In *Proc. 11th International Workshop on Functional and (Constraint) Logic Programming, WFLP 2002* (M. Falaschi, Ed.), Grado (Italy), pp. 89–102, 2002

Programming Autonomous Robots in Curry*

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 by an example.

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 [12, 15] with process-oriented features [5, 6], to the programming of concrete embedded systems.

The embedded systems we consider in this work are the Lego Mindstorms robots.[1] 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.

* 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.

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

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

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 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 time-out 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 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[2]). On the other hand, there are also imperative languages (e.g., "Not Quite C"[3]) or extensions of existing programming languages with compilers for the RCX. A popular representative of the latter kind is based on replacing the standard RCX firmware by a new operating system, *legOS*,[4] and writing programs in C with

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

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 work we will use a declarative multi-paradigm programming language (Curry) with synchronization and process-oriented features to program the RCX. The language Curry [12, 15] 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. [13] contains a proposal to extend Curry with a concept of ports (similar concepts exist also for other languages, like Erlang [2], Oz [18], etc) in order to support the high-level implementation of distributed systems. These ideas have been applied in [5] where a domain-specific language for process-oriented programming is proposed. The target of the latter is the application of Curry for the implementation of reactive and embedded systems. The example applications shown in that paper are simulators of artificial systems, e.g., a lift controller. In this paper we will show the application of 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.

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 model and a complete description of all language features can be found in [12, 15].

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[5] 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

[5] Curry has a Haskell-like syntax [17], 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 ("$f\ e$").

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.[6] Since `conc` is explicitly defined as flexible by "`eval flex`" (as a default, all functions except for constraints are rigid), 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`.

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 [12, 15], 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 [13] 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.

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

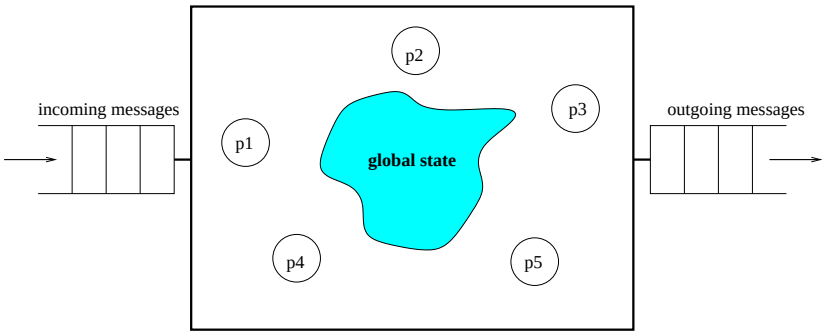

Fig. 2. Component of a dynamic system

3 Specification of Process Systems

In this section we review the framework for process-oriented programming in Curry as proposed in [5]. 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.[7] 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 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

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

the global state to a value s ("`Set` s"), the sending of a message m ("`Send` m"),[8] and the removing of a message m from the mailbox ("`Deq` m").[9]

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 *guard* (i.e., a condition on the mailbox and 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 achieved by recursive process calls with new arguments. Thus, the language of *process terms* p is very similar to process algebra [8] and defined by the following grammar:

$p ::=$	`Terminate`	successful termination
|	`Atomic [`a_1`,...,`a_n`]`	sequence of actions
|	`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

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

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

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

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.

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 *static* are type variables denoting the type of incoming messages, outgoing messages, and the global state in a concrete application, respectively.

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

A similar definition exists for the type `ProcExp` *proc inmsg outmsg static* which has the type *proc* of concrete processes in a system as an additional type parameter. Then the above process combinators (e.g., `>>>`, `<|>`) are operations on this data type.

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

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

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

Hence, the guard is just a standard 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. If a process is specified with several rules or guards, these rules can be considered as combined with the "`<%>`" operator, i.e., the first alternative with a valid guard 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 by the specification function `phil_spec` ("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
```

```
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 [Set (rpl (rpl forks i Used) ((i+1) `mod` n) Used)]
   >>> Proc (Eating i)

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

The mailbox parameter is not used in this simple example. Initially, all philosophers are thinking, which corresponds to the initial process term

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

and all forks are available, which is expressed by the initial state

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

The above specification describes the following behavior. 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 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 guard and the first sequence of actions is one atomic unit which cannot be interrupted by other processes (see also [5]). Therefore, the classical deadlock situation is avoided without low-level synchronization (e.g., semaphores) or additional constructions (e.g., room tickets).

In the next section we will apply this framework to the programming of autonomous 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 [9] 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. 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)`.

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 a concrete example to apply this framework in practice.

5 Examples

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
```

Fig. 3. Example of an obstacle avoiding robot

The control system of the rover contains the following process states:

```
data RoverProc = Go | WaitEvent | Turn TouchEvent | Wait Int
```

We describe the rover behavior by the specification function `rover` of the following type (the global state is not used here and thus of arbitrary type `st`):

```
rover :: RoverProc -> [TouchEvent] -> [st]
                   -> ProcExp RoverProc TouchEvent RobotCmd [st]
```

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:

```
rover Go _ _ =
    Atomic [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`:

```
rover WaitEvent (TouchLeft:_) _ =
   Atomic [Deq TouchLeft]  >>> Proc (Turn TouchLeft)
rover WaitEvent (TouchRight:_) _ =
   Atomic [Deq TouchRight] >>> Proc (Turn TouchRight)

rover (Turn touch) _ _ =
   Atomic [Send (MotorDir Out_A Rev), Send (MotorDir Out_C Rev)]
   >>> Proc (Wait 2000) >>>
   Atomic [Send (MotorDir
              (if touch==TouchLeft then Out_A else Out_C) Fwd)]
   >>> Proc (Wait 2000) >>>
   Proc Go
```

This simple example shows only the basic features of our framework to program autonomous robots. It does neither show the use of several parallel processes nor the use of the global state to synchronize them. 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 compiler to translate the Curry specifications into binary code that can run on the RCX. In order to test the programs, 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 [5, 6]. In order to run a typical robot program as shown in Section 5, the program needs some input from the sensors and one 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.

Currently, the simulator is very primitive. 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 [14]. 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 [16] where a functional robot control language is proposed which is executed on top of Haskell running on a powerful Linux system). In particular, the interpretation of the process ex-

pressions 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 can be translated in a straightforward 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 will be the next step.

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 [10] 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 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 [11]. This is a major drawback since one do not has 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 will investigate compilation techniques for producing 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.

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. 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.
6. 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.
7. 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.
8. W. Fokkink. *Introduction to Process Algebra*. Springer, 2000.
9. N. Halbwachs. Synchronous programming of reactive systems. In *Tenth International Conference on Computer-Aided Verification (CAV'98)*, pp. 1–16. Springer LNCS 1427, 1998.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.

16. 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.
17. S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. http://www.haskell.org, 1999.
18. 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.