# A Component-Based Approach to Concurrent 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.** In this paper we present a component-based approach to mobile concurrent declarative programming, where we model systems as sets of interacting components. We first give a *definition* of a component and its different constituents. Finally, we briefly present a prototype implementation through an example.

## 1 Introduction

Classical declarative languages, i.e., functional, logic and functional-logic languages, aim at providing high-level descriptions of systems. These languages have well-known nice features, such as abstraction, readability, compilation techniques, proof methods etc. However, the concepts of functions and predicates, underlying classical functional-logic languages, are not sufficient to capture the whole complexity of real-world applications where interactivity, concurrency and distributivity are needed [29].

On the other hand, component models, as for instance [5, 11, 17], allow the construction of complex (software) systems by assembling components which are characterised by their interface through which they can interact with their environment, e.g., other components or the user. While these models seem to allow short development times and high degrees of reuse, we are not aware of any common formal definition of a component.

In this paper we present a component-based approach to mobile concurrent declarative programming. We propose a *definition* of a component, extending the framework of [9] to several interacting components that may be written in different languages.

We model a system as a set of interacting components, which may be distributed over a network or reside on a single computer. Each component will be identified by a component-name $\mathtt{s}$. Internally, a component is organised as a set of processes $\mathtt{p}_i$, i.e., a concurrent program, and a set of formulæ $F$, i.e., a traditional declarative program, called store. Hence, the execution model of a system can be pictured as in Fig. 1. Processes ($\mathtt{p}_i$) communicate by modifying the stores, i.e., by altering, in a non-monotonic way, the current theories described by the stores, for example by simply redefining constants (e.g., assignment of a new value) or by adding or deleting formulæ in $F$.

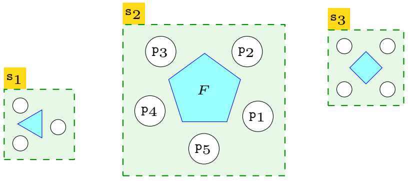


**Fig. 1.** Execution Model of a System

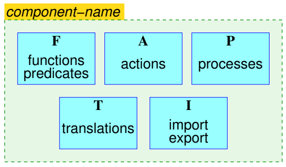


**Fig. 2.** A Programmer's View of a Component

All the changes of the stores are the result of the execution of *actions*. Roughly speaking, an action `{s @ a}` is a pair of an elementary action `a` and a storename `s` denoting the store $F$ on which the action is to be executed. Examples of actions are `{s @ c := v}`, to change the definition of the (variable) constant `c` (which is defined in the store named `s`) to the value `v` and `{Display @ (print 'hello')}` to print the string `hello` on the display `Display`.

Orthogonally to the execution of actions by processes, the stores of the components can be used as usual, i.e., for goal solving or the evaluation of expressions (to normal-form). This shows that our framework is a conservative extension of declarative programming, since a component without any processes corresponds to a declarative program in the classical sense.

As we focus in this paper on the description and definition of components, i.e., their internal structure, we make some simplifying assumptions about the communication between components. For example, we suppose that the underlying communication network is fault-free, that is to say, that all messages sent arrive eventually, in the order they have been sent and without duplication.

The rest of this paper is organised as follows. In the next section we give the definition of a component and illustrate its different constituents. Then we briefly present the principles of a prototype implementation and give an example of a program. We compare our framework to some related work in section 5. Finally, section 6 concludes. For the rest of the paper, we assume the reader is familiar with classical notions of rewriting [8, 16] as well as process algebra [3].

## 2 Components

Besides the (initial) store or declarative program and the definition of processes, a component will be characterised, on the one hand, by the set of actions which can be used for the modification of its store and, on the other hand, by its interface to the environment. Thus, we get the following definition of a component which is symbolized in Fig. 2.

**Definition 1.** *A* component *(identified by a* (component-) *or* (store-) name*)* `s` *is specified by five different (but interdependent) parts, namely*

***F*** *the (initial)* store *or declarative program,*
***P*** *the definitions of* processes*, together with the initial process term,*

***A*** *the definitions of the* actions *that are executable on the store $F$,*
***T*** *the definitions of* translations *for communicating values of the store to other stores that are possibly written in a different language and*
***I*** *the* imported *(respectively,* exported*) definitions from (respectively, to) the environment, i.e., other components of the system.*

In addition to these five parts, a component has also a *mailbox*, associated to it that is used for the interaction between components. In fact, the actions to be executed on a remote component are sent (via the communication network which we assume to be fault-free) to its mailbox. It is then up to the remote component to ensure the execution of these actions. Thus this mailbox is handled by the implementation and thus transparent for the programmer. Notice that any of the parts of a component may be missing. For instance, a pure declarative program or component will need neither processes, nor interaction with other components.

In the following, we will detail the contents of the parts one by one, using as a running example a system of multiple counters inspired from [14]. The application starts by creating a window (as shown in Fig. 3) representing a counter which can be incremented manually. In the counter window, 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 and the latter creates a new view (i.e., a new window) of the same counter. All views of a same counter should behave identically, e.g., they increase the counter at the same time.

This example illustrates some of the difficulties related to concurrent processes, such as dynamic creation of new constants (e.g., counters, windows and the associated 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" [28]. Our solution uses two components, one for the counters, with storename (or component name) `counters`, and another for the window system with storename `X11`. In this section, we present (parts of) the definition of the component `counters`.

### 2.1 *F* – Store

As mentioned in section 1, the store of a component is a classical declarative program. In this paper, we consider a simple functional-logic language based on constructor-based conditional term rewriting systems. For the examples, we will use a syntax similar to Curry [6]. Thus, a program is a pair of a signature and a set of (conditional) rewrite rules. A *signature* is a pair of a set of *sorts* (including `Bool`, the sort of boolean values) and a family of sorted *functions*. We distinguish between *constructors*, i.e., operations which are used to construct data terms, and *defined functions* or *operations* that operate on data terms. Predicates are modeled as functions yielding the predefined type `Bool`. A *(conditional) rewrite* *rule* is a triple "`lhs | cond = rhs`", where `lhs` is a linear pattern, i.e., an operation rooted term in which any variable occurs at most once, `rhs` is a term and the (optional) condition `cond` is a boolean valued term. We require that the free variables of `rhs` and `cond` are included in the free variables of `lhs`. The operational semantics of programs is defined by the evaluation strategy of weakly needed narrowing [1].

Considering our example, we model a counter as a pair of its current value and a list of windows associated to it. Thus the store for the counters defines a data type of counters, i.e., a pair of the current value and a list of identifiers of associated windows, represented as `strings`. Additionally, we define a data type of messages corresponding to the messages sent when a user clicks on the buttons of a counter window[1]. Since constructors are not defined by rules, they are declared along with the declaration of the corresponding `data` type.

```
data msg     = incr | link | copy | quit.
data counter = cnt natural [string].

value :: counter -> natural
value (cnt v a l) = v

windows :: counter -> [string]
windows (cnt v a l) = l
```

### 2.2 *A* – Actions

The actions that can be used (by the processes or the environment) to modify the store of a component are defined in this part, using a special language, called action description language (ADL) [10]. A natural candidate for an ADL is the language in which the declarative language is implemented, since an abstract data type (ADT) representing programs already exists. We call this ADT of programs `store`. In our implementation of the declarative language in `ocaml`, an (elementary) action can be defined as an `ocaml`-function of the type:

```
type elementary_action_code = parameter list -> store -> store
```

Due to space limitations, we cannot present `store`, the ADT of programs, completely, nor can we give detailed examples of definitions of actions. The interested reader may consult [10] for more details. Some classical actions are for instance `tell`, `del`, `:=`, `enq` and `deq`.

Informally, the action `(tell (lhs = rhs))` adds the rule `lhs`↓ `=` `rhs`↓ (here, `term`↓ stands for the normal-form of the term `term`) to the store, whereas `(del pattern)` removes all rules whose left-hand side match `pattern` from the store. In the sequel we will abbreviate `(tell (term = true))` for boolean `terms` to `(tell term)`. Certainly the most common elementary action is assignment `c := v`, which takes two parameters: the name `c` of a constant (traditionally considered as a variable) and a (new) value `v`. Notice the need for introducing a new parameterised type `Name(t)` to denote the type of the name of a symbol of

[1] We suppose predefined the types `strings`, `natural` numbers and polymorphic lists with elements of type `t`, written as `[t]`.

type `t`. If `c` is of type `Name(t)`, we denote by `!c` the associated symbol of type `t`. Executing `(enq q m)` puts the message `m` into the queue named `q`, and the execution of `(deq q)` removes the head of the queue named `q`.

Another important elementary action handles the creation of new symbols: `(new s t)` introduces two new symbols in the store, namely `s` of type `Name(t)` and `!s` of type `t`. `s` stands for the name of (or a reference to) the symbol `!s`. This elementary action together with the parameterised type `Name(t)` allows modeling mobility in the same way as the $\pi$-calculus [18]. In the multiple counters example, `new` allows the creation of new counters and the associated communication channels with the window-system.

### 2.3 *I* – Imports and Exports

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. 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 forming a store, whereas the latter is necessary for interaction between components, since a component must be able, for example, to construct the parameters of an action to be executed on a remote store.

Interaction between components in our framework is based upon the execution of action on the stores of remote components. Hence, 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 associated declarations of the store, e.g., sorts, functions and predicates, have to be imported. To avoid name-clashes, the "names" of the declarations could be prefixed with the name of the component they are defined in, similar to the prefixing of the module name in `ocaml` for example.

The actions imported from the component `X11` in charge of the graphical user interface (GUI) for the example of the multiple counters are specified as follows:

```
COMPONENT X11
action new_window ::
    int -> Name(counter) -> storename -> Name([msg]) -> store -> store.
action refresh_windows :: int -> [window] -> store -> store.
END
```

The action `new_window` takes the name of a counter and its current value, the name of a message-queue (to which the messages corresponding to clicks on the buttons in a window should be sent) and the storename of the store of the counters and has as effect the creation of a new window for the counter, displaying its current value. `refresh_windows` takes the current value of a counter and a list of windows (associated to the considered counter) and refreshes the value displayed in all the windows.

The exported declarations of a component are those which can be used by other components. Obviously they constitute a subset of the declarations of the component.

### 2.4 $T$ – Translations

Consider a system of several components the stores of which are written in different declarative languages. When the processes of such a system are to interact, the values sent from one (store) to another (store) have to be *translated*, i.e., values of one language have to be transformed into values of the other. Here we mean by values ground constructor terms, i.e., terms in normal-form. It seems natural to require a translation to be a total recursive function, since to any value that is to be communicated has to correspond a unique translation which should be computable.

Note that these translation functions cannot be part of one of the stores involved, since they define relations between objects in both of the languages. In fact, they can be seen as functions in a "union-language" that combines both stores, i.e., two programs written in different languages. Furthermore, a translation may need to use the operational semantics of the declarative languages, in order to reduce terms to normal-forms before and after the translation, if necessary.

We suggest to specify translations (from language $\mathcal{L}_1$ to language $\mathcal{L}_2$[2]) via a constructor-based term rewriting system, and to separate a translation into three steps. First, the term to be translated is reduced to normal-form (using the operational semantics of language $\mathcal{L}_1$). Then a corresponding expression is generated, by application of a special *translation function*. Finally the expression is reduced to normal-form (using the operational semantics of language $\mathcal{L}_2$) yielding the translation of the original term. The motivation of this separation is to let a programmer just specify the translation functions, and put the handling of the other phases into the implementation. Thus a translation function associates to a ground term `t` in normal-form a term `t'` whose normal-form corresponds to the translation of the term `t`.

A simplistic translation of the type of `natural` numbers which can be defined as `data natural = z | s natural` to the `int`'s of `ocaml` is

```
(int_of_natural    z ) = 0
(int_of_natural (s x)) = ((+) (int_of_natural x) 1)
```

Another possibility for interaction between components written in different languages is to suppose that both components understand a common language, and to provide built-in translations for the elements of this third language. Examples for this approach are, e.g., the Interface Definition Language (IDL) of the Common Object Request Broker Architecture (CORBA) [20] or the interface between `ocaml` and C.

### 2.5 $P$ – Processes

In our framework, processes are defined in the style of a process algebra (see, e.g., [3]). In this section, we restrict ourselves to the syntax of the current prototype implementation which we will present in section 3. For more details on the definition of processes, we refer the interested reader to [9].

[2] Notice that when $\mathcal{L}_1$ and $\mathcal{L}_2$ are the same language, a translation can be used to bridge between different (internal) in the two components involved.

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

$$\texttt{p} ::= \texttt{success} \mid (\texttt{q}\ \texttt{t}_1\ \ldots\ \texttt{t}_n) \mid \texttt{p;}\ \texttt{p} \mid \texttt{p||p}$$

The basic process terms of our framework are calls to processes $(\texttt{q}\ \texttt{t}_1\ \ldots\ \texttt{t}_n)$. The process `success` represents the process which terminates successfully. Process terms can be composed sequentially (`;`) or in parallel (`||`). Processes are defined by a set of bodies ordered by priority. Each body consists of a guarded action and a process term. A *guarded action* $[\texttt{g} \Rightarrow \{\texttt{s}_1\ @\ \texttt{a}_1\};\ \ldots;\ \{\texttt{s}_n\ @\ \texttt{a}_n\}]$ is composed of a guard `g`, i.e., a conjunction of atoms or boolean expression, and a sequence of pairs $\{\texttt{s}_i\ @\ \texttt{a}_i\}$ of a storename $\texttt{s}_i$ and an elementary action $\texttt{a}_i$ to be executed on the store $\texttt{s}_i$. The execution of a guarded action is locally atomic, i.e., the check of validity of the guard together with the sequence of actions for the local store are to be executed atomically, and all (sub-)sequences of actions for a given (remote) storename are to be executed atomically when they are received at the remote store.

Each counter window is controlled by a process `cnt_ctrl`. It has two parameters: the name `c` of the associated counter and the name `e` of the event-queue to which all the events occurring in the window are sent. Thus `cnt_ctrl` takes the events occurring in the window it controls one by one from the queue `e` and reacts accordingly. The definition of `cnt_ctrl` is as follows:

```
process cnt_ctrl c e :-
 [(head !e) == incr =>
  {counters @ (deq e)};
  {counters @ c := (cnt ((value !c) + 1) (windows !c))};
  {X11      @ (refresh_windows (int_of_natural  (value   !c))
                             (list_of_winlist (windows !c))})];
 (cnt_ctrl c e),

 [(head !e) == link =>
  {counters @ (deq e)};
  {counters @ (new e1 [msg])}; {counters @ e1 := nil};
  {X11      @ (new_window (int_of_natural (value !c)) c counters e1)}];
 (cnt_ctrl c e)

 [(head !e) == copy =>
  {counters @ (deq e)};
  {counters @ (new c2 counter)}; {counters @ c2 := (cnt (value !c) nil)};
  {counters @ (new e2 [msg])};   {counters @ e2 := nil};
  {X11      @ (new_window (int_of_natural (value !c2)) c2 counters e2)}];
 (cnt_ctrl c e)
end
```

For instance, `cnt_ctrl` will react to a click on the copy-button by creating a new counter `c2` and a new event-queue `e2`, initialising these new constants appropriately, creating a window for the new counter and launching a concurrent process handling the new window for the new counter `c2`. Notice that not all parameters of the actions executed on the store `X11` have to be translated. In fact, we suppose that all components share a common "process language". For instance, we do not need to translate storenames.

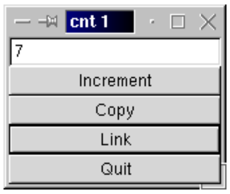


**Fig. 3.** A Counter Window

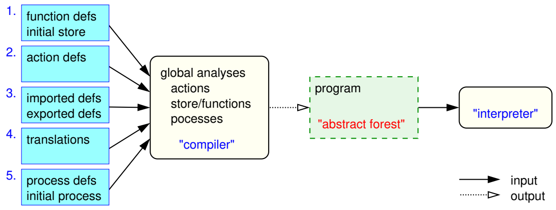


**Fig. 4.** Global Vision of the Interpretation Process of a Component in Sabir

Besides the definitions of the process abstractions, this part contains also the initial process term, i.e., the process which will be executed when the program is started. Note that the initial process term may be a call to the special process `success`, which means that no processes are to be executed. In the example of the counters, the initial process term is a call to a (parameterless) process `start`, which creates a new counter, a new message queue and a corresponding window, and then starts a process for controlling the counter-window.

## 3 A Prototype Implementation

The general scheme of the interpretation process for a single component in our prototype, which we call Sabir, is shown in Fig. 4. For each component in the system, an interpreter has to be started separately (where the addresses of the mailboxes of the different component are mutually known). In this paper we give just a brief overview of the architecture of the prototype implementation. For a description of the operational semantics of a component we refer the reader to [9].

In Sabir, a component is described by a set of five separate files corresponding to the different parts. The names of the files are obtained by adding different suffixes to the storename of the component. These files are processed by the "compiler", in the order indicated by the numbers in Fig. 4. Using all this information, the "compiler" produces an "abstract forest" (or parse-tree), i.e., an internal, intermediate representation of the component. Finally, the "interpreter" executes the component using this internal representation, that is to say, a handler for the mail-box is set up, the connections to the other components are established, the execution of the initial process is started and two (interactive) interpreters are launched. The first is an interpreter of the declarative language for the store and the second is an interpreter for the interactive execution of actions on the store. The former can be used for interactive use of the theory described by the store, and the second allows a user to update the theory (or program) remotely, similar to the primitives for exchanging the code of modules in Erlang [2].

To simplify the implementation, all declarations of stores are considered as imported (respectively, exported) in Sabir (we just have to read the file defining the initial program of the remote component). However, the set of actions executable on the remote store have to be mentioned explicitly.

To allow for a more efficient execution of actions during interpretation of a component, actions are used in form of *compiled* `ocaml`-functions that are (dynamically) linked to the interpreter. Thus, before the interpretation of a component, the definitions of actions have to be compiled using the standard `ocaml`-compiler and a library containing the necessary definitions of Sabir.

## 4 Example of a Lift Controller

Consider a building with $m$ floors in which a system of $n$ lifts is installed. On every floor there is a button for requesting a lift, and each lift is equipped with buttons for ordering the lift to stop at a given level. Suppose we are to model this system, i.e., control the lifts, ensuring that all orders and requests are eventually handled. For simplicity, we do not consider the capacity of the lifts, assuming that it is always sufficient for the requests to handle.

Roughly speaking we will model the lifts as independent processes that share the information of requests coming from the floors. Hence, our model does not depend on the numbers of floors and lifts, and we can, for instance, easily add an auxiliary lift. This feature might come in handy when we want to use the model for the evaluation of the number of lifts actually needed for the building. In the following we give some samples of the description of the component modeling the lifts. Due to space limitations, we do not show the complete component, but give only a sample of the most interesting parts.

As in the counters example, we suppose that we have a second component, namely `X11`, which is used to display a GUI for the lifts. The actions executable on the component `X11` are the following:

```
action move_to_floor :: Name(lift) -> int -> store -> store.
action open_doors    :: Name(lift) -> int -> store -> store.
action close_doors   :: Name(lift) -> int -> store -> store.
```

Their first parameter denotes the (name of the) lift, and the second the current level of this lift. We also use the translation `int_of_natural` of the counters, presented in section 2.4. Figure 5 shows a sample of the store for the `lifts` and Fig. 6 gives the definition of the lift-control process `lift_ctrl`.

The intuitive idea of our model is as follows. We distinguish between the `requests` issued by the buttons on the floors which can be handled by any of the lifts and `orders` which have to be handled by a given lift. `orders` are either issued by the buttons inside the lift or requests that have been assigned to the lift. In order to optimise the assignment of requests to lifts, we will try to make the lifts move as long as possible in one direction, i.e., either `up` or `down`. Thus we can model a `lift` as a triple `(L dir pos orders)` containing its current level or `position` (represented by a `natural`), its current `direction` and a list of `orders` to handle, represented as a list of floors-numbers. Orders from the buttons inside a lift are directly put in the list of `orders` of the lift, while requests on the floors are put into a list `requests` shared by all lifts. A process controlling the lift

```
data direction = up | down.
data lift      = L direction natural [natural].

order_to_handle (L d p (cons x xs)) = (member p (cons x xs))
order_to_handle (L d p nil)         = (member p requests)

rm_request p nil                  = nil
rm_request p (cons x xs) | p == x =         (rm_request p xs)
rm_request p (cons x xs) | p < x  = (cons p (rm_request p xs))
rm_request p (cons x xs) | x < p  = (cons p (rm_request p xs))

rm_order (L d p r) = (L d p (rm_request p r))

request_in_dir (L up   p o) | (nearest p (head requests)) =
        p <= (head requests)
request_in_dir (L down p o) | (nearest p (head requests)) =
      (head requests) <= p

nearest p r = (all_above (dist p r) (get_dists r lifts_pos))

all_above x nil                  = true
all_above x (cons y ys) | x <= y = (all_above x ys)

get_dists x nil         = nil
get_dists x (cons y ys) = (cons (dist x y) (get_dists x ys))

dist z     z     = z
dist z     (s x) = (s x)
dist (s x) z     = (s x)
dist (s x) (s y) = (dist x y)

next    x  up   | x < top = (s x)
next (s x) down           =    x

next_floor (L d p r) = (L d (next p d) r)
```

**Fig. 5.** Sample of the store for the `lifts` example

`ln` moves a request from the list `requests` into the list of `orders` of `ln`, if the request is in the current direction of `ln`and if `ln` is among the lifts that are nearest to the request. Whenever a lift handles an order all requests for the same floor are handled as well.

A definition for the process `lift_ctrl` controlling a lift named `ln` is shown in Fig. 6 (where we denote by `s^` the name of a symbol `s`). The process is defined by four "rules". The guard of the first rule, i.e., `(order_to_handle !ln)`, checks if there is an order (or request) to handle on the current floor (of the lift `ln`). Handling an order (and/or request) means to remove it from the lists of orders (and/or requests). The third action displays the handling in the GUI by executing the action `open_doors` on the component `X11`. After executing the actions, the processes calls the special process `wait` which waits a certain time. Finally the process `handle` is called, closing the doors and reentering the handling loop. The second rule of the process `lift_ctrl` describes the transformation of the first request of the list `requests` into an order of the lift. The third rule moves the lift, if it has pending orders in its current direction. Finally, the last rule

```
process lift_ctrl ln :-
  [(order_to_handle !ln) =>
    {lifts @ ln        := (rm_order   !ln)};
    {lifts @ requests^ := (rm_request (pos !ln) requests)};
    {X11   @ (open_doors ln (int_of_natural (pos !ln)))}];
  wait; (handle ln),

  [(request_in_dir !ln) =>
    {lifts @ ln        := (add_lift_requests (head requests) !ln)};
    {lifts @ requests^ := (tail requests)}];
  (lift_ctrl ln),

  [(order_in_dir !ln) =>
    {lifts @ lifts_pos^ := (rm_first (pos !ln) lifts_pos)};
    {lifts @ ln         := (next_floor !ln)};
    {lifts @ lifts_pos^ := (pos !ln) :: lifts_pos};
    {X11   @ (move_to_floor ln (int_of_natural (pos !ln)))}];
  (lift_ctrl ln),

  [(further_requests !ln) => {lifts @ ln := (opposite !ln)}];
  (lift_ctrl ln)
end

process handle ln :-
  [true => {X11 @ (close_doors ln (int_of_natural (pos !ln)))}];
  (lift_ctrl ln)
end
```

**Fig. 6.** Processes for the lifts example

changes the direction of the lift, if it has orders in the opposite direction. If none of the guards applies, the lift just waits, since there are no orders or requests for it to handle.

Some of the functions used for the description of the process are shown in the sample of the store in Fig. 5. Accessing the fields or constituents of a `lift` is possible by means of the functions `dir`, `pos` and `orders`. Informally, there is an order (or request) to handle for a lift, whenever the current position of the lift is a `member` of the list of orders of the lift or of the list of global `requests`. The definition of `order_to_handle` reflects this intuition. The function `rm_request` (respectively, `rm_first`) takes a request, i.e., natural number `n`, and a list of naturals `nat_list` and returns a list of naturals which is obtained from `nat_listl` by removing all (respectively, the first) occurrences of `n`. `rm_order` removes the current position from the list of `orders` of a lift. All these removal functions behave as the identity function when the list does not contain the position to be removed. When assigning `requests` to lifts we choose a lift (at position `p`) which is among the `nearest` ones to the request `r` using `lifts_pos`, a list of the current positions of all lifts. Notice that this list is also modified when moving a lift. The next floor of a lift is described by the function `next_floor`, where `top` denotes the top floor of the building.

Using a similar programming style as in the example of the multiple counters, we can easily extend our program to allow the dynamic creation of lifts. While this feature may seem not very realistic, it might be useful when the program is to be used as a simulation to determine the number of lifts actually needed in a building. For this purpose, we have simply to add a process that adds requests according to a stochastic distribution, and to observe the evolution of the number of outstanding requests.

## 5 Related Work

Most existing concurrent extensions of declarative languages, e.g., [2, 7, 15, 21, 23–25, 27], do not distinguish clearly between the different constituents of a component. In these languages, processes are rather encoded in terms of the concepts underlying the declarative language, actions are "built-in", and interaction and translations are hidden or intertwined with the store. As an example, consider the process scripts of CML [23] or the behaviour-expressions of Facile [27], which allow functions and processes to call one another mutually, making difficult to reason about these two concepts separately. Thus these extant approaches seem to be tailored to a specific language hindering a straightforward extension to a general framework.

Similar to concurrent extensions of declarative languages, where processes have to be encoded as, e.g., functions or predicates, programming languages uniquely based on process calculi encode the notions of functions and predicates via processes [12, 22].

Coordination languages, e.g., [4, 13, 19], model the interaction between processes, most of them following the model of Linda [13], where processes communicate via a (hierarchy of) shared tuple space, using several primitive actions, namely primitives for adding, reading and removing tuples. Our approach is more general since we avoid the need to encode complex communication structures by using use (declarative) programs instead of simple tuples spaces. Furthermore, in our framework the set of action can be defined by the programmer, whereas the set of Linda-operations is fixed.

The current popular component models, as for instance the (Distributed) Component Object Model ((D)COM) [17] or JavaBeans [11], focus mainly on the composition of systems using components as basic building blocks of components and systems. These approaches regard components as "black-boxes", the structure of which is left unspecified, since the essential properties for a *user* of a component is its *interface* [26]. This implies that, in order to build any system, a set of predefined or built-in components has to be provided, for instance in form of a component library. The only definition of a component we are aware of is given in [5], where a component is characterised by a function mapping input streams to output streams. Hence this definition also considers merely the input/output behaviour and neglects the internal structure of a component.

In contrary, we have given in this paper a description of components *including* their structure (and semantics) in the domain of concurrent declarative programming. Thus our approach allows the definition of the atomic built-in components necessary for the approaches mentioned before. This combination of the composition principles, as in the current component models, with the definition of components, as in our approach, seems to be a promising field of research.

## 6 Conclusion and Future Work

In this paper we have presented a component-based approach to concurrent declarative programming. In this framework, components are defined by means of five different parts and may be written in different languages. Interaction is based on the modification of declarative programs using user-definable actions. We have illustrated the principles by a short presentation of a prototypical implementation of an interpreter and some examples of programs.

Extension of our approach to imperative stores is possible. However, the semantics of the framework risks to become hard to understand if the imperative language is used in an unrestricted manner, e.g., side-effects (assignments of global "variables") during the evaluation of a guard.

Currently, we plan to several further additions to the framework and their implementation, namely the introduction of notions related to time and the combination with dedicated constraint solving algorithms.

## References


1. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of ICLP '97*, pp. 138–152, 1997. The MIT Press.
2. J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.
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. J.-P. Banâtre and D. L. Métayer. Programming by multiset transformation. *CACM*, 36(1):98–111, Jan. 1993.
5. M. Broy. A uniform mathematical concept of a component. *Software: Concepts and Tools*, 19(1):57–59, 1998.
6. Curry: An integrated functional logic language. available at `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.
7. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual*. Springer, 1996.
8. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chap. 6, pp. 243–320. Elsevier, Amsterdam, 1990.
9. R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In J. Lloyd et al., editors, *Proc. of CL 2000*, vol. 1861 of *LNAI*, pp. 300–314, July 2000. Springer.
10. R. Echahed and W. Serwe. Integrating action definitions into concurrent declarative programming. In *Proc. of WFLP 2001*, Kiel, Sept. 2001. This volume.

11. R. Englander. *Developing Java Beans.* The Java Series. O'Reilly & Associates, Inc., June 1997.
12. C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. *The JoCaml language (beta release): Documentation and User's Manual.* INRIA, Jan. 2001. available at `http://pauillac.inria.fr/jocaml/htmlman/index.html`.
13. D. Gelernter. Generative communication in Linda. *ACM TOPLAS*, 7(1):80–112, Jan. 1985.
14. GUI Fest '95 Post-Challenge: Multiple counters. available at `http://www.cs.chalmers.se/~magnus/GuiFest-95/`, July 1995. organized by S. Peyton Jones and P. Gray as part of the Glasgow Research Festival.
15. M. Hanus. Distributed programming in a multi-paradigm declarative language. In G. Nadathur, editor, *Proc. of PPDP '99*, vol. 1702 of *LNCS*, pp. 188–205, 1999. Springer.
16. J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pp. 1–112. Oxford University Press, 1992.
17. Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, Oct. 24, 1995. version 0.9, available at `http://www.microsoft.com/com/resources/comdocs.asp`.
18. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999.
19. 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.
20. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, revision 2.4.2 edition, Feb. 2001. available at `http://www.omg.org/cgi-bin/doc?formal/01-02-33`.
21. S. L. Peyton Jones, A. D. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL '96*, pp. 295–308, St Petersburg Beach, Florida, Jan. 1996.
22. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report 476, CSCI, Indiana University, Mar. 1997. Dedicated to Robin Milner on the occasion of his $60^{th}$ birthday.
23. J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
24. V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, 1993.
25. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, vol. 1000 of *LNCS*, pp. 324–343. Springer, 1995.
26. C. A. Szyperski. Emerging component software technologies–a strategic comparison. *Software: Concepts and Tools*, 19(1):2–10, June 1998.
27. 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, chap. 5, pp. 97–144. Springer, 1996.
28. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, Sept. 1997.
29. P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, Feb. 1998.