de tels supports de publication. Dans ce dernier cas, les textes proposés sont l'objet d'une évaluation par deux membres du Comité de Rédaction.

### **Comité de rédaction**

- \* mathématiques discrètes et recherche opérationnelle  
Gerd Finke, Andrés Sebõ
- \* logique et mathématique pour l'informatique  
Ricardo Caferra, Rachid Echahed
- \* informatique de la connaissance  
Yves Demazeau, Daniel Memmi,
- \* EIAH et didactique des mathématiques  
Nicolas Balacheff, Jean-Luc Dorier, Denise Grenier

Contact Gestion & Réalisation : Jacky Coutin  
Directeur de la publication : Nicolas Balacheff  
ISSN : 1298-020X - © laboratoire Leibniz