appeared in the Proc. of the First International Conference on Computational Logic (CL2000)

Combining Mobile Processes and Declarative Programming

Rachid Echahed and 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

Abstract. We propose a general framework for combining mobile processes and declarative programming languages, e.g., functional, logic or functional-logic languages. In contrast to existing concurrent extensions of declarative languages, we distinguish clearly between the notion of processes and that of functions or predicates. Thus, our framework 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 and may be modified thanks to the actions performed by processes.

1 Introduction

Classical declarative languages, i.e., functional, logic and functional-logic languages, aim to provide high-level descriptions of applications or systems. These languages have well-known nice features (e.g., abstraction, readability, compilation techniques, proof methods etc.) since functions and predicates are well mastered mathematical concepts which have been successfully used in describing algorithms even before the invention of computers. However, these concepts, which constitute the basis of classical functional-logic languages, are not sufficient to capture the whole complexity of real-world applications [30] where interactivity, concurrency and distributivity are needed.

On the other hand, processes have been used as a means for the description of interactive applications. Informally, a process can be described as the set of its possible runs, i.e., the set of possible action-sequences which can be performed by the process. Thus, it is immediate that processes are different from functions or predicates.

Most existing concurrent extensions of declarative languages do not distinguish clearly between processes and the concepts underlying the declarative language, but rather try to *encode* processes in terms of the latter [4, 5, 8–11, 13, 16, 17, 21, 23–28]. Thus each of these approaches seems to be tailored for a specific language rendering the extension to a general framework not straightforward.

Classically, declarative languages allow one to describe functions by means of equations (rewrite rules) or lambda abstractions and predicates by Horn clauses (with constraints). As for processes, there are as many ways to define them as there are different programming styles (including temporal logic programming). Nevertheless, process algebras have been well investigated, see e.g., [3], and provide a clean framework for the description of concurrent processes. An extension, that allows the communication structure to vary, is the π-calculus [20]. It provides a basis for modelling mobile computations.

But similar to concurrent extensions of declarative languages, programming languages uniquely based on process calculi *encode* the notions of functions and predicates via processes, e.g., [22].

This paper aims at a new combination of mobile processes and declarative languages where we distinguish clearly 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 framework 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. Our resulting proposal extends both, declarative programming with mobile processes, and PA process algebra in order to deal with parameter passing, mobile processes and interactive declarative programming.

Theoretically, our framework 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 and mobile 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: The next section gives a broad outline of our framework. Section 3 discusses the definitions of actions. The definition of processes is given in Sect. 4 and some examples are presented in Sects. 5 and 6. In Sect. 7 we give the operational semantics. A comparison with some related work is subject of Sect. 8. Finally, Sect. 9 concludes.

2 Overview of our Proposal

In our framework, an application is modelled via several components. Due to space limitations, we focus in this paper on applications with a single component; but the framework presented here can be extended easily to several components. Roughly speaking, a program or a component in our framework will consist of two parts $C = (\mathcal{F}, \mathcal{P})$. $\mathcal{P}$ is a set of process definitions and $\mathcal{F}$ is a set of formulæ describing a traditional declarative program, called a *store* in the sequel. The execution model of a component can be schematised as in Fig. 1. Processes (p_i) communicate by modifying the common store $\mathcal{F}$, i.e., by altering, in a non-monotonic way, the current theory described by the store, for example by simply redefining constants (e.g., adding a message in a queue) or by adding or deleting formulæ in $\mathcal{F}$. Hence, the execution of processes will cause the transformation of the store $\mathcal{F}$. Every change of the store is the result of the execution of an *action*. Thus actions constitute the basic entities for building processes. In general, processes are also able to modify the stores of other components and thus interact with the environment of their component. Orthogonally, the store can be used as usual for a functional-logic program, i.e., for goal solving or the evaluation of expressions.

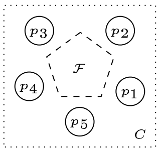

Fig. 1. execution model

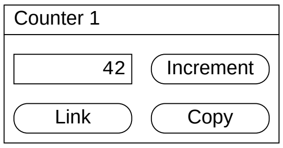

Fig. 2. a counter window

A real-world instance of a system as depicted in Fig. 1 is for example the file-system on a UNIX-workstation: while there are processes running which may modify the store, i.e., the file-system, it is always possible to investigate the current state by commands as `ls` or `find`. Following the same idea, there are some processes keeping the list of currently logged users up-to-date; this list can be consulted via the command `who`. This investigation may be done "on-line", that is to say, the set of possible questions a user might ask is not known in advance; in fact, the user might ask *any* well-formed question.

Another example is an application emitting flight tickets. Obviously, declarative languages are well-suited to describe the function calculating the price of a ticket. However, if we want to allow the modification of this function without stopping the entire application, then we need to extend classical declarative languages.

To illustrate our extension further, we consider a (simplistic) application inspired from [15], a system of multiple counters. The application starts by creating a window (as shown in Fig. 2) representing a counter which can be incremented manually. In the window of the counter, a copy-button and a link-button allow one to create new counter windows: the former creates an independent counter (with an associated new window) and initialises it with the current value of the counter being copied; the latter 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. A sample of our solution for this problem is subject of Sect. 5.

This example illustrates some of the difficulties when modelling concurrent processes, such as dynamic creation of new constants (e.g., counters, windows and the corresponding channels) or resources shared by several processes. Obviously, we need to extend a pure declarative language to cope with this *interactive* application. This has also been noted in the literature on declarative programming: “Some interactions appear most straightforward to express in an imperative style, and we should not hesitate to do so” [29].

3 Actions

In our framework, an application may be composed of several components, each consisting of a store and a set of processes. These processes execute actions modifying the stores of different components. Classical examples of actions are for instance tell of ccp (see Sect. 8) which adds a constraint to a constraint store, or `setq` of Scheme which updates destructively the value of a constant. Other examples of actions concern interaction with the environment, as for instance the commands controlling external physical machines, e.g., opening a door or starting an engine.

Actions are composed of a guard and a sequence of elementary actions. Informally, an elementary action is a function(al), that when supplied with its arguments, returns a total recursive function from stores to stores.

Definition 1. *An* action α *is a pair consisting of a guard g and a sequence of pairs of a storename (expression) s_i and an (parameterised) elementary action a_i, written: $[g \Rightarrow \langle s_1, \mathsf{a}_1\rangle; \ldots ; \langle s_n, \mathsf{a}_n\rangle]$. A* guard *is a formula whose validity (in the (current) store) is decidable. An* elementary action a *is a total recursive mapping whose type is of the following shape: types_of_arguments $\rightarrow$ (store $\rightarrow$ store).*

Roughly speaking, executing an action means to test the validity of the guard in the (local) store, and, upon the validity of the guard, to execute (atomically) the sequence of the elementary actions, $\langle s_i, \mathsf{a}_i\rangle$, that is to say replace the store denoted by the storename (expression) s, say $\mathcal{F}$, by the result of the application of the elementary action a to it, i.e., $\mathsf{a}(\mathcal{F})$. Informally, a storename can be seen as a symbolic identifier for a store in the same way as the symbolic host-names stand for (numeric) IP-addresses. In the sequel, we will note a sequence $\gamma_1; \ldots ; \gamma_n$ as γ_i. Thus, an action $[g \Rightarrow \langle s_1, \mathsf{a}_1\rangle; \ldots ; \langle s_n, \mathsf{a}_n\rangle]$ will be represented by $\left[g \Rightarrow \left\langle s_i, \mathsf{a}_i\right\rangle_i\right]$.

Due to space limitations, we suppose in this paper that the set of elementary actions is given, and thus restrain ourselves to give below just some examples of useful elementary actions. In our full framework [12], actions are definable by the programmer.

We call skip the elementary action which does not modify the store. Certainly the most common elementary action is assignment (:=). := takes two parameters: the name c of a constant (traditionally considered as a variable) and a (new) value v. Thus we introduce a new parameterised type $\texttt{Name}(t)$ to denote the type of the name of a symbol of type t. If c is of type $\texttt{Name}(t)$, we denote by $c\!\uparrow$ the associated symbol of type t. Hence, $:=(c, v)(\mathcal{F})$ represents the store obtained from $\mathcal{F}$ by erasing all equations of the form $c\!\uparrow == term$ and adding the new equation $c\!\uparrow == v$. In the sequel, and by abuse of notation, we write c in place of $c\!\uparrow$ whenever there is no ambiguity.

The elementary action tell (respectively, del) adds (respectively, removes) clauses to (respectively, from) the store.

As all processes of a component share the store, broadcasting messages between processes is realised naturally via the common store by executing appropriate elementary actions. To simulate message passing, we can use elementary actions handling queues representing the buffers for (incoming) messages. To send a message, e.g., m, to a process, we need to know the name, e.g., q, of its queue for incoming messages (q is a constant of type $\mathtt{Name}(\mathit{List}(\mathit{messages}))$). Executing the elementary action $\mathsf{enq}(q, m)$ will put our message in the queue of our (communication-) partner. To read a message from a queue, we access the queue, i.e., q, just as we usually access a constant. In order to erase a message from the queue (after having read it), we introduce the elementary action $\mathsf{deq}(q)$.

Another important elementary action handles the creation of new symbols: $\mathsf{new}(s, t)$ introduces two new symbols in the store, namely s of type $\mathtt{Name}(t)$ and $s\!\uparrow$ of type t. s stands for the name of (or a reference to) the symbol $s\!\uparrow$. In the multiple counters example, new allows the creation of new counters and the associated communication channels with the window-system.

Example 1. The following is an example of an action, inspired from the program of the "Dining Philosophers" given in Sect. 6:

$$\left[\begin{array}{lcl} \text{stick}(x) \qquad \wedge & & \langle F,\ \mathsf{del}(\text{stick}(x))\rangle; \\ \text{stick}(x{+}1 \bmod n) & \Rightarrow & \langle F,\ \mathsf{del}(\text{stick}(x{+}1 \bmod n))\rangle; \\ & & \langle F,\ \mathsf{tell}(\text{is\_eating}(x))\rangle \end{array}\right]$$

This action says that whenever the sticks number x and x+1 mod n are available on the table (i.e., $\big(\text{stick}(x) \wedge \text{stick}(x{+}1 \bmod n)\big)$ holds in the store F), then the philosopher number x can get these sticks (i.e., remove them from the table by deleting the corresponding predicates via the elementary actions $\mathsf{del}(\text{stick}(x))$ and $\mathsf{del}(\text{stick}(x+1 \bmod n))$ from the store F) and start eating (i.e., add the predicate is_eating(x) to F by using the elementary action $\mathsf{tell}(\text{is\_eating}(x))$).

4 Processes

Before giving the formulæ defining processes, we introduce first the notion of process-terms. Such terms are constructed by means of basic processes as well as operators which combine processes into new ones.

Basic processes are $\mathsf{success}$, i.e., the process which terminates successfully, (guarded) actions α, or process calls $\mathrm{p}(t_1, \ldots, t_n)$. As usual in process algebra (see, e.g., [3]), we provide some operators for combining processes: parallel ($\|$) and sequential (;) composition, nondeterministic choice (+) and choice with priority ($\oplus$). The last operator 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 $p_1 \oplus p_2$ is: "execute the process p_2 only if the process p_1 cannot be executed", i.e., the process p_1 has a higher priority than the process p_2.

Definition 2. *A* process term *p is a well-typed expression defined by the following grammar:*

$$p ::= \mathsf{success} \mid \left[g \Rightarrow \left\langle s_i,\, \mathsf{a}_i \right\rangle_i\right] \mid \mathrm{p}(t_1,\ldots,t_n) \mid p;p \mid p \,\|\, p \mid p+p \mid p \oplus p$$

Process abstractions are intended to give a description of the behaviour of processes. Some restrictions are required on the (recursive) definitions of process abstractions in order to avoid pathologic cases, especially processes with an infinite branching degree. A common solution to avoid such problems consists in requiring process abstractions to be *guarded.*

Definition 3. *A* process abstraction p *is defined by a sentence of the form*

$$\mathrm{p}(x_1,\ldots,x_n) \Leftarrow \bigoplus_{i=1}^{m} \alpha_i;\; p_i$$

where (for each i) α_i is an action and p_i is a process term. For a readable presentation, we omit here some formal technical conditions on the use of variables.

According to definition 3 a process abstraction is defined by a set (ordered by priority) of "formulæ", which consist of a (guarded) action and a process term.

Example 2. inc-dec is a process on a store C receiving via a queue named q messages for incrementation (*inc*) or decrementation (*dec*) of the value of a counter named *val*.

$$\begin{aligned} \text{inc-dec} \Leftarrow\ & [\mathit{head}(q) == \mathit{inc} \Rightarrow \langle C,\ \mathit{val}{:=}\mathit{val}{+}1\rangle;\ \langle C,\ \mathit{deq}(q)\rangle];\ \text{inc-dec} \\ \oplus\ & [\mathit{head}(q) == \mathit{dec} \Rightarrow \langle C,\ \mathit{val}{:=}\mathit{val}{-}1\rangle;\ \langle C,\ \mathit{deq}(q)\rangle];\ \text{inc-dec} \end{aligned}$$

The elementary action new presented in the previous section allows to extend the signature of the store with new symbols for functions or predicates. As these symbols are created dynamically and can be passed, we can model mobile processes in the same way as in the π-calculus [20], namely by passing the *names* of communication channels.

As an example, consider a process, p_1, which creates a new channel, q, of type t and communicates it to a process p_2 in order to receive messages from p_2 via channel q. The creation of q is performed by the action $\mathsf{new}(q, \mathit{List}(t))$. There are several ways for p_1 to communicate the channel q to p_2 (e.g., by sending q through a channel or by using parameter passing). Since channels in our setting are modelled as changing constants, passing q to p_2 requires some caution. Indeed, passing the channel q to p_2 does not mean passing the *value* of q but rather its name (or a reference to it) in order to enable p_2 to send messages to p_1 via q.

5 Example of the multiple counters

Our solution for the problem of the multiple counters presented in Sect. 2 separates the counters from the window system. This is similar to real window systems, where applications may run on a machine connected via the network to the machine controlling the monitor. Thus we have two stores: one for the processes modelling the counters, say C, and one for the window system, say X. In this paper, we focus on the component with store C.

The store C describes a theory for counters. We model a counter c as a constant of type *Cnt*, i.e., a pair $\langle \mathit{val}, \mathit{wins}\rangle$, where $c.\mathit{val}$ indicates the current value of the counter c and $c.\mathit{wins}$ is a list of the window identifiers of the windows associated with c, i.e., the windows displaying the value of c. The only (high-level) events (occurring in a counter window) we consider are clicks on the different buttons. Thus we define the type *Evts* to be the set $\{\mathit{inc}, \mathit{copy}, \mathit{link}\}$. In this paper, we assume that the translation of the low-level events, e.g., “click at position (x, y)” is handled by another process. For a counter c, the function *bmp* returns the bitmap to display for c.

The store X of the window system is a description of a theory for windows and contains in the (changing) constant *winlist* descriptions for all the windows currently displayed on the screen. Each descriptor contains at least the identifier of the window, its position, the bitmap to be displayed and the event-queue where the events occurring in the window should be sent to. To simplify, we do not precise the data-structure of *winlist* further.

Figure 3 shows the process abstractions for the processes located on the store C. The process abstraction controlling a counter-window is cnt_ctrl. It has three parameters: the name c of the associated counter, the identifier w of the window and the name e of the event-queue to which the window system sends all the events occurring in the window w. cnt_ctrl takes the (high-level) events (occurring in the window w) one by one from the queue e and reacts accordingly. For instance an event representing a click on the copy-button will create a new counter, named c', and a new event-queue, named e', send a request for creating a new window to the queue *newwins* on the store X and launch a concurrent process for handling the new window for the counter named c'. An event corresponding to a click on the increment-button will increment the counter c and trigger the redrawing of all windows associated to c by executing the elementary action refresh_wins on the store X. refresh_wins has two parameters: a list of window identifiers l and a bitmap b. For each window identifier w in l, it updates the bitmap associated to w in the constant *winlist*.

The creation of a new window proceeds in two steps: requests sent to the store X are eventually acknowledged by the attribution of a new window identifier. These requests are pairs of a bitmap for the window and the name of the event-queue where events occurring in the window should be sent to. Hence, before starting a process controlling a new window, we have to wait for this acknowledgement. For this, we use another process abstraction, namely create_win.

The initial process for the store C starts with creating a counter, initialises this counter and then creates the first window:

$$\left[\text{true} \Rightarrow \begin{array}{l} \langle C,\ \mathsf{new}(c, \mathit{Cnt})\rangle;\ \langle C,\ \mathsf{new}(e, \mathit{List}(\mathit{Evts}))\rangle;\ \langle C,\ c.\mathit{wins}{:=}\mathit{nil}\rangle; \\ \langle C,\ c.\mathit{val}{:=}0\rangle;\ \langle C,\ e{:=}\mathit{nil}\rangle;\ \langle \mathrm{X},\ \mathsf{enq}(\mathit{newwins}, \langle \mathit{bmp}(c), e\rangle)\rangle \end{array}\right];\ \text{create\_win}(c, e)$$

$$
\boxed{\begin{array}{l}
\mathrm{cnt\_ctrl}(c\colon \mathtt{Name}(\mathit{Cnt});\ e\colon \mathtt{Name}(\mathit{List}(\mathit{Evts}));\ w\colon \mathit{Wid}) \Leftarrow \\
\quad \left[\mathit{head}(e) == \mathit{inc} \Rightarrow \begin{array}{l} \langle C,\ c.\mathit{val}{:=}c.\mathit{val}{+}1\rangle;\ \langle C,\ \mathsf{deq}(e)\rangle; \\ \langle \mathrm{X},\ \mathsf{refresh\_wins}(c.\mathit{wins}, \mathit{bmp}(c))\rangle \end{array} \right];\ \mathrm{cnt\_ctrl}(c, e, w) \\
\oplus \left[\mathit{head}(e) == \mathit{copy} \Rightarrow \begin{array}{l} \langle C,\ \mathsf{new}(c', \mathit{Cnt})\rangle;\ \langle C,\ c'.\mathit{wins}{:=}\mathit{nil}\rangle;\ \langle C,\ c'.\mathit{val}{:=}c.\mathit{val}\rangle; \\ \langle C,\ \mathsf{new}(e', \mathit{List}(\mathit{Evts}))\rangle;\ \langle C,\ e'{:=}\mathit{nil}\rangle; \\ \langle \mathrm{X},\ \mathsf{enq}(\mathit{newwins}, \langle \mathit{bmp}(c), e'\rangle)\rangle;\ \langle C,\ \mathsf{deq}(e)\rangle \end{array} \right]; \\
\quad \mathrm{create\_win}(c', e') \parallel \mathrm{cnt\_ctrl}(c, e, w) \\
\oplus \left[\mathit{head}(e) == \mathit{link} \Rightarrow \begin{array}{l} \langle C,\ \mathsf{new}(e', \mathit{List}(\mathit{Evts}))\rangle;\ \langle C,\ e'{:=}\mathit{nil}\rangle; \\ \langle C,\ \mathsf{deq}(e)\rangle;\ \langle \mathrm{X},\ \mathsf{enq}(\mathit{newwins}, \langle \mathit{bmp}(c), e'\rangle)\rangle \end{array} \right]; \\
\quad \mathrm{create\_win}(c, e') \parallel \mathrm{cnt\_ctrl}(c, e, w) \\
\mathrm{create\_win}(c\colon \mathtt{Name}(\mathit{Cnt});\ e\colon \mathtt{Name}(\mathit{List}(\mathit{Evts}))) \Leftarrow \\
\quad \left[\mathit{head}(\mathit{newwids}).\mathit{evt} == e \Rightarrow \begin{array}{l} \langle C,\ c.\mathit{wins}{:=}\mathit{cons}(\mathit{head}(\mathit{newwids}).\mathit{wid}, c.\mathit{wins})\rangle; \\ \langle C,\ \mathsf{deq}(\mathit{newwids})\rangle \end{array} \right]; \\
\quad \mathrm{cnt\_ctrl}(c, e, \mathit{head}(c.\mathit{wins}))
\end{array}}
$$

Fig. 3. process abstractions for processes located on the store C

6 Example of the Dining Philosophers

Coping with mutual exclusion is simple in our framework, since the execution of actions is (locally) atomic, see Sect. 7. Thus, we can easily program the problem of the "Dining Philosophers" by using an action which atomically tests for the presence of the two sticks (or forks) and gets them, if they are present.

We model the situation with two predicates: stick(x) and is_eating(y). The former represents the fact that stick x is lying on the table, and the latter holds whenever philosopher y is eating. We can model the behaviour of a philosopher by the process abstractions of Fig. 4.

$$
\boxed{\begin{array}{ll}
\mathrm{thinks}(x, n\colon \mathit{Nat}) \Leftarrow & \left[\begin{array}{l} \mathrm{stick}(x) \quad \wedge \\ \mathrm{stick}(x{+}1 \bmod n) \end{array} \Rightarrow \begin{array}{l} \langle F,\ \mathsf{del}(\mathrm{stick}(x))\rangle; \\ \langle F,\ \mathsf{del}(\mathrm{stick}(x{+}1 \bmod n))\rangle; \\ \langle F,\ \mathsf{tell}(\mathrm{is\_eating}(x))\rangle \end{array} \right];\ \mathrm{eats}(x, n) \\
\mathrm{eats}(x, n\colon \mathit{Nat}) \Leftarrow & \left[\mathrm{true} \Rightarrow \begin{array}{l} \langle F,\ \mathsf{del}(\mathrm{is\_eating}(x))\rangle; \\ \langle F,\ \mathsf{tell}(\mathrm{stick}(x))\rangle; \\ \langle F,\ \mathsf{tell}(\mathrm{stick}(x{+}1 \bmod n))\rangle \end{array} \right];\ \mathrm{thinks}(x, n)
\end{array}}
$$

Fig. 4. process abstractions for the Dining Philosophers (located on store F)

Note that we need neither low-level synchronisation (like semaphores), nor auxiliary constructions (as placing the table in a separate room or asymmetric philosophers) nor any special assumptions about the underlying logic program.

7 Operational Semantics

To simplify the presentation, we consider in this paper only the operational semantics of a single component, since the semantics presented here can be easily extended to several components. Informally, a component in our framework is a pair $C = \langle \mathcal{F}, \mathcal{P} \rangle$ where $\mathcal{F}$ stands for the store (with storename s) and $\mathcal{P}$ for the set of process abstractions.

The operational semantics of a component has to take into account two different aspects, namely the execution of processes and actions, and interactive goal-solving or expression evaluation. Thus, we present the operational semantics in two steps. First, we describe the execution of processes by a transition system T. Then we combine these rules in an "orthogonal" way with rules describing the interactive use of the store, leading to a second transition system $\mathcal{T}$ which defines the operational semantics of a component.

The operational semantics of a process is the set of sequences of actions that can be observed when executing a process. The execution of a process is described by the transition system $\mathsf{T} = \big\langle \mathsf{Q}, \longrightarrow, \langle \mathcal{F}^{\mathsf{i}}, p^{\mathsf{i}}, \mathit{nil} \rangle \big\rangle$. The states of T, i.e., Q, are triples, e.g., $\langle \mathcal{F}, p, \mathsf{m} \rangle$, consisting of a store $\mathcal{F}$, a process term p and a mail-box m, acting as a fifo-channel on which the component receives (sequences of) elementary actions to execute (emanating from other components). In the sequel, we represent the mail-boxes as lists, and write $[e]$ for the (singleton) list $\mathit{cons}(e, \mathit{nil})$ and $l_1 :: l_2$ for the concatenation of the lists l_1 and l_2. The initial state of T is built from the initial store $\mathcal{F}^{\mathsf{i}}$ and the initial process term p^{i}, which are both specified by the programmer. The transition relation $\longrightarrow$ is defined by a set of inference rules shown in Fig. 5.

According to Rule (R1), execution of the process $\mathsf{success}$ is always possible and yields the special symbol ss. The latter witnesses successful termination.

The execution of a closed action, i.e., an action without free variables, is described by Rule (R2), under the premise of the validity of the guard in the current store ($\mathcal{F} \vdash g$). Since an action may contain elementary actions on several stores, we have to distinguish between elementary actions meant for the local store (denoted by the (storename) constant self) and all the others. In the first case, we update the local store, in the second we send a message containing the sequence of elementary actions (together with their arguments) to the remote component which has to ensure the execution.

Hence, Rule (R2) uses two auxiliary functions, namely *exec* and *sel*. *sel* filters the elementary actions for a given storename from a sequence of pairs of storenames and elementary actions, and *exec* describes the execution of a sequence of elementary actions. Their definitions are as follows:

$$\mathit{sel}(s, l) = \begin{cases} l & \text{if } l \text{ is empty, i.e., } l = \mathit{nil}, \\ \mathit{cons}\big(\mathsf{a}, \mathit{sel}(s, \mathit{tail}(l))\big) & \text{if } \mathit{head}(l) = \langle s', \mathsf{a} \rangle \text{ and } s = s', \\ \mathit{sel}(s, \mathit{tail}(l)) & \text{otherwise.} \end{cases}$$

$$\mathit{exec}(\mathsf{a}_1; \ldots ; \mathsf{a}_n, \mathcal{F}) = \mathsf{a}_n \underbrace{(\cdots (}_{n-2} \mathsf{a}_2(\mathsf{a}_1(\mathcal{F})) \underbrace{) \cdots)}_{n-2}$$

$$(\mathsf{R1})\,\frac{}{\langle \mathcal{F}, \mathsf{success}, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}, \mathsf{ss}, \mathsf{m}'\rangle}$$

$$(\mathsf{R2})\,\frac{\mathcal{F} \vdash g}{\langle \mathcal{F}, \big[g \Rightarrow \big\langle s_i,\ \mathsf{a}_i\big\rangle_i\big], \mathsf{m}\rangle \longrightarrow \big\langle \mathit{exec}\big(\mathit{sel}\big(\mathsf{self}, \big\langle s_i,\ \mathsf{a}_i\big\rangle_i\big), \mathcal{F}\big), \mathsf{ss}, \mathsf{m}'\big\rangle}$$

$$(\mathsf{R3})\,\frac{\begin{array}{l}\big(\mathrm{p}(x_1, \ldots, x_n) \Leftarrow \bigoplus_{i=1}^{m} \alpha_i;\ p_i\big) \in \mathcal{P} \\ \big\langle \mathcal{F}, \big(\bigoplus_{i=1}^{m} \mathit{rename}(\alpha_i;\ p_i)\big)[v_i/x_i], \mathsf{m}\big\rangle \longrightarrow \langle \mathcal{F}', p', \mathsf{m}'\rangle\end{array}}{\langle \mathcal{F}, \mathrm{p}(v_1, \ldots, v_n), \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p', \mathsf{m}'\rangle}$$

$$(\mathsf{R4})\,\frac{\langle \mathcal{F}, p_1, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1', \mathsf{m}'\rangle}{\langle \mathcal{F}, p_1; p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1'; p_2, \mathsf{m}'\rangle} p_1' \neq \mathsf{ss} \qquad (\mathsf{R4}')\,\frac{\langle \mathcal{F}, p_1, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', \mathsf{ss}, \mathsf{m}'\rangle}{\langle \mathcal{F}, p_1; p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_2, \mathsf{m}'\rangle}$$

$$(\mathsf{R5})\,\frac{\langle \mathcal{F}, p_1, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1', \mathsf{m}'\rangle}{\langle \mathcal{F}, p_1 \parallel p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1' \parallel p_2, \mathsf{m}'\rangle} p_1' \neq \mathsf{ss} \qquad (\mathsf{R5}')\,\frac{\langle \mathcal{F}, p_1, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', \mathsf{ss}, \mathsf{m}'\rangle}{\langle \mathcal{F}, p_1 \parallel p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_2, \mathsf{m}'\rangle}$$

$$(\mathsf{R6})\,\frac{\langle \mathcal{F}, p_1, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1', \mathsf{m}'\rangle}{\langle \mathcal{F}, p_1 + p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1', \mathsf{m}'\rangle}$$

$$(\mathsf{R7})\,\frac{\langle \mathcal{F}, p_1, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1', \mathsf{m}'\rangle}{\langle \mathcal{F}, p_1 \oplus p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_1', \mathsf{m}'\rangle} \qquad (\mathsf{R7}')\,\frac{\langle \mathcal{F}, p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_2', \mathsf{m}'\rangle}{\begin{array}{c}\langle \mathcal{F}, p_1 \oplus p_2, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p_2', \mathsf{m}'\rangle \\ \text{if } \nexists\, p_1'(\neq p_1),\ \nexists\, \mathcal{F}'' \text{ s.t. } \langle \mathcal{F}, p_1\rangle \longrightarrow \langle \mathcal{F}'', p_1'\rangle\end{array}}$$

Fig. 5. inference rules defining the transition relation $\longrightarrow$ of T
We omitted the symmetric versions of the rules ($\mathsf{R5}$), ($\mathsf{R5}'$) and ($\mathsf{R6}$).
The side-condition $\exists \mathsf{m}'' : \mathsf{m}' = \mathsf{m} :: \mathsf{m}''$ applies to all rules.

Note 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 step. An example for the usefulness of the atomic execution of actions is the program for the dining philosophers (see Sect. 6) where a philosopher may take the two needed sticks in a single atomic step.

According to Rule ($\mathsf{R3}$), a call to a process abstraction corresponds to execute an (instantiated) *variant* of the definition of the process abstraction. Such a variant is obtained by renaming in the defining "formulæ" of the process abstraction the symbols defined in elementary actions new by fresh ones. We denote this renaming by the operation *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.

Rules ($\mathsf{R4}$) to ($\mathsf{R6}$) describe the standard semantics of the operators ;, $\parallel$ and $+$. In the process-term $p_1 \oplus p_2$, the process p_2 will only be executed when (in the current store) an execution of p_1 is impossible (see Rule ($\mathsf{R7}'$)). In contrary, p_1 can be executed independently from the executability of p_2 (see Rule ($\mathsf{R7}$)).

Intuitively, the modifications of the channel m model the reception of (sequences of) elementary actions emanating from other components of the system. Due to space limitations we do not detail these communications between components further. However, the execution of elementary actions from the channel m is described by the rule (R8):

$$(\mathsf{R8})\frac{}{\langle \mathcal{F}, p, \big[(\mathsf{a}_i)_i\big] :: \mathsf{m}\rangle \longrightarrow \big\langle exec\big((\mathsf{a}_i)_i, \mathcal{F}\big), p, \mathsf{m}'\big\rangle}\exists \mathsf{m}'' : \mathsf{m}' = \mathsf{m} :: \mathsf{m}''$$

Besides the execution of the processes modifying the store, described by the transitions system T, the operational semantics of a component C has another, orthogonal aspect, namely the classical operational semantics of the underlying declarative program used for interactive goal-solving. We suppose that the latter is described by a relation $\twoheadrightarrow$.

Therefore we can describe the operational semantics of a component C via a new transition system, $\mathcal{T} = \langle \mathcal{Q}, \longmapsto, \langle \mathcal{F}^{\mathtt{i}}, p^{\mathtt{i}}, nil, \mathsf{g}^{\mathtt{i}}\rangle\rangle$, where $\mathsf{g}^{\mathtt{i}}$ denotes the (possibly empty) initial goal the user wants to solve. The states of $\mathcal{T}$ are *configurations*, i.e., tuples $\langle \mathcal{F}, p, \mathsf{m}, \mathsf{g}\rangle$, where $\mathcal{F}$ is the current store, p is the current process term, m is the mailbox and g is the current expression to evaluate (according to the operational semantics of the underlying declarative language).

Classically, configurations of a concurrent language are described only by the first two parts, say $\langle \mathcal{F}, p\rangle$, which are enough to express the execution of processes. As for declarative languages, a configuration classically uses the first and the fourth parts $\langle \mathcal{F}, \mathsf{g}\rangle$ which allow to express the rules of the operational semantics $\twoheadrightarrow$ of the declarative language. Combining these two operational semantics adds the possibility to run concurrent applications without loosing the characteristics of declarative languages. For instance, goals can be solved while the processes keep on running.

The transition relation of $\mathcal{T}$, i.e., $\longmapsto$, is defined by the two inference rules:

$$(\mathsf{G})\frac{\langle \mathcal{F}, \mathsf{g}\rangle \twoheadrightarrow \langle \mathcal{F}, \mathsf{g}'\rangle}{\langle \mathcal{F}, p, \mathsf{m}, \mathsf{g}\rangle \longmapsto \langle \mathcal{F}, p, \mathsf{m}', \mathsf{g}'\rangle}\exists \mathsf{m}'' : \mathsf{m}' = \mathsf{m} :: \mathsf{m}'' \qquad (\mathsf{P})\frac{\langle \mathcal{F}, p, \mathsf{m}\rangle \longrightarrow \langle \mathcal{F}', p', \mathsf{m}'\rangle}{\langle \mathcal{F}, p, \mathsf{m}, \mathsf{g}\rangle \longmapsto \langle \mathcal{F}', p', \mathsf{m}', \mathsf{g}^{\mathtt{i}}\rangle}$$

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, we might ask for the eating philosophers by solving the goal is_eating(x). Rule (P) describes the modifications of the store by the processes via the transition system T. When a process modifies the store, we have to restart the goal-solving at the initial goal ($\mathsf{g}^{\mathtt{i}}$), as the modification may invalidate the already achieved derivation. Obviously, rule (P) has to be refined. For instance, if the execution of a process does not alter the definitions used so far in the evaluation of the goal, restarting the goal-solving is unnecessary.

8 Related Work

Due to lack of space we do not survey all the propositions in the area in depth, but focus on some of those which are close to our suggested framework.

One of the main advantages of logic programming is its semantics (least Herbrand model). However, if we consider a Horn clause such as $p(x) \Leftarrow q, p(x)$, the denotation of p is empty according to the classical semantics of logic programs (i.e., for any x, $p(x)$ does not hold). But seen as a process, the semantics of p is the infinite sequence q^ω. Hence, we preferred to distinguish processes and predicates, which is not the case for most concurrent extensions of logic programming.

Besides this fundamental difference, there are several similarities. In concurrent constraint programming (ccp, [24]), agents communicate also via a common (constraint) store. However, the only actions on this store are telling a new constraint and asking if the current store entails a constraint. While the original model is limited to a monotonic evolution of the store, there have been several suggestions for non-monotonic extensions, e.g., [4, 9, 10, 13, 25–27]. These extension either provide new built-in actions [9, 10, 27] or use non-monotonic logics, as logic with defaults [25] or linear logic [4, 13, 26].

The logic programming language Prolog provides two "predicates" that allow one to modify the logic program by adding and deleting clauses, namely `assert` and `retract` [11]. Since these "extra-logical operators" have no declarative reading, they have been interpreted in [5, 8] as send and receive operations on a multiset of atoms, called "blackboard". Our approach is more general, as our stores allow the modification of more than only a multiset of atoms.

The notion of *ports* as a many-to-one communication medium has been introduced in AKL [17]. It is argued that the introduced port primitives have a "logical reading" and preserve the monotonicity of the constraint store. In our non-monotonic setting, we can provide the behaviour of ports via appropriate (elementary) actions, e.g., enq and deq. Recently, the idea of ports, has been extended and integrated into the functional-logic language Curry in order to cope with distributed applications [16].

Pure functional programs can also be easily parallelised: as the evaluation of expressions has no side-effects, the function-arguments can be evaluated in parallel. But this *implicit* parallelism is to be distinguished from (by the programmer) *explicitly* specified concurrency. Several concurrent extensions have been suggested for functional languages, most of them do not distinguish between processes and functions [21, 23].

Concurrent Haskell (CH) [21] introduces new primitives starting processes (e.g., `forkIO`) which can communicate and synchronise via mutable variables (of a new built-in type `MVar`). These primitives are meant as "raw iron from which more friendly abstractions can be built" [21]. This is different from our philosophy: we want to exhibit the high-level abstractions needed for easy programming and not the minimal set of low-level primitives. Also, our actions together with, e.g., "changing constants", are a more flexible support for communication. CML [23] also provides new primitives, supporting a synchronisation mechanism based on "events", a new abstract type representing "potential communication". To effectively communicate, a process has to synchronise on an event.

Facile [28] extends the functional language ML with "behaviour-expressions" and (synchronous communication) "channels". As behaviour-expressions can be transformed into a value, called a process "script", programming with processes in a functional style is supported. A script `s` can be executed by evaluating `spawn s` which has the side-effect of spawning a new process. Thus, Facile allows functions and processes to use each other mutually. In our framework, the use of processes for the definition of functions or predicates is not necessary. However, the communication between processes in Facile is restricted to message passing through channels, which is a special case of the communication via stores.

Our framework extends classical process algebras, e.g., CCS [19], CSP [6] or ACP [3] in several aspects. First, our processes (and actions) are parameterised. Second, our actions have a precise semantics as transformations of a global store. Last but not least, our framework provides the possibility of (interactive) declarative programming, i.e., evaluation of expression and goal solving, along with concurrent execution of processes.

Our elementary action new together with the parameterised type $\mathtt{Name}(t)$ allows to model mobility in the same way as the π-calculus [20]. However, the communication mechanism of the π-calculus is synchronous and exclusively based on message passing (thus one-to-one), whereas our framework is based on asynchronous, broadcasting communication via a shared store. Thus, our proposal is more related to the asynchronous trend of the π-calculus. Our guards allow the atomic reception on several channels, whereas in the π-calculus processes can only wait on a single channel. Extensions of the (asynchronous) π-calculus without this restriction are the join-calculus [14] and $\mathcal{L}_\pi$ [7].

In the join-calculus [14] processes can be seen as communicating via a multiset of messages: sending a message corresponds to place it in the multiset, and 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 π-calculus-based language.

Last but not least, promising approaches using linear logic as a basis to define frameworks for the description of the semantics of higher-level languages which integrate declarative programming with object-oriented or concurrent features have been proposed in the literature, see e.g., [2, 7, 18]. These approaches can be adapted to our framework in order to give an alternative semantics based on proof search in linear logic.

9 Conclusion

We have presented a generic framework for the combination of mobile processes and declarative programming languages. Our proposition differs from most related frameworks in that it distinguishes syntactically and semantically between the notion of processes and that of functions or predicates. Thus our framework does not force programmers to encode some concepts by means of others, but allows to express everything by the most appropriate concept.

Processes are parameterised and communicate asynchronously via the modification of stores. Classical communication media, namely channels, are modelled as (changing) constants. This view of channels has both, a functional reading and a (constraint) logical reading. New channels (as well as new functions) can be created thanks to the (built-in) action new. This action, combined with the parameterised type `Name`(t), allows one to model mobility and thus generalises the notions introduced in the π-calculus to declarative languages.

The complete formal definitions of this framework together with the extension to several components can be found in [12]. The implementation of a prototype is under progress.

Acknowledgements. We are grateful to Bernd Braßel, Michael Hanus and Frank Steiner for fruitful discussions on a preliminary version of this paper.

References

1. J.-R. Abrial. *Formal Methods for Industrial Applications*, volume 1165 of *LNCS*, chapter Steam-Boiler Control Specification Problem, pages 500–510. Springer, 1996.
2. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3–4):445–473, 1991. selected papers from the 7^{th} Int. Conf. on Logic Programming, 1990.
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. E. Best, F. de Boer, and C. Palamidessi. Partial order and SOS semantics for linear constraint programs. In D. Garlan and D. L. Métayer, editors, *Proc. of the* 2^{nd} *Int. Conf. on Coordination: Languages and Models (Coordination '97)*, volume 1282 of *LNCS*, pages 256–273, Berlin, September 1997. Springer.
5. A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM TOPLAS*, 13(1):99–123, January 1991.
6. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.
7. L. Caires and L. Monteiro. Verifiable and executable logic specifications of concurrent objects in $\mathcal{L}_\pi$. In C. Hankin, editor, *Proc. of the* 7^{th} *European Symp. on Programming (ESOP '98)*, volume 1381 of *LNCS*, pages 42–56, Lisbon, March–April 1998. Springer.
8. M. Carro and M. Hermenegildo. Concurrency in prolog using threads and a shared database. In *Proc. of the* 16^{th} *Int. Conf. on Logic Programming (ICLP '99)*, Las Cruces, November 1999. MIT Press.
9. P. Codognet and F. Rossi. Nmcc programming: Constraint enforcement and retraction in cc programming. In *Proc. of ICLP '95.* MIT Press, 1995.
10. F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Non-monotonic concurrent constraint programming. In *Proc. of the Int. Symp. on Logic Programming (ILPS '93)*, pages 315–334. The MIT Press, 1993.
11. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual.* Springer, 1996.
12. R. Echahed and W. Serwe. A multiparadigm programming scheme. Technical Report, forthcoming.
13. F. Fages, P. Ruet, and S. Soliman. Phase semantics and verification of concurrent constraint programs. In *Proc. of the* 13^{th} *Annual IEEE Symp. on Logic in Computer Science (LICS '98)*, 1998.
14. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. of the* 23^{rd} *Symp. on Principles of Programming Languages (POPL '96)*, pages 372–385, St. Petersburg Beach, Florida, January 1996.

15. 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.
16. M. Hanus. Distributed programming in a multi-paradigm declarative language. In G. Nadathur, editor, *Proc. of the Int. Conf. on Principles and Practice of Declarative Programming (PPDP '99)*, volume 1702 of *LNCS*, pages 188–205, Paris, 1999. Springer.
17. S. Janson, J. Montelius, and S. Haridi. Ports for objects in concurrent logic programs. In Agha, Wegner, and Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.
18. D. Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, September 1996.
19. R. Milner. *A calculus of communicating systems*, volume 92 of *LNCS*. Springer, 1980.
20. R. Milner, J. G. Parrow, and D. J. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992.
21. S. L. Peyton Jones, A. D. Gorden, and S. Finne. Concurrent Haskell. In *Proc. of the* 23^{rd} *Symp. on Principles of Programming Languages (POPL '96)*, pages 295–308, St Petersburg Beach, Florida, January 1996.
22. 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*. MIT Press, 1998.
23. J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
24. V. A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.
25. 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.
26. V. A. Saraswat and P. Lincoln. Higher-order, linear, concurrent constraint programming. Technical report, Xerox PARC, 1992.
27. G. Smolka. *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, chapter The Oz Programming Model, pages 324–343. Springer, 1995. Jan van Leeuwen (Ed.).
28. 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, 1996.
29. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.
30. P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, February 1998.