Electronic Notes in Theoretical Computer Science 76 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume76.html 19 pages

Programming Autonomous Robots in Curry [1]

Michael Hanus Klaus Höppner

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,klh}@informatik.uni-kiel.de

Abstract

In this paper we present a framework to program autonomous robots in the declarative multi-paradigm language Curry. This is an experiment to use high-level declarative programming languages for the programming of embedded systems. Our programming model is based on a recent proposal to integrate a process-oriented specification language in Curry. We show the basic ideas of our framework and demonstrate its application to robot programming.

1 Motivation

Although the advantage of declarative programming languages (e.g., functional, logic, or functional logic languages) for a high-level implementation of software systems is well known, the impact of such languages to many real world applications is quite limited. One reason for this might be the fact that many real-world applications have not only a logical (declarative) component but demand also for an appropriate modeling of the dynamic behavior of a system. For instance, embedded systems become more important applications in our daily life than traditional software systems on general purpose computers, but the reactive nature of such systems seems to make it fairly difficult to use declarative languages for their implementation. We believe that this is only partially true since there are many approaches to extend declarative languages with features for reactive programming. In this paper we try to apply one such approach, the extension of the declarative multi-paradigm language Curry [13,17] with process-oriented features [6,7], to the programming of concrete embedded systems.

The embedded systems we consider in this paper are the Lego Mindstorms

[1] This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2, by the DAAD/NSF under grant INT-9981317, and by the DAAD under the PROCOPE programme.

Fig. 1. The RCX, the "heart" of a Mindstorm robot

robots.[2] Although these are toys intended to introduce children to the construction and programming of robots, they have all the typical characteristics of embedded systems. They act autonomously, i.e., without any connection to a powerful host computer, have a limited amount of memory (32 kilobytes for operating system and application programs) and a specialized processor (Hitachi H8 16 MHz 8-bit microcontroller) which is not powerful compared to current general purpose computers. In order to understand the examples in this paper, we shortly survey the structure of these robots.

The Robotics Invention System (RIS) is a kit to build various kinds of robots. The heart of the RIS kit is the Robotic Command Explorer (RCX, see Fig. 1) containing a microprocessor, ROM, RAM, connections to sensors and actuators, etc. To react to the external world, the RCX contains three input ports to which various kinds of sensors (e.g., touch, light, temperature, rotation) can be connected. To influence the external world, the RCX has three output ports for connecting actuators (e.g., motors, lamps), a simple speaker for playing sounds, and a small LCD display. Furthermore, it has an infrared (IR) interface for communicating with a host computer (e.g., for downloading programs) or with other RCX bricks. Since the RCX has no keyboard (except for four control buttons) and only a small one-line display, programs for the RCX are usually developed on standard host computers (PCs, workstations), cross-compiled into code for the RCX and then transmitted to the RCX via the IR interface.

The RIS is distributed with a simple visual programming language (RCX code) to simplify program development for children. This programming language is based on colored bricks that are put together in order to yield the control program for the RCX. The different kinds of bricks include commands (like *actuator on/off*, *wait*, *set motor direction*, *set power*, etc), sensor watchers (code blocks executed in case of sensor events), control and macro blocks. In comparison to traditional programming languages, the language has interesting extensions (multi-threading, sensor and actuator control, delay and timeout primitives). However, the language is also quite limited at the same time: no concept of variables (only a simple counter), no expressions, no parameterized functions, no arbitrarily nested control structures, no synchronization with protected resources etc. Therefore, various attempts have been made

[2] `http://mindstorms.lego.com/` Note that these names are registered trademarks although we do not put trademark symbols at every occurrence of them.

to replace the standard program development environment by more advanced systems. On the one hand, one can find more advanced visual languages (e.g., Robolab [3]). On the other hand, there are also imperative languages (e.g., "Not Quite C" [4]) or extensions of existing programming languages with compilers for the RCX. A popular representative of the latter kind is based on replacing the default Lego RCX firmware by a new operating system, *legOS*, [5] and writing programs in C with specific libraries and a variant of the compiler `gcc` with a special back end for the RCX controller. The resulting programs are quite efficient (machine code instead of byte code) and provide full access to the RCX's capabilities.

In this paper we will use a declarative multi-paradigm programming language (Curry) with synchronization and process-oriented features to program the RCX. The language Curry [13,17] can be considered as a general purpose declarative programming language since it combines in a seamless way functional, logic, constraint, and concurrent programming paradigms. In order to use it also for reactive programming tasks, different extensions have been proposed. [14] contains a proposal to extend Curry with a concept of ports (similar concepts exist also for other languages, like Erlang [2], Oz [21], etc) in order to support the high-level implementation of distributed systems. These ideas have been applied in [6] to implement a domain-specific language for process-oriented programming, inspired by the proposal in [7] to combine processes with declarative programming. The target of the latter is the application of Curry for the implementation of reactive and embedded systems. The example applications shown in [6] are simulators of artificial systems, e.g., a lift controller. In this paper we will apply this framework to a real embedded system: the Mindstorms robots described above.

This paper is structured as follows. In the next section we sketch the features of Curry as necessary for the understanding of this paper. Section 3 surveys the framework for process-oriented programming in Curry. We apply this framework to the programming of autonomous robots in Section 4 and show in Section 5 concrete programming examples before we make some remarks about the current implementation of our framework in Section 6 and conclude in Section 7 with a discussion of related work. The appendix contains the definition of the operational semantics of the framework for process-oriented programming in Curry.

2 Curry

In this section we survey the elements of Curry which are necessary to understand the examples in this paper. More details about Curry's computation

[3] `http://www.lego.com/dacta/robolab`
[4] `http://www.enteract.com/~dbaum/nqc/`
[5] `http://www.legos.sourceforge.net/`

model and a complete description of all language features can be found in [13,17].

Curry is a 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 [6] 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 flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule ("*narrowing*"). Calls to rigid functions (e.g., external functions like arithmetic operators [5] or functions implementing concurrent objects [16]) are suspended if a demanded argument is uninstantiated ("*residuation*").

Example 2.1 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 []     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. [7] Due to the logic programming features of Curry, 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.,

[6] Curry has a Haskell-like syntax [19], 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 ("$f\ e$").

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

for a given `xs`, the only solution to this equation satisfies that `x` is the last element of `xs`.

Functions are generally defined by (*conditional*) *rules* of the form "$f\,t_1 \ldots t_n$ `|` c `=` e" where f is a function, $t_1, \ldots, t_n$ are data terms, each variable occurs only once on the left-hand side, the *condition* c (which can be omitted) is a constraint (i.e., an expression of the built-in type `Success`), 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.

The operational semantics of Curry, described in detail in [13,17], 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 and the initial goal) and (concurrent) logic programming. Concurrent programming is supported by a concurrent conjunction operator "`&`" on constraints, i.e., a 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 [14] 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.

3 Specification of Process Systems

In this section we review the framework for process-oriented programming in Curry as originally proposed in [6]. Here we will present a slightly modified and improved version. The application of this framework to the programming of autonomous robots will be discussed in the next section.

The motivation for the process-oriented extension of Curry is the fact that purely declarative languages are often not adequate for the modeling and programming of systems where the dynamic (reactive) behavior is important, like embedded systems. For this purpose, a process-oriented language is proposed which is embedded into Curry by describing processes as expressions of a distinct type.

In this framework, a *process system* 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), see Fig. 2.[8] For instance, an embedded control system corresponds to a process system that reacts on messages received from external sensors by sending messages to the actuators. The behavior of a process system is defined by the behavior of each process. A process can be activated depending on conditions on the global state and the mailbox. If a process is activated (e.g., because a particular message arrives in the mailbox), it

[8] In the original framework [7], such a process system is a component of a dynamic system which consists of several components that cooperate by exchanging messages, see also [8].

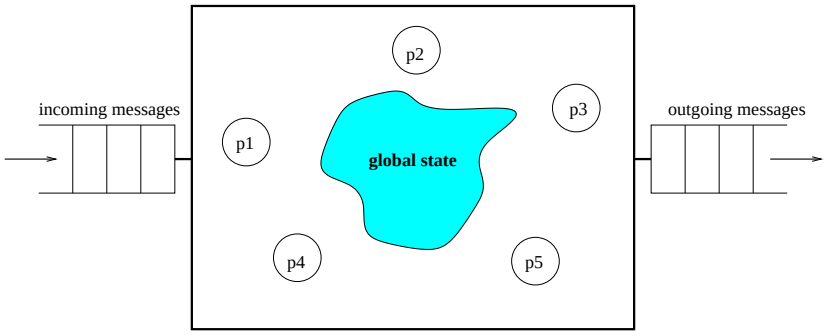

Fig. 2. Component of a dynamic system

performs actions and may start other processes (since systems are based on an interleaving semantics, at most one process can perform actions so that actions are atomic entities). Similarly to Erlang [2], the mailbox is a finite list of all currently available messages. Since processes can access the complete mailbox (and not only the first message), it is fairly easy to implement "alarm" processes that immediately react on important messages contained in the mailbox at any position.

The reaction of a process to the change of its external context (i.e., mailbox or global state) consists of a sequence of *actions*. Possible actions are the change of the global state to a value s ("`Set` s"), the sending of a message m ("`Send` m"),[9] and the removing of a message m from the mailbox ("`Deq` m").[10]

The *global state* of a component can be accessed and manipulated by all processes of this component. Thus, it also serves as a facility for process synchronization. In general, the global state is just a tuple of data items. Since these items can be of arbitrary type, they can also store dynamically evolving data structures.

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

- a *condition* (on the mailbox and state),
- a sequence of *actions* (to be performed when the condition 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

[9] For the sake of simplicity, all outgoing messages are sent via the same channel. This is sufficient for embedded systems where the messages can be interpreted as commands to control the connected actuators.

[10] Note that messages are not automatically removed after reading since there may be several processes that must react on the same message.

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 achieved by recursive process calls with new arguments. Thus, the language of *process terms* p is very similar to process algebra [9] and defined by the following grammar:

p	::=	`Terminate`	successful termination
	|	`Proc (p` $t_1 \ldots t_n$`)`	run process `p` with parameters $t_1 \ldots t_n$
	|	p_1 `>>>` p_2	sequential composition
	|	p_1 `<|>` p_2	parallel composition
	|	p_1 `<+>` p_2	nondeterministic choice
	|	p_1 `<%>` p_2	nondeterministic choice with priority
	|	p_1 `<~>` p_2	parallel composition with priority

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. For instance, we use it in some of our applications for a background process that removes unused sensor messages from the mailbox. A detailed specification of the operational semantics of process terms can be found in the appendix.

In order to specify processes in Curry following the ideas above, there are data types to define the structure of actions and processes. The following data type declaration represents the possible actions, where *inmsg*, *outmsg*, and *state* are type variables denoting the type of incoming messages, outgoing messages, and the global state of a concrete application, respectively.

```
data Action inmsg outmsg state =
          Send outmsg          -- send message
        | Set  state           -- set global state
        | Deq  inmsg           -- remove message from mailbox
```

A similar definition exists for the type "`ProcExp` *inmsg outmsg state*" defining the language of process terms as above. Then the process combinators (e.g., `>>>`, `<|>`) are operations on this data type. Furthermore, we define a *guarded process* as a pair of a list of actions and a process term:

```
data GuardedProc inmsg outmsg state =
```

```
                GuardedProc [Action inmsg outmsg state]
                            (ProcExp inmsg outmsg state)
```

For the sake of readability, we define an infix operator to construct guarded processes:

```
acts |> pexp = GuardedProc acts pexp
```

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

```
type Process inmsg outmsg state =
             [inmsg] -> state -> GuardedProc inmsg outmsg state
```

This definition has the advantage that one can use standard function definitions by pattern matching for the specification of processes, i.e., one can define the behavior of a process `p` with parameters $x_1, \ldots, x_n$ in the following form:

```
p x1...xn mailbox state
                | < condition on x1,...,xn, mailbox, state >
                = [actions] |> process term
```

Hence, the condition is just a (decidable) constraint on the parameters $x_1, \ldots, x_n$, `mailbox`, and `state` so that we need no global variables or auxiliary constructs to access the current global state and mailbox. A process can run in a current system state if its condition is satisfied. In this case, its 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) and the process is replaced by the new process term following the list of actions. If a process is specified with several rules or conditions, these rules can be considered as combined with the "`<%>`" operator, i.e., the first alternative with a valid condition is selected for executing this process.

As an example, we show a specification of the classical dining philosophers example. The global state in this example is a list of `forks` where each fork has either the value `Avail` ("available") or `Used`. The entire system consists of processes `thinking` and `eating` that are parameterized by the number of the philosopher. The behavior of these processes is determined as follows ("l `!!` i" denotes the i-th element of list l and "`rpl` l i v" denotes the result of replacing the i-th element of the list l by v):

```
data ForkStatus = Avail | Used

n = 5   -- here we have five philosophers

thinking :: Int -> Process _ _ [ForkStatus]
```

```
thinking i _ forks
 | forks!!i == Avail && forks!!((i+1)`mod`n) == Avail
 = [Set (rpl (rpl forks i Used) ((i+1)`mod`n) Used)]
   |> Proc (eating i)

eating :: Int -> Process _ _ [ForkStatus]
eating i _ forks
 = [Set (rpl (rpl forks i Avail) ((i+1)`mod`n) Avail)]
   |> Proc (thinking i)
```

The mailbox parameter is not used in this simple example. Thus, we use the anonymous type variable "`_`" for the type of incoming and outgoing messages. Initially, all philosophers are thinking, which corresponds to the initial process term

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

which can be simply defined by

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

exploiting the higher-order features of Curry. Furthermore, all forks are available, which is expressed by the initial state

```
take n (repeat Avail)
```

(which evaluates to a list of length `n` and elements `Avail`). The above specification describes the following behavior. If philosopher i is thinking, which corresponds to the existence of a process `(thinking` i`)`, and both forks are available, then he can use both forks and 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. This is due to the fact that the successful check of the condition and the execution of the sequence of actions is one atomic unit which cannot be interrupted by other processes (see also [6]). Therefore, the classical deadlock situation is avoided without low-level synchronization (e.g., semaphores) or additional constructions (e.g., room tickets).

An advantage of this embedding of a process-oriented language into Curry is the reuse of the abstraction facilities of Curry for process definitions. For instance, we can extend the language of process terms by an "`atomic`" process (i.e., a sequence of actions executed as one atomic operation) by the following definition:

```
atomic :: [Action inmsg outmsg state]
                                    -> ProcExp inmsg outmsg state
atomic actions = Proc (\_ _ -> actions |> Terminate)
```

In the next section we will apply this framework to the programming of au-

tonomous robots described in Section 1.

4 Programming Autonomous Robots

A difficulty in the programming of reactive systems is the description how and when they should react to the external environment measured by the available sensors. Synchronous languages [10] are one possible programming model for this purpose since one can describe with them the immediate reaction to sensor values. On the other hand, asynchronous programming styles can be easier integrated in general purpose languages. In order to use our asynchronous programming model as described in Section 3 for the programming of robots, we propose the separation of the entire programming task into two parts. The description of the actions to be executed in reaction to some sensor events will be described in an asynchronous manner as a process system where we assume that the sensor sends messages whenever some relevant value is measured. The description of these "relevant events" will be specified in a synchronous component which always controls the sensor inputs and sends relevant values as messages to the process system. For instance, for a mobile robot that tries to avoid obstacles, the only relevant events are signals from the touch sensors in order to register the bumping against an obstacle. A mobile robot that tries to follow a black line must only react to the change (of a light sensor) between dark and bright light values. In order to measure time intervals (e.g., time outs, waiting), clock events become relevant. We do not describe the implementation of this synchronous component (compare for instance [4] for the combination of a functional logic language with features for synchronous programming) but assume in the following that such sensor events are sent as messages to the process system which reacts to them with appropriate actions.

As mentioned in Section 1, there are three output ports to connect actuators to the RCX. We describe these ports by

```
data OutPort = Out_A | Out_B | Out_C
```

The standard actuators are motors and lamps connected to these ports. The control of these actuators can be described by the following messages which are sent by the process system to the robot:

```
data RobotCmd = MotorDir   OutPort MotorDirection
              | MotorSpeed OutPort Int
              | Lamp       OutPort Bool -- True=on / False=off

data MotorDirection = Fwd | Rev | Off
```

These messages will change the state of the actuator. For instance, if a motor is turning forward, it will be turning backwards after receiving the message `(MotorDir` *port* `Rev)`. A motor is turned off if it receives the message `(MotorDir` *port* `Off)`.

Fig. 3. Example of an obstacle avoiding robot

As discussed above, we assume that the robot is informed by a synchronous component about sensor events. Therefore, we cannot specify the possible events in general but these depend on the connected sensors, application relevant sensor values etc. We only assume that there always exists a process `(wait` n`)` which terminates n milliseconds after its first activation (in principle, this can be described by sending and waiting for appropriate messages from the synchronous subsystem).

In the following section we show concrete examples to apply this framework in practice.

5 Examples

As a first example, we want to implement an autonomous robot that moves on the ground and tries to avoid obstacles that it detects with two touch sensors mounted at the left and right front of the robot. Fig. 3 shows an example of such a robot, which we call "rover" in the following. We assume that the following messages are sent from the sensors to the robot control system whenever the rover touches an obstacle with the left or right sensor:

```
data TouchEvent = TouchLeft | TouchRight
```

The control system of the rover contains the processes `go`, `waitEvent`, and `turn`. The initial process `go` just starts the rover by setting both motors (for the left and right wheel connected at ports `Out_A` and `Out_C`, respectively) into forward direction (which also starts their engines) and then waits for events from the touch sensors:

```
go _ _ =
    [Send (MotorDir Out_A Fwd), Send (MotorDir Out_C Fwd)]
    |> Proc waitEvent
```

The `waitEvent` process is activated on an event from one of the touch sensors. It reacts by driving back for 2 seconds and turning the rover by setting one of the motors into forward direction followed by the initial process state `go`:

```
waitEvent (touchmsg:_) _ =
   [Deq touchmsg]  |> Proc (turn touchmsg)
```

```
turn touch _ _ =
   [Send (MotorDir Out_A Rev), Send (MotorDir Out_C Rev)] |>
   Proc (wait 500) >>>
   atomic [Send (MotorDir
          (if touch==TouchLeft then Out_A else Out_C) Fwd)] >>>
   Proc (wait 500) >>>
   Proc go
```

The second example is a "line follower" rover, i.e., a robot similar to the previous one but with a light sensor measuring the light on the ground instead of the touch sensors. The task is to locate a black line and then follow it. We assume that the synchronous sensor subsystem sends messages of the form `(Light` i`)` to the control system in regular intervals indicating the current value i measured by the light sensor (darkness corresponds to small and brightness to bigger values). The constant `dark_thresh` and `bright_thresh` are the threshold values separating a black line from a bright ground. Then the control system can be specified by the following processes (first, the rover tries to locate a line by moving to a dark region and then it turns whenever it leaves the dark line):

```
locateLine _ _ =
    [Send (MotorDir Out_A Fwd), Send (MotorDir Out_C Fwd)] |>
    Proc waitDark

waitDark (Light i:_) _ =
    [Deq (Light i)] |>
    Proc (if i<dark_thresh then go else waitDark)

go _ _ =
    [Send (MotorDir Out_A Fwd), Send (MotorDir Out_C Fwd)] |>
    Proc waitBright

waitBright (Light i:_) _ =
   [Deq (Light i)] |>
   Proc (if i>bright_thresh then turn else waitBright)

turn _ _ = [Send (MotorDir Out_A Rev)] |> Proc waitDark
```

These simple examples show only the basic features of our framework to program autonomous robots. They do neither show the use of several parallel processes nor the use of the global state to synchronize them (but we have also implemented a small production line with a conveyor belt and a robot arm where these features are important). Nevertheless, it should be clear from the description in Section 3 how to use these features to model more complex robot control systems (e.g., including planning capabilities, parallel control of several sensors).

6 Implementation

Our current implementation does not yet include a full compiler to translate the Curry specifications into binary code that can run on the RCX (this is under implementation so that our current examples are manually translated). In order to test the behavior of programs before translating and running them on the RCX, we have implemented a simulator in Curry for the framework described above.

Basically, the simulator is an interpreter for the process expressions following the operational semantics of the process algebra [6,7] (see also the appendix). In order to run a typical robot program as shown in Section 5, the program needs some input from the sensors and has to show the messages sent to the actuators. For this purpose, the simulator also contains two further components, a virtual sensor suite and a command log for logging the messages sent to the actuators.

The virtual sensor suite is a process that generates sensor messages for the robot program. This process controls a graphical user interface (GUI) with buttons and sliders that represent the sensors of the robot. The user can simulate sensor inputs for the robot through this GUI and can check how the system will react. Since the only externally observable reactions are messages sent to the actuators, there is another process, the command log, for showing all these messages. This process simply waits for messages sent to the actuators and prints them with a time stamp in a terminal window.

Our current simulator is very simple so that, for every new robot with different sensors, one has to design a new virtual sensor suite with the appropriate buttons and sliders (which corresponds to the implementation of the synchronous component for controlling sensor events). This task is fairly easy thanks to the use of the Curry library `Tk` for high-level GUI programming [15]. Nevertheless, one could also write a function that generates such a virtual sensor suite from a specification of the sensors connected to the input ports. In a similar way, one could also improve the purely text-based command log by adding a graphical representation of actuators showing the current state of them (e.g., a symbol for a motor that shows if it is spinning forward, backward, or off). Such a representation could be also generated from a description of the type of the connected actuators.

Our final goal is the compilation of the Curry programs into code for the RCX. For doing so, one has to complete the descriptions shown in Section 5 by a specification of the synchronous component for controlling the sensors. We plan to compile these descriptions into C code that can be further compiled into RCX code by the legOS compiler mentioned in Section 1. Due to the (speed and time) limitations of the RCX, a simple approach, like porting a Curry implementation to the RCX, will not work (this is in contrast to [18] where a functional robot control language is proposed which is executed on top of Haskell running on a powerful Linux system). In particular, the interpreta-

tion of the process expressions on the RCX causes too much overhead so that a direct compilation of the processes into more primitive code is necessary. Fortunately, legOS is a POSIX-like operating system offering an appropriate infrastructure like multi-threading, semaphores for synchronization etc. Nevertheless, many optimizations are required to map the operational semantics of Curry into the features available in legOS. We plan to start with a restricted subset of Curry, which is sufficient for our current applications and can be translated in a simple way, and extend it together with appropriate optimization tools. In this way, we will investigate general principles to compile high-level languages into specialized systems with limited capabilities.

7 Conclusions and Related Work

We have presented a framework to program autonomous robots with a declarative language extended by a process concept. For this purpose, we have proposed a domain-specific language for process-oriented programming. This language is based on process algebras and offers parameterized processes (with priorities) and a global store for the synchronization and exchange of data between processes. Processes can be activated depending on the arrival of particular messages and also on the occurrence of values in the global store (set by other processes). Since this language is embedded in the declarative multi-paradigm language Curry, we can use the high-level features of declarative programming for the implementation of embedded systems. A prototypical implementation has been performed with a simulator. The full implementation by compiling into directly executable code is currently developed.

Some work related to high-level languages for programming embedded or process-oriented systems has already been mentioned above. For embedded system programming, synchronous languages like Esterel [3] or Lustre [11] are often used. Thus, one can also apply such languages to program embedded systems like the Lego Mindstorms robots. Actually, there already exist compilers for those languages into C [11] so that one can use the legOS compiler to produce RCX code. The translation of the synchronous languages normally produces sequential code by synthesizing the control structure of the object code in the form of an extended finite automaton [12]. This is a major drawback since one does not have much control on the size of the generated C program. In some cases only slight modifications in a robot specification can result in a big increase in the size of the generated code. Another drawback is the state explosion for large programs which could be a problem due to the limited amount of memory in the Mindstorms robots.

Some future work has already been mentioned in Section 6. First, we will develop a language to specify the synchronous component which controls the sensors and informs the process system by sending messages. Then, we

[11] `http://www.emn.fr/x-info/lego/`

will investigate compilation techniques for producing efficient executable code. Another interesting topic is the development of a graphical tool to specify the process structure with visual elements where the corresponding Curry code is automatically generated. Finally, it would be also interesting to describe the behavior of several robots in one system and generate the code for the individual robots and their communication automatically. This would enable the implementation of more complex systems with many sensors and actuators.

Acknowledgements. The authors are grateful to Frank Huch for fruitful discussions and suggestions to improve 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.

[2] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, Vol. 19, No. 2, pp. 87–152, 1992.

[4] J. Blanc and R. Echahed. Adding Time to Functional Logic Programs. In *Proc. of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pp. 31–44. Report No. 2017, University of Kiel, 2001.

[5] S. Bonnier and J. Maluszynski. Towards a Clean Amalgamation of Logic Programs with External Procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 311–326. MIT Press, 1988.

[6] B. Braßel, M. Hanus, and F. Steiner. Embedding Processes in a Declarative Programming Language. In *Proc. Workshop on Programming Languages and Foundations of Programming*, pp. 61–73. Aachener Informatik Berichte Nr. AIB-2001-11, RWTH Aachen, 2001.

[7] 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.

[8] R. Echahed and W. Serwe. A Component-Based Approach to Concurrent Declarative Programming. In *Proc. of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pp. 285–298. Report No. 2017, University of Kiel, 2001.

[9] W. Fokkink. *Introduction to Process Algebra*. Springer, 2000.

[10] N. Halbwachs. Synchronous programming of reactive systems. In *Tenth International Conference on Computer-Aided Verification (CAV'98)*, pp. 1–16. Springer LNCS 1427, 1998.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1305–1320, 1991.

[12] N. Halbwachs, P. Raymond, and C. Ratel. Generating Efficient Code From Data-Flow Programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, pp. 207–218. Springer LNCS 528, 1991.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.

[18] J. Peterson, P. Hudak, and C. Elliott. Lambda in Motion: Controlling Robots With Haskell. In *Proc. First International Workshop on Practical Aspects of Declarative Languages*, pp. 91–105. Springer LNCS 1551, 1999.

[19] S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. `http://www.haskell.org`, 1999.

[20] A. Podelski and G. Smolka. Operational Semantics of Constraint Logic Programs with Coroutining. In *Proc. of the Twelfth International Conference on Logic Programming (ICLP'95)*, pp. 449–463. MIT Press, 1995.

[21] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pp. 324–343. Springer LNCS 1000, 1995.

A Operational Semantics

In this section we define the operational behavior of dynamic systems. To simplify the presentation, we only describe the behavior of a single component of a dynamic system. One can derive the operational behavior of a complete system, consisting of several components, from the transition relation of the single components extended by the exchange of messages between the message queues of the components.

We assume that $\mathcal{S}$ is a system specification, i.e., a set of process abstractions as introduced in Section 3. We describe the execution of a component by a transition relation $\langle S, p\rangle \to \langle S', p'\rangle$ where p, p' are process terms and S, S' are component states. A *component state* consists of the global state and the lists of incoming and outgoing messages. Each action manipulates the component state, i.e., an action is a mapping between component states. For the sake of simplicity, we do not specify the precise meaning of actions but denote by $a(S)$ the new component state obtained by applying action a to component state S.

The transition relation for components is defined by the set of inference rules in Fig. A.1. To simplify the definition of the transition relation, we separate the basic transition steps and the simplification of process terms by introducing a congruence relation on process terms, similarly to [20]. This avoids the introduction of specific transition rules for process simplification which would have no operational effect. For this purpose, we define $\equiv$ as the smallest congruence relation satisfying the following laws: [12]

`Terminate >>>` p	$\equiv$	p			
`Terminate <|>` p	$\equiv$	p	p `<|> Terminate`	$\equiv$	p
`Terminate <+>` p	$\equiv$	`Terminate`	p `<+> Terminate`	$\equiv$	`Terminate`
`Terminate <%>` p	$\equiv$	`Terminate`	p `<%> Terminate`	$\equiv$	`Terminate`
`Terminate <~>` p	$\equiv$	`Terminate`	p `<~> Terminate`	$\equiv$	p

We assume that all inference rules are interpreted w.r.t. the congruence $\equiv$. Formally, this can be expressed by extending the basic transition relation $\to$ defined in Fig. A.1 to a transition relation modulo congruent process terms "$\Rightarrow$" defined as follows:

$$\frac{p_1 \equiv p_1' \quad p_2 \equiv p_2' \quad \langle S, p_1'\rangle \to \langle S', p_2'\rangle}{\langle S, p_1\rangle \Rightarrow \langle S', p_2\rangle}$$

The initial component state and process term is specified by the programmer (see Section 3), and the final configurations are those with a process term

[12] Actually, the interpretation of these laws as rewrite rules yields a confluent and terminating rewrite system so that the congruence relation is easily decidable by term rewriting.

$$\text{R1}\ \frac{}{\langle S, \texttt{Proc (p}\ t_1 \ldots t_n\texttt{)}\rangle \to \langle a_n(\cdots(a_1(S))\cdots), \sigma(p_i)\rangle}$$

if $\begin{array}{ll}\texttt{p}\ x_1 \ldots x_n & \texttt{|}\ c_1\ \texttt{=}\ ac_1\ \texttt{|>}\ p_1 \\ \vdots & \\ & \texttt{|}\ c_k\ \texttt{=}\ ac_k\ \texttt{|>}\ p_k\end{array}$ is a variant of a definition in $\mathcal{S}$,

$\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, $S \vdash \sigma(c_i)$, $S \nvdash \sigma(c_j)$ for all $j < i$,

and $ac_i = \texttt{[}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{]}$

$$\text{R2}\ \frac{\langle S, p_1\rangle \to \langle S', p_1'\rangle}{\langle S, p_1 \texttt{ >>> } p_2\rangle \to \langle S', p_1' \texttt{ >>> } p_2\rangle}$$

$$\text{R3}\ \frac{\langle S, p_1\rangle \to \langle S', p_1'\rangle}{\langle S, p_1 \texttt{ <|> } p_2\rangle \to \langle S', p_1' \texttt{ <|> } p_2\rangle} \qquad \text{R3'}\ \frac{\langle S, p_2\rangle \to \langle S', p_2'\rangle}{\langle S, p_1 \texttt{ <|> } p_2\rangle \to \langle S', p_1 \texttt{ <|> } p_2'\rangle}$$

$$\text{R4}\ \frac{\langle S, p_1\rangle \to \langle S', p_1'\rangle}{\langle S, p_1 \texttt{ <+> } p_2\rangle \to \langle S', p_1'\rangle} \qquad \text{R4'}\ \frac{\langle S, p_2\rangle \to \langle S', p_2'\rangle}{\langle S, p_1 \texttt{ <+> } p_2\rangle \to \langle S', p_2'\rangle}$$

$$\text{R5}\ \frac{\langle S, p_1\rangle \to \langle S', p_1'\rangle}{\langle S, p_1 \texttt{ <\%> } p_2\rangle \to \langle S', p_1'\rangle}$$

$$\text{R5'}\ \frac{\langle S, p_2\rangle \to \langle S', p_2'\rangle}{\langle S, p_1 \texttt{ <\%> } p_2\rangle \to \langle S', p_2'\rangle}\ \text{if } \nexists S'', p_1' \text{ with } \langle S, p_1\rangle \to \langle S'', p_1'\rangle$$

$$\text{R6}\ \frac{\langle S, p_1\rangle \to \langle S', p_1'\rangle}{\langle S, p_1 \texttt{ <~> } p_2\rangle \to \langle S', p_1' \texttt{ <~> } p_2\rangle}$$

$$\text{R6'}\ \frac{\langle S, p_2\rangle \to \langle S', p_2'\rangle}{\langle S, p_1 \texttt{ <~> } p_2\rangle \to \langle S', p_1 \texttt{ <~> } p_2'\rangle}\ \text{if } \nexists S'', p_1' \text{ with } \langle S, p_1\rangle \to \langle S'', p_1'\rangle$$

Fig. A.1. Operational semantics of process terms

congruent to `Terminate`.[13] Rule R1 describes the application of a process abstraction to a process, where "$S \vdash c$" denotes the validity of condition c in component state S (thus, we omit the mailbox and state parameter in the concrete program rule since they are determined by S). The conditions are checked in their textual order and the first alternative with a valid condition is selected, where the sequence of actions in the corresponding guard are exe-

[13] Usually, components do not terminate and we are interested in observing the sequence of actions performed by a component.

cuted from left to right in one atomic step. We use variants (i.e., renamings of variables) of definitions of process abstractions in order to get fresh names for local variables (which is important for a functional logic base language). To accomodate also base languages with constraint solving and search facilities, as in logic programming, we could also allow the instantiation of unbound variables in guards by requiring "$S \vdash \varphi(\sigma(g))$" and continuing with $\varphi \circ \sigma$ instead of σ.

The remaining rules are quite standard in process algebra, whereas rules R5' and R6' specify the particular priority semantics of the operations `<%>` and `<~>` as informally explained in Section 3 (for these rules and rule R1 the decidability of conditions in process abstractions is important).