Embedding Processes in a Declarative Programming Language*

– Extended Abstract –

Bernd Braßel[1], Michael Hanus[2], and Frank Steiner[2]

[1] RWTH Aachen, Germany
brassel@halifax.rwth-aachen.de
[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,fst}@informatik.uni-kiel.de

Abstract. While declarative programming languages are based on the idea of specifying the static relationships of problems, the right modeling of the dynamic behavior is equally important for many practical applications. To support a high-level specification of both aspects of computational systems, we propose the embedding of a process-oriented specification language in a multi-paradigm declarative language. Since this embedding is done in a seamless way, the features of the declarative base language can be exploited (1) for a high-level specification of the computational needs in single state transitions of a dynamic system, and (2) to reuse the abstraction facilities of the base language for the specification of the structure of dynamic systems. We show an implementation of these ideas in the declarative multi-paradigm language Curry. This implementation has been used for a prototypical implementation of embedded and distributed systems in a high-level manner.

1 Introduction

Declarative programming languages (e.g., functional, logic, or functional logic languages) aim to support *high-level* descriptions of software systems, which are *executable* at the same time. Such programming languages have many advantages w.r.t. the efficient development, reliability, maintenance, analysis, and verification of programs. The general idea of declarative programming is the specification of the static relationships of a given problem by well-understood mathematical entities (functions and/or predicates). However, many real-world applications demand also for an appropriate modeling of the dynamic behavior of a complex software system, which may be distributed into communicating active parts or embedded in an environment where they must react on external events.

Processes are an appropriate notion to describe dynamic behavior, and high-level formalisms [3] have been developed to describe the essence of process behaviors, like communication, parallelism, process creation, on an abstract level.

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD under the PROCOPE programme.

We want to keep the advantages of declarative programming but make them applicable also for a wider range of applications, like distributed or embedded systems. For this purpose, we propose an embedding of a process-oriented specification language in a declarative language, where we choose the declarative multi-paradigm language Curry [4, 9] as our base. On the one hand, our proposal is based on a clear separation between the declarative and dynamic aspects of an application system by making processes a distinguished data type (this has some similarities to the separation of pure functions and functions manipulating the external world by the introduction of monads [11]). On the other hand, our embedding of processes is done in a seamless way by defining processes as a standard data type so that the features of the base language can be exploited in two ways: (1) the computational needs in single state transitions of a dynamic system can be specified in a high-level style, and (2) the abstraction facilities of the base language can be reused for the specification of the structure of dynamic systems.

Our objective is to provide a domain-specific language for the description of dynamic systems that should react on external events (e.g., embedded systems). The integration of such a domain-specific language into an existing high-level programming language has the advantage that we can reuse the functionality of the base language in application programs and we can provide a prototypical implementation of the entire framework with a limited effort. We have used our implementation for a prototypical implementation of embedded and distributed systems in a high-level manner.

This paper is structured as follows. In the next section we provide a basic introduction into our framework. Section 3 sketches the features of Curry as necessary for the understanding of this paper. In Section 4 we show the modeling of our process-oriented specifications in Curry. Its application is demonstrated in Section 5 by several examples before we make some remarks about the implementation of our framework in Section 6 and conclude in Section 7.

2 Specification of Process Systems

Our framework is based on the partition of a dynamic system into several components that cooperate by exchanging messages. In an embedded system, such a component could correspond to a controller that reacts on messages received from external sensors by sending messages to other active components. In a distributed software system, these components may run on different computers and exchange messages via the Internet. Our goal is to provide a framework for the high-level description of such components.

The structure of each component is sketched in Fig. 1. A *component* consists of a set of processes (p1,p2,...), a global state (i.e., data visible for all processes inside a component but not visible from outside) and a mailbox (queue of messages sent to this component). The behavior of a dynamic system is defined by the behavior of each process. A process can be activated depending on conditions on the global state or the mailbox. If a process is activated (e.g., because a

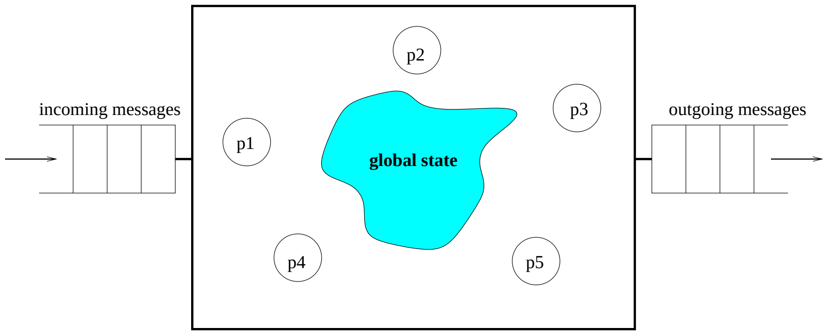


Fig. 1. Component of a dynamic system

particular message arrives in the mailbox), it performs an action and may start other processes (since we have an interleaving semantics, at most one process can perform an action so that actions are atomic entities).

The reaction of a process to the change of its external context (i.e., mailbox or global state) consists of a sequence of *actions*. Currently, possible actions are the change of the global state, the sending of a message to another component,[1] or the removing of a message from the mailbox. (Note that messages are not automatically removed after reading since there may be several processes that must react on the same message.) Of course, the set of possible actions can be extended but our current set is sufficient for our case studies.

The *global state* of a component can be accessed and manipulated by all processes of the same component. Thus, it also serves as a facility for process synchronization. In general, the global state is just a tuple of data items. However, our case studies have shown that it is quite useful to partition the state into a static part with a fixed number of items and a dynamic part with an evolving number of items.[2] Therefore, we provide different actions to manipulate the static or the dynamic part of the state. The static part is changed by defining a new value for it (where changes to single items could be expressed by record updates) ("`SetState` *s*"). For the manipulation of the dynamic part, there are two actions: one for creating a new item ("`NewName` *v ref*"), which can subsequently be accessed via the newly created name *ref*, and one for changing the value associated to a dynamic item ("*ref* `:=` *v*"). Furthermore, there is a

[1] For the sake of simplicity, all outgoing messages are sent via the same channel. If a component wants to send messages to different other components, the messages must be tagged (to identify the receiving component) and it is the purpose of a distributor connecting the different components to forward the outgoing messages to the right receiver. Of course, one can extend our model so that a component can send messages directly to different other components but we have made the experience that our restricted model provides more modularity.

[2] Note that this distinction is not strictly necessary, since the static part can also contain dynamic data structures like lists, but useful to structure components (see the multiple counter example in Section 5).

function `get` to extract the associated value of a dynamic item in a store. Thus, the following table summarizes the current set of actions:

`Send` m	send message m
`SetState` s	set static state to s
ref `:=` v	set dynamic state object *ref* to value v
`NewName` v *ref*	create new dynamic state object *ref* with initial value v
`Deq` m	remove message m from mailbox

As described above, *processes* are activated, depending on a particular condition on the mailbox and/or global state, and perform an action followed by the creation of new processes. Thus, the behavior of each process is specified by

- a *guard* (i.e., a condition on the mailbox and/or state),
- a sequence of *actions* (to be performed when the guard is satisfied and the process is selected for execution), and
- a *process term* describing the further activities after executing the actions.

In order to structure dynamic system specifications in an appropriate manner, we allow *parameterized processes* since this supports the distinction between *local and global state*: process parameters are only accessible inside a process and, therefore, they correspond to the local state of a process, whereas the global state is visible to all processes inside a component. Changes to the local state can simply be obtained by recursive process calls with new arguments. Thus, the language of *process terms* p is very similar to process algebra [3] and defined by the following grammar:

$$\begin{array}{rll}
p ::= & \texttt{Terminate} & \text{successful termination} \\
\mid & \texttt{[}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{]} & \text{sequence of actions} \\
\mid & \texttt{p}\ t_1 \ldots t_n & \text{run process } \texttt{p} \text{ with parameters } t_1 \ldots t_n \\
\mid & p_1\ \texttt{>>>}\ p_2 & \text{sequential composition} \\
\mid & p_1\ \texttt{<|>}\ p_2 & \text{parallel composition} \\
\mid & p_1\ \texttt{<+>}\ p_2 & \text{nondeterministic choice} \\
\mid & p_1\ \texttt{<\%>}\ p_2 & \text{nondeterministic choice with priority} \\
\mid & p_1\ \texttt{<\~{}>}\ p_2 & \text{parallel composition with priority}
\end{array}$$

A sequence of actions is executed from left to right as one atomic operation (having a sequence of actions instead of one single action is useful to specify larger critical regions in many applications, e.g., see the dining philosophers example below). The operators "`>>>`", "`<|>`", and "`<+>`" are standard in process algebra, whereas the last two operators are not very common but useful in applications where a simple nondeterministic choice is not appropriate. The meaning of "p_1 `<%>` p_2" is: "If process p_1 can be executed, execute p_1 (and remove p_2), otherwise execute process p_2 (and remove p_1), if possible." The meaning of "p_1 `<~>` p_2" is: "Execute processes p_1 and p_2 in parallel (like "p_1 `<|>` p_2") but p_2 is executed only if p_1 cannot be executed; if p_1 terminates, then also p_2 terminates." The latter combinator is useful for idle background processes like concurrent garbage collectors.

Using this language of process terms, the behavior of a parameterized process is defined by a *process abstraction* of the following form:

$$\begin{array}{ll} \texttt{p}\ x_1 \ldots x_n & \mid\ guard_1\ \texttt{=}\ actions_1\ \texttt{>>>}\ p_1 \\ & \vdots \\ & \mid\ guard_n\ \texttt{=}\ actions_n\ \texttt{>>>}\ p_n \end{array}$$

where $guard_i$ is a condition on the mailbox, the global state and the process parameters, $actions_i$ is a sequence of actions, and p_i is a process term ($i = 1, \ldots, n$). The different guards together with their right-hand sides are considered to be combined with the "`<%>`" operator, i.e., the first alternative with a valid guard is selected for execution.

As a first example for the use of our framework, consider the classical "dining philosophers". The global state in this example has only a static component, namely the list (or array) of `forks` where each fork has either the value `Avail` ("available") or `Used`. The entire component consists of processes `Thinking` or `Eating` that are parameterized by the number of the philosopher. Then the complete specification is as follows (`forks[`i`]` denotes the value of the i-th fork and `forks[`i`<-`v`]` denotes a new state identical to `forks` but with the value v for the i-th component):

```
Thinking i | forks[i]==Avail ∧ forks[i+1 mod n]==Avail
           = [SetState forks[i<-Used, i+1 mod n <- Used]]
             >>> Eating i

Eating i   = [SetState forks[i<-Avail, i+1 mod n <- Avail]]
             >>> Thinking i
```

Thus, if philosopher i is thinking (which corresponds to the existence of a process term "`Thinking` i") and both forks are available, then he can use both forks to turn into the `Eating` process. Note that the change of the global state, i.e., the use of both forks, can only be performed (in an atomic manner) if both forks are really available. Therefore, the classical deadlock situation is avoided without low-level synchronization (e.g., semaphores) or additional constructions (e.g., room tickets).

An example where a global state with a dynamic part becomes important will be shown later. Next we will show how this specification language can be embedded into the declarative multi-paradigm language Curry in order to obtain an executable specification language for modeling dynamic systems. Before doing so, we review the basic elements of Curry.

3 Curry

In this section we survey the elements of Curry which are necessary to understand the design and implementation of our language for specifying processes. More details about Curry's computation model and a complete description of all language features can be found in [4, 9].

Curry is a modern multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program[3] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of functions and the data types on which the functions operate. Functions are evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation annotations of functions which can be either *flexible* or *rigid*. Calls to rigid functions are suspended if a demanded argument, i.e., an argument whose value is necessary to decide the applicability of a rule, is uninstantiated ("*residuation*"). Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments to the required values in order to apply a rule ("*narrowing*").

Example 1. The following Curry program defines the data types of Boolean values and polymorphic lists (first two lines) and a function to compute the concatenation of two lists:

```
data Bool   = True | False
data List a = []   | a : List a

conc :: [a] -> [a] -> [a]
conc eval flex

conc []     ys = ys
conc (x:xs) ys = x : conc xs ys
```

The data type declarations introduce `True` and `False` as constants of type `Bool` and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[4] Since `conc` is explicitly defined as flexible[5] (by "`eval flex`"), an equation "`conc ys [x] =:= xs`" can be solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., for a given `xs`, the only solution to this equation satisfies that `x` is the last element of `xs`.

[3] Curry has a Haskell-like syntax [10], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

[4] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[5] As a default, all functions except for constraints are rigid.

In general, functions are defined by (*conditional*) *rules* of the form "l | c = e" where l has the form $f\, t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c (which can be omitted) is a constraint, and e is a well-formed *expression* which may also contain function calls, lambda abstractions etc. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable. A *constraint* is any expression of the built-in type `Success`. Each Curry system provides at least equational constraints of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms (i.e., terms without defined function symbols). In contrast, e_1 `==` e_2 denotes an *equality test* which is successful only if both sides e_1 and e_2 are reducible to identical *ground* data terms, i.e., the test suspends in the presence of free variables.

The operational semantics of Curry, precisely described in [4, 9], is based on an optimal evaluation strategy [1] and can be considered as a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Concurrent programming is supported by a concurrent conjunction operator "`&`" on constraints, i.e., a non-primitive constraint of the form "c_1 `&` c_2" is evaluated by solving both constraints c_1 and c_2 concurrently. Furthermore, distributed programming is supported by ports [5] which allows the sending of arbitrary data terms (also including logic variables) between different computation units possibly running on different machines connected via the Internet. The port concept has been used to integrate object-oriented features into Curry [8] and for high-level GUI (Graphical User Interface) programming in Curry [6]. Furthermore, it is relevant for the work described in this paper since the different components of a dynamic system communicate via ports (which is, however, not directly visible to the programmer).

4 Specification of Process Systems in Curry

Now we are ready to define an implementation of process-oriented specifications, as introduced in Section 2, in Curry. The implementation is guided by the motivation to enable the writing of specifications in the high-level style of Section 2. The main difference (and advantage!) is the fact that Curry is a typed language (so that we have a type checker for specifications for free) and allows definitions by pattern matching.

First, we introduce the languages of actions and process terms as data types in Curry. The following data type declaration defines the possible actions. `ObjRef` is an abstract data type denoting references to dynamic objects, and *inmsg*, *outmsg*, *static*, and *dyn* are type variables denoting the type of incoming messages, outgoing messages, the static part and the dynamic items of the global state in a concrete specifcation.

```
data Action inmsg outmsg static dyn =
           Send outmsg             -- send message
         | SetState static         -- set static state
```

```
          | Assign ObjRef dyn      -- set dynamic state object
          | NewName dyn ObjRef     -- create new dynamic object
          | Deq inmsg              -- remove message from mailbox
```

In order to support the same notation as in Section 2, we define the following function (infix operator) as a synonym for the Assign action:

```
ref := cont  =  Assign ref cont
```

The data type of process terms has a similar definition but with the type *proc* of concrete processes as an additional type parameter:

```
data ProcExp proc inmsg outmsg static dyn =
                  Terminate
                | Atomic [Action inmsg outmsg static dyn]
                | Proc proc
                | ParProc   (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | SeqProc   (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | ChProc    (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | ChPriProc (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
                | ParIdle   (ProcExp proc inmsg outmsg static dyn)
                            (ProcExp proc inmsg outmsg static dyn)
```

Again, we support the same notation as in Section 2 by the following operator definitions:

```
p1 >>> p2  =  SeqProc   p1 p2
p1 <|> p2  =  ParProc   p1 p2
p1 <+> p2  =  ChProc    p1 p2
p1 <%> p2  =  ChPriProc p1 p2
p1 <~> p2  =  ParIdle   p1 p2
```

In order to exploit the language features of Curry for the specification of dynamic systems, we consider a *system specification* as a mapping which assigns to each process, mailbox (list of incoming messages), static and dynamic state (list of dynamic objects) a process term (similarly to Haskell, a `type` definition introduces a type synonym in Curry):

```
type Specification proc inmsg outmsg static dyn =
              proc -> [inmsg] -> static -> [DynObj dyn]
                   -> ProcExp proc inmsg outmsg static dyn
```

This definition has the advantage that one can use standard function definitions by pattern matching for the specification of systems, i.e., one can define the behavior of processes in the following form:

```
spec (p x1...xn) mailbox state refs
                 | < condition on x1,...,xn, mailbox, state, refs >
```

```
                  = Atomic [actions] >>> process term
```

Hence, the guard is just a standard constraint on the parameters $x_1, \ldots, x_n$, `mailbox`, `state`, and `refs` so that we need no global variables or auxiliary constructs to access the current global state or mailbox (note that the access to these entities was left unspecified in Section 2).

As an example we show the complete specification of the dining philosophers of Section 2. It consists of the definition of data types for the values of forks, the philosopher processes and the definition of the specification function `phil_spec` ("`rpl` l i v" replaces the i-th element of the list l by v):

```
data ForkStatus = Avail | Used

data PhiloProc = Eating Int | Thinking Int

n = 5   -- here we have five philosophers

phil_spec (Thinking i) _ forks _
 | forks!!i == Avail && forks!!((i+1)`mod`n) == Avail
 = Atomic [SetState (rpl (rpl forks i Used) ((i+1)`mod`n) Used)]
   >>> Proc (Eating i)

phil_spec (Eating i) _ forks _ =
 Atomic [SetState (rpl (rpl forks i Avail) ((i+1)`mod`n) Avail)]
 >>> Proc (Thinking i)
```

Note that neither the mailbox nor the dynamic part of the state is used in this simple example. Initially, all philosophers are thinking. This can be expressed by a process term where five philosopher processes are combined in parallel:

```
phils = foldr1 (<|>) (map (\i->Proc (Thinking i)) [0..n-1])
```

Note that we can use standard higher-order functions like `foldr1` or `map` to create complex process terms since process terms are first-order objects in our specification language. Hence, the expression `phils` reduces to the term

```
Proc (Thinking 0) <|> ··· <|> Proc (Thinking 4)
```

A system specification is executed by providing

1. a port name for incoming messages
2. a port name for outgoing messages
3. an initial process term
4. a system specification
5. an initial (static) state[6]

This is the purpose of the main function `exec_system` so that we can execute our specification as follows:

```
exec_system "in"  "out"  phils  phil_spec
            (take n (repeat Avail))  -- all forks are available
```

[6] The dynamic part of the state is always empty at the beginning.

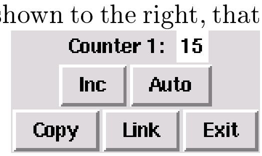
The main advantage of our embedding of a process-oriented language in Curry (rather than defining a complete new specification language) is the reuse of the features of Curry for the specification language, in particular:

- The type checker of Curry can be also used to type check specifications and detect inconsistencies in specifications.
- Functional programming is useful to compute values in actions, process parameters, new states etc.
- Constraint programming is useful for checking complex conditions.
- The standard abstraction facilities of Curry (e.g., higher-order functions) are useful to structure the specification of dynamic systems, in particular, we can define functions to compute process terms (compare `phils` above).

5 Examples

Due to lack of space, we can only sketch two further examples that are implemented using our framework. The first example is a challenge from the Glasgow Research Festival[7], a system of multiple counters.

The application starts be creating a single window, as shown to the right, that visualizes a counter. This counter can be manually (button "Inc") or automatically (periodically) incremented (after pressing the button "Auto"). Pressing the "Copy" button creates a new counter with its own independent state, and pressing the "Link" button creates a new view (counter window) to the same counter.



We implement this system by the specification of a *counter control system* which is responsible to control all counters and organize the communications with the different windows. If the user presses a button in a window *win*, an appropriate message (e.g., `(Inc` *win*`)`, `(Copy` *win*`)`) is sent to the counter controller which must correctly react to this request. The global state of the counter controller has only a dynamic part since a new counter object is created in the state whenever the users presses the "Copy" button. The value of a counter object has the form `(Counter` *val wins mode*`)` where *val* is the current value of the counter, *wins* is a list of windows where this counter is displayed, and *mode* is the increment mode of the counter (`Manual` or `Automatic`). As a consequence, the individual processes of the controller are parameterized with references to counter objects and windows. For instance, there is a process `(Manual_Ctrl c w)` for each counter object `c` and window `w` where `c` is displayed. This process is responsible for processing the messages received from window `w`. Using pattern matching, there is one rule for each message in the specification. For instance, the rule for the message `Inc` is as follows:

```
cctrl (Manual_Ctrl c w) (Inc win:_) _ store | win==w
 = let Counter val windows _ = get c store in
   Atomic [c := Counter (val+1) windows Manual, Deq (Inc win)]
```

7 http://www.cs.chalmers.se/~magnus/GuiFest-95/

```
    >>> Proc (Refresh_Window c) <|> Proc (Manual_Ctrl c w)
```

Thus, the guard consists of checking whether the message comes from the window for which this process is responsible. If this is the case, the value of the counter is incremented (where the increment mode is set to `Manual`), the message is removed from the mailbox, and a new process for refreshing all windows for this counter is created. The latter process sends update messages to all appropriate windows, where we apply some standard higher-order functions:

```
cctrl (Refresh_Window c) _ _ store =
   let Counter val windows _ = get c store in
   foldr (<|>) Terminate
         (map (\w->Atomic [Send (Update w val)]) windows)
```

The remaining cases are similarly defined. In particular, for each counter object `c` there is a process `(Automatic_Ctrl b c)` which is responsible for incrementing counters in automatic mode. This is done by an external clock which sends clock ticks as messages to the controller so that the `Automatic_Ctrl` processes are activated on these messages. Since there may be many of these processes, the clock tick message should not be deleted in the mailbox by any of these processes (since all of them must have the chance to react). This is the purpose of a background process `Delete_Clocks` which is simply defined as (each clock signal has a Boolean flag to distinguish successive signals):

```
cctrl Delete_Clocks (ClockSignal flag:_) _ _ =
   Atomic [Deq (ClockSignal flag)] >>> Proc Delete_Clocks
```

The complete specification of the counter controller, which is omitted due to lack of space, consists of nine rules (in addition to the three rules above, four further rules for handling the counter button messages, one rule for the `Automatic_Ctrl` process and one rule for the `Create_Window` process that creates a window together with a `Manual_Ctrl` process for it) which specify in a readable way the behavior of all processes in the controller. The initial configuration is defined by the following process term (where `c` is a free variable denoting the reference to the first counter object created by `NewName`):

```
Atomic [NewName (Counter 0 [] Manual) c] >>>
((Proc (Create_Window c) <|> Proc (Automatic_Ctrl True c))
 <~> Proc Delete_Clocks)
```

Note that it is important to create the process `Delete_Clocks` as a background process with lowest priority so that the clock signals are deleted only if no other process can be active. The remaining parts of the complete implementation, namely the counter GUIs, are also only a few lines of code thanks to the use of the Curry library for high-level GUI programming [6].

Our second example is a lift control system as visualized in Fig. 2. It consists of a number of request buttons that are inside a lift (left) or outside on the different floors (right), and a lift that can move up and down as well as open and close the doors. Instead of controlling a real lift, we simulate the lift also as a dynamic system. Thus, our implementation consists of two components in the

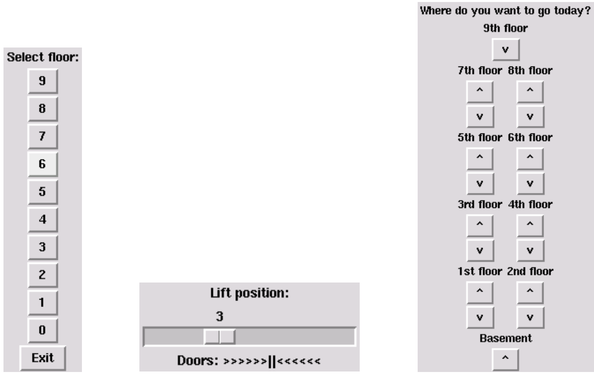


Fig. 2. A lift control system

sense of Fig. 1: a *lift controller* that accepts requests from the buttons, reacts on sensor messages from the lift (e.g., arrival at some floor), and sends appropriate control commands to the lift unit, and a *lift simulator* which simulates the lift by reacting on commands from the lift controller and sending sensor messages to the controller and the GUI (shown in the middle of Fig. 2).

The specification of the entire system can be appropriately expressed in our framework. For instance, the lift controller consists of two processes running in parallel: a process `Sorting` which is responsible to react on user requests by computing a list of floors where the lift should stop (this list is sorted according to the movement of the lift), and a second process for controlling the lift. This process can be either `Moving` or `Stopped` according to the state of the lift unit (e.g., `Moving` waits for sensor messages from the lift unit about the reached floor, and `Stopped` waits for floor requests put in by `Sorting` in the global state).

Due to lack of space, we cannot show further details from this specification, but the complete implementation is available from the authors.

6 Implementation

Our process-oriented specification language is implemented as a standard Curry library so that it can be used in any Curry program. It is freely available as a library for PAKCS (Portland Aachen Kiel Curry System) [7] and completely implemented in Curry, using the features for distributed programming [5] to implement the communication between different components of a system. The current implementation is based on an interpreter for process terms according to the operational semantics of dynamic systems (see also [2]). Although the interpreter approach is not very efficient, it is fast enough to run our examples and required only a limited implementation effort (the complete implementation consists of approximately 200 lines of Curry code, without the imported standard libraries of Curry).

7 Conclusions

We have presented a domain-specific language for process-oriented programming. Since this language is embedded in the declarative multi-paradigm language Curry, we enable process-oriented programming in Curry, which is useful for the implementation of distributed or embedded systems. On the other hand, we can reuse the programming language features of Curry for the high-level specification of dynamic systems. The specification language is based on process algebras and offers parameterized processes and a global store for the exchange of data between processes. Thus, all internal communication (synchronization) between processes is performed via the store, whereas the external communication between different dynamic systems is done by sending messages. Although our language allows high-level specifications as in other process-oriented specification languages, it is *fully executable* at the same time. Therefore, it is a useful tool to implement and test dynamic systems in a prototypical manner. We have shown the appropriateness of our framework by several case studies.

There are many proposals for process-oriented specification languages (for example, see [3]). However, as far as we know, our work is the first fully implemented approach to exploit the high-level features of both functional and logic programming for process-oriented specifications. The most similar proposal to our approach is [2] (which is not accidental since our work is inspired by many discussions with the authors of [2]). [2] contains a "generic" framework for the extension of declarative (functional, logic, functional logic) languages to include processes where there is a strict distinction between the language of processes and the underlying programming language. In particular, declarative programs are considered as the global state between transition steps of processes. Thus, the modification of declarative programs are allowed without restrictions, i.e., arbitrary program clauses can be added or deleted. This complicates the implementation of their framework. Moreover, when modifying values associated to names, the evaluation time becomes important, but this is not clearly specified in their framework. To provide an effective implementation, we have restricted all modifications to a set of well-defined data items (partitioned into a static and dynamic part of the global store). As shown by our case studies, this is sufficient for all examples discussed in [2]. Moreover, we could provide fully executable specifications of all examples, which is due to the use of the Curry libraries for distributed [5] and GUI [6] programming.

For future work we will consider more applications to study the appropriateness of our approach or necessary extensions (like real-time conditions). Furthermore, it would be interesting to consider the translation of our specification language into other existing specification or control languages in order to reuse existing verification or implementation frameworks. This would enable the use of high-level declarative programming techniques in new application fields.

Acknowledgements. The authors are grateful to Rachid Echahed and Wendelin Serwe for fruitful discussions that led to the development described in this paper.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
2. R. Echahed and W. Serwe. Combining Mobile Processes and Declarative Programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pp. 300–314. Springer LNAI 1861, 2000.
3. W. Fokkink. *Introduction to Process Algebra.* Springer, 2000.
4. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
5. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.
6. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
7. M. Hanus, S. Antoy, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2000.
8. M. Hanus, F. Huch, and P. Niederau. An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 89–106. Springer LNCS 2011, 2001.
9. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
10. J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.
11. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.