final version:
Robotics and Autonomous Systems, 39 (2002), p.115-127, Elsevier

Programming with Behavior-Processes

Andreas Birk [a,1] Holger Kenn [a] Luc Steels [b]

[a] *International University Bremen, Campus Ring 1, 28759 Bremen, Germany*

[b] *Vrije Universiteit Brussel, AI-Lab, Pleinlaan 2, 1050 Brussels, Belgium*

Abstract

The so-called CubeOS is a special software environment for behavior-oriented robotics. It ranges from a dedicated nano-kernel and hardware drivers for a broad set of sensors and actuators over operating system support for concurrent and real-time programming to a special high-level language suited for novices in the field. As most special feature, the CubeOS framework includes a novel scheduler, designed for the particular needs of behavior-oriented robotics.

Key words: realtime, scheduling, control, operating system, behavior oriented

1 Introduction

The field of robotics has undergone tremendous changes since the mid-eighties on the commercial as well as on the scientific side. The robotics market until the mid-eighties was almost completely dominated by robot-arms used in industrial manufacturing. Meanwhile, service robots [Eng89], edutainment robots [ADB$^+$00], and various smaller niches [Bir98a] broadened and extended the robotics market. On the scientific side, the novel branch of so-called behavior-oriented robotics [Ark98] emerged, following Brooks' famous critique on "classic" AI and robotics [Bro91,Bro86b,Bro86a]. These two simultaneous shifts in focus, sometimes even dubbed revolutions, came along with a series of fundamental up to philosophical debates. Especially, the notion of "behavior", which runs as a red thread through both shifts, is used within a wide range of interpretations and definitions as pointed out for example in [Ste94a].

[1] a.birk@iu-bremen.de

As behavior-oriented robotics and its applications become more and more mature, it is time to focus on efficient implementations of its principles instead of keeping on discussing what these principles are. Here, we deal with a "behavior" from a software engineering viewpoint, namely as a software process with a particular set of properties. The most important one is that several behaviors can be "active" at the same time. From a practical viewpoint, this means that behaviors must be executed in (pseudo-)parallel, i.e., there must be support for *concurrent* programming. In addition, a software environment for behavior-oriented robotics obviously deals with control. Hence, there must be support for *real-time* processes, ensuring guaranteed time-related qualities of service. Existing behavior-oriented programming languages like the subsumption architecture [Bro86b,Bro90] or motor schemas [Ark87,Ark92] came out of early scientific work in this field. Accordingly, they did not incorporate any considerations on efficiency or software-engineering, forcing the user to do a lot of hand-tailoring for each particular application. As a consequence, these languages are not widely distributed. Instead, the complete software environment for every behavior-oriented project around the globe is usually developed from scratch.

The so-called CubeSystem-project is an attempt to overcome this situation. The CubeSystem is a kind of advanced construction-kit for robotics, including hardware as well as software components. The software side, on which we focus here, centers around the so-called CubeOS, a special operating system designed to support behavior-oriented programming. First of all, it features standard programming constructs for real-time and concurrent programming [BW97,Mel83,You82]. Furthermore, it supports a wide range of devices employed in the CubeSystem through libraries and it facilitates the development of new drivers to incorporate further devices, let it be sensors, actuators, or computational hardware. Last but not least, it features a novel scheduling scheme designed for behavioral processes. This so-called B-scheduling can handle behaviors running on different time-scales represented through so-called exponential effect priorities. It is usually neglected that behavioral processes can span very different time-periods. A process doing pulse-width-modulation (PWM) has for examples to operate for some DC-motors in the 20 kHz range, i.e., on a time-basis of $5 \cdot 10^{-5}$ seconds. A behavior monitoring batteries in contrast operates on a scale of minutes. Some adaptive or learning behaviors can operate on much higher scales like hours or even days. The idea of exponential effect priorities is therefore to cover a wide range of time-scales. Hence, the periodicity of a process is halved when its priority value is increased by one. Scheduling processes with such widely spread periods is a non-trivial task. The novel scheme of B-scheduling results in guaranteed performance regarding the periodicity of the processes, a very important feature for control, while eliminating idle-time, i.e., B-scheduling achieves time-optimal execution of processes.

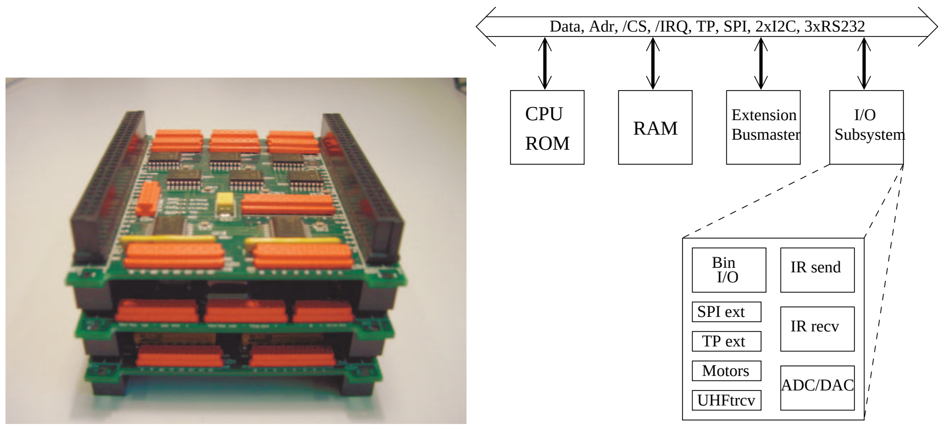


Fig. 1. A picture of the RoboCube (left) and the layout of its internal bus structure (right).

The rest of this article is structured as follows. Section two gives a short overview on the hardware side of the CubeSystem and presents some of the applications where it is employed. In section three, the basic technical details of the CubeOS are introduced. Section four introduces B-scheduling and presents results. In section five, the process description language (PDL) as high-level option to program with CubeOS is shortly presented. Section six concludes the article.

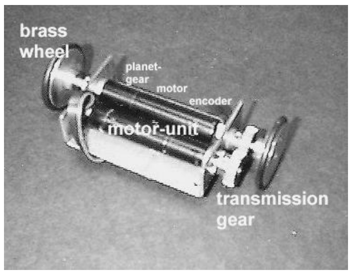
2 The Hardware-Side of the CubeSystem

2.1 The RoboCube as Embedded Controller

The CubeOS runs on different hardware platforms [Ken00]. The so-called RoboCube (figure 1) is the most important one within the CubeSystem. The RoboCube [BKW00,BKW98] has a open bus architecture which allows to add “infinitely” many sensor/motor-interfaces (at the price of bandwidth). But for most applications the standard set of interfaces should be more than sufficient. RoboCube’s basic set of ports consists of

- 24 analog/digital (A/D) converter,
- 6 digital/analog (D/A) converter,
- 16 binary Input/Output (binI/O),
- 5 binary Inputs,
- 10 timer channels (TPC),
- 3 DC-motor controller with pulse-accumulation (PAC)

The basic RoboCube features a 32-bit processor, the Motorola MC68332, 1 Mbyte Flash-EPROM, and 1 Mbyte SRAM. The RoboCube is extremely com-



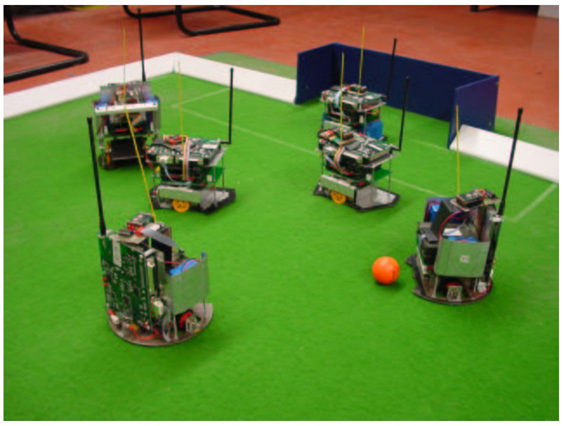


Fig. 2. The drive unit (left) as a mechanical building-block, which can be integrated into several different robots for the RoboCup small size league, like e.g. the ones shown on the right.

pact, namely 50 mm × 60 mm × 80 mm, as special stacking connectors are used to build the global bus perpendicular to the plane of the boards. The system can therefore be easily extended by stacking additional boards on top of the others. This layout is also mechanically very stable and guarantees secure connections. It leads to a cubic form of the controller, hence the name RoboCube. RoboCubes can be networked together with host-PCs via several serial ports or in a wireless manner via special RF-modules included in the CubeSystem.

2.2 A Versatile System

One application of the CubeSystem is within the Small Robots League of RoboCup, the worldchampionship of robot soccer [KAK+97,KTS+97]. There, the computational core is used together with specially engineered, solid mechanical building blocks (figure 2). The main research themes for this team are on-board control and the exploitation of heterogeneity [BK99]. A detailed description of the team is found in [BWBK99,BWB+98].

The infrastructure for the Small Size team is also used for educational work that originated at the Vrije Universiteit Brussel (VUB) and that is now continued at the International University Bremen (IUB). In addition, mid-sized robots based on the CubeSystem and mechanical construction kits (figure 3) have been used for a course on Autonomous Systems at the VUB and the German University of Koblenz-Landau. The course consists of a theoretical lecture and practical exercises where the students build and program robots [ADB+00].

The so-called VUB ecosystem is an other robotic environment where the

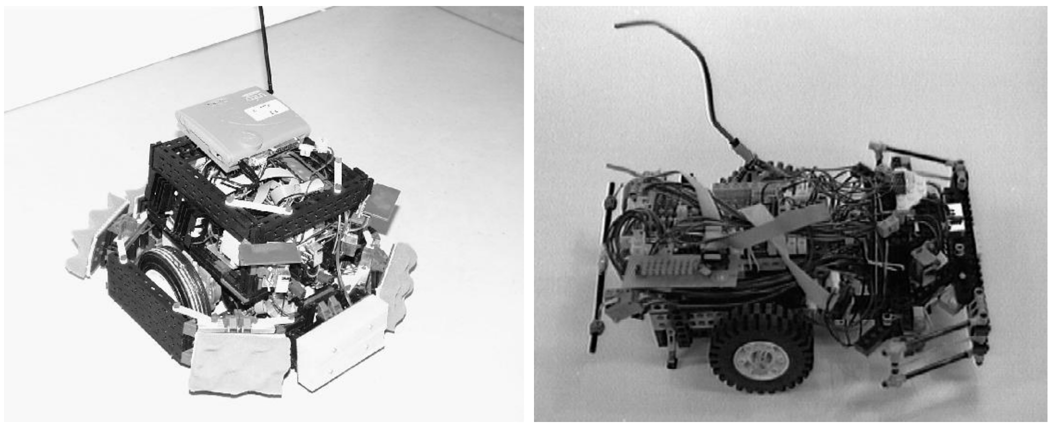


Fig. 3. Different mid-sized robots based on the CubeSystem and mechanical components from Fischertechnik^{TM} (left) and Lego^{TM} (right).

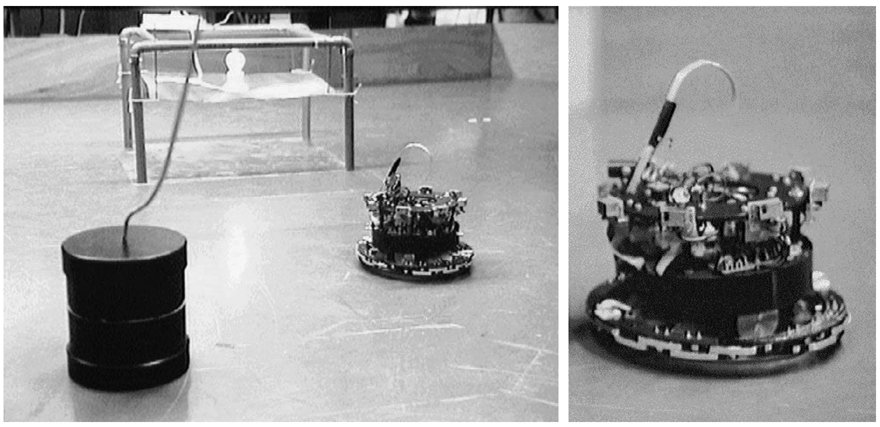


Fig. 4. A partial view of the so-called ecosystem (left) with a charging station, one of the mobile robots and one of the so-called competitors. Mobile robots (right) can operate over extended periods in time in the ecosystem by autonomously re-charging their batteries. The competitors establish a kind of working task.

CubeSystem provides the infrastructure. In the basic ecosystem (figure 4), mobile robots stay operational over extended periods in time by autonomously re-charging their batteries [Bir97]. So-called competitors establish a working task, such that the robots are kept busy [Ste94b,McF94]. In an extended version, robots also face dangerous situations which must be avoided [BB97]. Despite its simplicity, the VUB ecosystem provides many possibilities for research on various subjects including basic economic concepts [BW98], learning [Bir98b,Ste96a,Ste96b], heterogeneity [BB98], cooperation [BW00], trust [Bir00a,Bir00b], and many more. In addition to education and basic research, the CubeSystem is incorporated in industrial projects. One is the so-called RoboGuard (figure 5), a semi-autonomous robot for surveillance applications, marketed by the Belgian SME Quadrox.

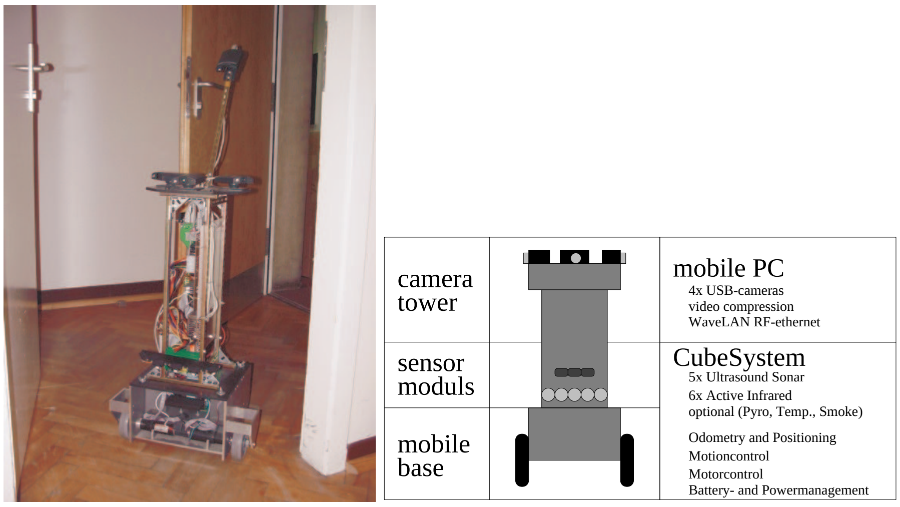


Fig. 5. The inside core of the RoboGuard base, a commercial semi-autonomous robot for surveillance applications.

3 Inside the CubeOS

As motivated in the introduction, CubeOS was developed as a modularized realtime executive for behavior-based robotics. The CubeOS target code consists of a small memory footprint nanokernel, a number of sensor- and actuator software drivers and a network stack for wireless communication. The application and the necessary parts of the target code are linked together on the host system to form the binary application image that is then downloaded into the RoboCube hardware. The target code consists of the following core modules:

- *The Nanokernel* provides basic OS functionality. Among others, it implements multithreading, interrupt service, IPC, semaphores and mutexes, timer and clock functions, basic i/o and system initialization and configuration services.
- *The Network Stack* implements functions to communicate over a simple wired or wireless network and for platform-independent data exchange.
- *The CubeOS API* provides access to all the kernel and driver functions. It is a subset of the POSIX standard which is enhanced by several additional functions.

Access to sensors and actuators is provided by a library of functions that in turn uses the CubeOS API for accessing the hardware. By using this layered approach, the framework hides the details and provides the user with a simple interface to control the sensor- and actuator devices of the system. The internal realtime clock of the nanocore provides millisecond resolution. This clock is also used to trigger the preemption of the application threads by the nanocore scheduler and to drive the CubeOS functions that deal with time. The nanocore's internal scheduler is a preemptive round-robin scheduler with priorities. It is mainly used to provide CPU time to the internal CubeOS services such as communication. Although the internal CubeOS threads have a higher priority than the application program, they are often suspended, and therefore leaving most of the CPU time to the application program. The internal network stack implements a layered communication infrastructure. Its lowest level is formed by a hardware-triggered state machine that receives a stream of bytes. It breaks it into frames which are then presented to the higher protocol layers. These run within the nanocore multithreading and are using the nanocores IPC mechanisms to communicate. Depending on the application, there are multiple internal communication layers which provide media arbitration, resend of lost data, packetizing and depacketizing of streams, and platform-independent data encoding (XDR).

4 Priorities and Efficient Scheduling

4.1 Exponential-Effect Priorities

It is often neglected that behavioral processes can span very different time-periods. A process doing pulse-width-modulation (PWM) has for examples to operate for some DC-motors in the 20 kHz range, i.e., on a time-basis of $5 \cdot 10^{-5}$ seconds. A behavior monitoring batteries in contrast operates on a scale of minutes. Some adaptive or learning behaviors could operate on much higher scales like hours or even days. So, it is desirable to span several orders of magnitude for the time-periods of different processes. A linear priority scheme is not suited for this. Therefore, so-called *exponential effect priorities* are introduced here. The idea is that for each increase in a priority-value by one, the periodicity is halved.

In the remainder of this article the following naming conventions are used:

- the set of processes: $\mathcal{P}= \{p_0, ..., p_{N-1}\}$
- the priority-value of process p_i: $pv[p_i]$
- the set of processes with priority k or the k-th priority class: PC_k
- the highest used priority-value: $maxpv$

So, the exact semantic of a priority-value $pv[p_i]$ of process p_i within *exponential effect priorities* is:

- $pv[p_i] = 0 \iff p_i$ is executed with the maximum frequency f_0
- $pv[p_i] = n \iff p_i$ is executed with the frequency f_n which is half the frequency of the previous priority-class, i.e., $f_n = f_{n-1}/2$

```
/* Initialization */
/* computing the initial $wait$-values for each process $p_{id}$ */
quicksort($\mathcal{P}$)
$pc = 1$
$start = 0$
$n_{slots} = 1$
$\forall i \in \{0, ..., maxpv - 1\}$ : {
        $start = 2 \cdot start$
        $n_{slots} = 2 \cdot n_{slots}$
        $\forall id$ with $pv[p_{id}] = pc$ : {
                $wait[p_{id}]$ = reverse($(start + id)$ modulo $n_{slots}$)
        }
        $start = (start + \#\{p_{id} \mid pv[p_{id}] = pc\})$ modulo $n_{slots}$
        $pc = pc + 1$
}
```

Fig. 6. The initialization of B-scheduling.

name	p1.1	p1.2	p1.3	p2.1	p2.2	p3.1	p4.1	p4.2	p4.3
$pv[]$	1	1	1	2	2	3	4	4	4
$2^{pv[]}$	2	2	2	4	4	8	16	16	16
wait	0	1	0	1	3	0	4	12	2

Table 1
A set of processes with their priority-values $pv[]$, their according waiting-time between executions, and their initial wait values calculated with the algorithm shown in figure 6. The wait values lead to the schedule shown in table 2 when the B-scheduler (figure 7) is invoked.

4.2 B-Scheduling

For solving the task of finding a suitable order of execution of the processes, we use a *cyclic executive scheduling* approach [BW97]. This means there is a so-called *major cycle*, which is constantly repeated. The major cycle consists of several so-called *minor cycles*. Each minor cycle is a set of processes, which are executed when the minor cycle is activated. The general problem of finding a suitable schedule within this approach is NP-hard as it can be reduced to the Bin-Packing-problem in a straight-forward manner. We present an extremely efficient, namely linear-time algorithm, which is based on the restriction to exponential-effect-priorities. As motivated above, we do not see this as a limitation, but even as a feature.

```
/* Execute the Major Cycle */
for($round = 0$; $round < n_{mic}$; $round = round + 1$) {
        /* Execute the Minor Cycle */
        $id = 0$
        $done = 0$
        while( $(done < perfect)\ \wedge\ (id < \#\mathcal{P})$ ) {
                if($wait[p_{id}] == 0$) {
                        execute $p_{id}$
                        $wait[p_{id}] = 2^{pv[p_{id}]}$
                        $done = done + 1$
                }
                $id = id + 1$
        }
        $\forall p_{id} \in \mathcal{P}$ : if($wait[p_{id}] > 0$) : $wait[p_{id}] = wait[p_i] - 1$
}
```

Fig. 7. The execution of a B-schedule.

B-scheduling is implemented in CubeOS with C. Figure 6 and figure 7 show the critical parts of B-scheduling in a pseudo-code. An important variable in both parts is $wait[p_{id}]$. It specifies for each process p_{id} how long it has to wait in number of cycles until it is executed again. During the execution of a B-schedule (figure 7), wait is constantly decremented in each cycle. When a process p_{id} is executed, its wait $wait[p_{id}]$ is set to $2^{pv[p_{id}]}$. Therefore, the execution of p_{id} is spread evenly over the minor cycles in the major cycle.

The dynamic execution part of a B-schedule (figure 7) is more or less straightforward. The "real magic" is done in the static initialization of the *wait*-values (figure 6). Note that the initial value of $wait[p_{id}]$ determines in which minor cycle p_{id} will be executed for the first time. So, computing suited initial waits produces a B-schedule. Note, that the number of *wait*-values is equal to the number of processes $\#\mathcal{P}$. So, the complete schedule which is of size $O(2^{\#\mathcal{P}})$ is represented in a single variable for each process, i.e., in the overall size $O(\#\mathcal{P})$.

Before discussing the initialization of the *wait*-values in more detail, a special command from figure 6 has to be explained. The `reverse()` is used to reverse the bit-order of a binary number. More concretly, let $B_n = [b_0, ..., b_{n-1}]$ and $R_n = [r_0, ..., r_{n-1}]$ denote two binary numbers, each represented as array of bits b_i, respectively r_i. The function `reverse`() is then defined as:

$$\texttt{reverse}(B_n) = R_n \text{ with } r_i = b_{n-i}$$

minor cycle number	processes within the cycle
0	p1.1 p1.3 p3.1
1	p1.2 p2.1
2	p1.1 p1.3 p4.3
3	p1.2 p2.2
4	p1.1 p1.3 p4.1
5	p1.2 p2.1
6	p1.1 p1.3
7	p1.2 p2.2
8	p1.1 p1.3 p3.1
9	p1.2 p2.1
10	p1.1 p1.3
11	p1.2 p2.2
12	p1.1 p1.3 p4.2
13	p1.2 p2.1
14	p1.1 p1.3
15	p1.2 p2.2

Table 2
A simple example of a major cycle computed with B-scheduling. The notation $pX.Y$ denotes process number Y within priority-class PC_X. Note that there is no straight-forward distribution of dirty and perfect cycles, i.e., minor cycles which consist in this example of either two or three processes.

The main idea when computing suited initial *wait*-values is as follows. Imagine a set $\mathcal{S}$ of natural numbers with a cardinality equal to a power of 2. Let $S(start, d)$ denote a sequence which begins at the number *start* and "jumps" further to numbers x which are distance d away, i.e., $x = (k \cdot d)\texttt{modulo}\#\mathcal{S}$ with $k \in \mathbb{N}$. When *start* and d are powers of 2, S is called harmonic. It holds that for each harmonic list S, we can create two harmonic lists S_1 and S_2 such that $S = S_1 \cup S_2$, namely:

- $S_1 = S(start, 2 \cdot d)$
- $S_2 = S(start + d/2, 2 \cdot d)$

The overall set $\mathcal{S}$ can be expressed as $S(0, 1)$. It can recursively be divided in smaller lists and sublist.

When computing the initial *wait*-values, the goal is to distribute processes such, that the minor cycles are equally filled up. Each execution process of class PC_k can be seen as a list $S(start, 2^{maxpv-k})$ of minor cycles. The first value for *start* is zero, i.e., the first slot in the first minor cycle is used. The distance d is $2^{maxpv-pv[p_0]}$. From then on, further lists can be computed. The difficulty is to keep track of the *start* position. Especially, so-to-say left-overs, i.e., empty lists not used up by class PC_{k-1}, have to be used when the class PC_{k-1} is handled.

Table 1 shows as an example a set of processes with their priority-values $pv[]$, their according waiting-time $2^{pv[]}$ between executions, and their initial wait values calculated with the algorithm shown in figure 6. The interested reader can try to find a time-optimal, well balanced schedule of the processes (of course without using the pre-computed wait-values). The time-optimal, well balanced schedule computed by B-scheduling is shown in table 2.

5 High-level Language Support

The so-called Process Description Language (PDL) was introduced in [Ste92] and later on extended [BKS00]. PDL provides behavior-oriented programming functionality in a high-level language format on top of CubeOS. Therefore, it facilitates an easy start for novices to the field as has been proven in various educational activities. PDL enables the efficient description of a network of dynamical processes in terms of variables whose state changes at the beginning of each program execution cycle.

The basic PDL-programming constructs are:

`quantity` : A bounded variable q, i.e., a variable with fixed minimum and maximum value. Sensor- and motor-values are represented by basic quantities which can only read, or respectively be written.

`process` : A piece of program which is executed in (virtual) parallel with other processes in so-called PDL-cycles.

`value(`q`)` : This function returns the value of the quantity q from the previous PDL-cycle.

`add_value(`q`, `e`)` : This procedure influences the value of a quantity q by summing the evaluation of the expression e to q. The change takes only effect at the end of the PDL-cycle in which the procedure was activated. Note that other `add-value` commands in the same process or in other processes can influence q at the same time.

`dt()` : this function returns the time-difference between the start of the recent PDL-cycle and the start of the previous PDL-cycle

In the implementation in the CubeOS framework, the quantities are represented by a `struct` datastructure that holds both the current and the future numerical value. All native numerical datatypes of C can be used here, i.e. `float` or `short`, however, the programmer has to take care of the specific properties of the datatype to prevent overflows or imprecisions. The PDL processes are implemented as simple argumentless C functions that do not return values. Instead, the only data exchange with other parts of the program are implemented through the access functions to quantities which are global variables. The access functions `value(q)` and `add_value(q,x)` are implemented as macros to increase efficiency. To make the PDL runtime system aware of the presence of a PDL process, a special C function `add_process()` is implemented that takes the C-function implementing the PDL process as argument. By calling the `run_pdl()` function, the application program then invokes the B-scheduler as one thread of the internal CubeOS multithreading that in turn executes the predefined PDL processes.

6 Conclusion

The article described a software environment for behavior-oriented robotics. This environment is constructed around CubeOS, a special operating framework, from a dedicated nano-kernel and hardware drivers for a broad set of sensors and actuators over operating system support for concurrent and real-time programming to a special high-level language suited for novices in the field. The CubeOS is the software part of the CubeSystem, a kind of construction kit for behavior-oriented robotics which is successfully used in a constantly growing number of applications.

The CubeOS is not only an engineering effort for providing useful software functionality within a behavior-oriented robotics background. In addition, the CubeOS framework includes a novel scheduler, designed for the particular needs of behavior-oriented robotics. This so-called B-scheduling can handle behaviors running on different time-scales represented through so-called exponential effect priorities, covering a wide range of time-scales. Concretly, the periodicy of a process is halved when its priority value is increased by one. Scheduling processes with such widely spread periods is a non-trivial task. The novel scheme of B-scheduling results in guaranteed performance regarding the periodicy of the processes, a very important feature for control, while eliminating idle-time, i.e., B-scheduling achieves time-optimal execution of processes.

References

[ADB$^+$00] Minoru Asada, Raffaello D'Andrea, Andreas Birk, Hiroaki Kitano, and Manuela Veloso. Robotics in edutainment. In *Proceedings of the International Conference on Robotics and Automation, ICRA'2000*, 2000.

[Ark87] R. C. Arkin. Motor schema based navigation for a mobile robot. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 264–271, 1987.

[Ark92] Ronald C. Arkin. Cooperation without communication: Multiagent schema-based robot navigation. *Journal of Robotic Systems*, 9(3):351–364, April 1992.

[Ark98] Ronald C. Arkin. *Behavior-Based Robotics.* The MIT Press, 1998.

[BB97] Tony Belpaeme and Andreas Birk. On the watch. In *Proceedings of the 30th International Symposium on Automative Technology and Automation*, 1997.

[BB98] Andreas Birk and Tony Belpaeme. A multi-agent-system based on heterogeneous robots. In Alexis Drogoul, Milind Tambe, and Toshio Fukuda, editors, *Collective Robotics, CRW'98*, LNAI 1456. Springer, 1998.

[Bir97] Andreas Birk. Autonomous recharging of mobile robots. In *Proceedings of the 30th International Symposium on Automative Technology and Automation*, 1997.

[Bir98a] Andreas Birk. Behavior-based robotics, its scope and its prospects. In *Proc. of The 24th Annual Conference of the IEEE Industrial Electronics.* IEEE Press, 1998.

[Bir98b] Andreas Birk. Robot learning and self-sufficiency: What the energy-level can tell us about a robot's performance. In *Proceedings of the Sixth European Workshop on Learning Robots*, LNAI 1545. Springer, 1998.

[Bir00a] Andreas Birk. Boosting cooperation by evolving trust. *Applied Artificial Intelligence Journal*, 14(8), September 2000.

[Bir00b] Andreas Birk. Learning to trust. In Falcone, Singh, and Tan, editors, *Deception, Fraud and Trust in Agent Societies*, LNAI. Springer, 2000.

[BK99] Andreas Birk and Holger Kenn. Heterogeneity and on-board control in the small robots league. In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, number 1856 in LNAI, pages 196 – 209. Springer, 1999.

[BKS00] Andreas Birk, Holger Kenn, and Luc Steels. Efficient behavioral processes. In Meyer, Berthoz, Floreano, Roitblat, and Wilson, editors, *From Animals to Animats 6, SAB 2000 Proceedings Supplement Book*. The International Society for Adaptive Behavior, 2000.

[BKW98] Andreas Birk, Holger Kenn, and Thomas Walle. Robocube: an "universal" "special-purpose" hardware for the robocup small robots league. In *4th International Symposium on Distributed Autonomous Robotic Systems*. Springer, 1998.

[BKW00] Andreas Birk, Holger Kenn, and Thomas Walle. On-board control in the robocup small robots league. *Advanced Robotics Journal*, 14(1):27 – 36, 2000.

[Bro86a] Rodney Brooks. Achieving artificial intelligence through building robots. Technical Report AI memo 899, MIT AI-lab, 1986.

[Bro86b] Rodney A. Brooks. A robust layered control system for a mobile robot. In *IEEE Journal of Robotics and Automation*, volume RA-2 (1), pages 14–23, April 1986.

[Bro90] Rodney A. Brooks. The behavior language; user's guide. Technical Report A. I. MEMO 1227, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, April 1990.

[Bro91] Rodney Brooks. Intelligence without reason. In *Proc. of IJCAI-91*. Morgan Kaufmann, San Mateo, 1991.

[BW97] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[BW98] Andreas Birk and Julie Wiernik. Economic aspects of a real-world ecosystem featuring several robot species. In *2nd Workshop on Economics with Heterogeneous Interacting Agents*. Elgar, 1998.

[BW00] Andreas Birk and Julie Wiernik. An n-player prisoner's dilemma in a robotic ecosystem. In *8th International Symposium on Intelligent Robotic Systems, SIRS'00*, 2000.

[BWB^{+}98] Andreas Birk, Thomas Walle, Tony Belpaeme, Johan Parent, Tom De Vlaminck, and Holger Kenn. The small league robocup team of the vub ai-lab. In *Proc. of The Second International Workshop on RoboCup*. Springer, 1998.

[BWBK99] Andreas Birk, Thomas Walle, Tony Belpaeme, and Holger Kenn. The vub ai-lab robocup'99 small league team. In *Proc. of the Third RoboCup*. Springer, 1999.

[Eng89] Joseph F. Engelberger. *Robotics in Service*. MIT Press, Cambridge, Massachusetts, 1989.

[KAK⁺97] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *Proc. of The First International Conference on Autonomous Agents (Agents-97).* The ACM Press, 1997.

[Ken00] Holger Kenn. Cubeos, the manual. Technical Report MEMO 00-04, Vrije Universiteit Brussel, AI-Laboratory, 2000.

[KTS⁺97] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The robocup synthetic agent challenge 97. In *Proceedings of IJCAI-97*, 1997.

[McF94] David McFarland. Towards robot cooperation. In Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 3. Proc. of the Third International Conference on Simulation of Adaptive Behavior.* The MIT Press/Bradford Books, Cambridge, 1994.

[Mel83] Mellichamp. *Real-Time Computing.* Van Nostrand Reinhold, New York, 1983.

[Ste92] Luc Steels. The pdl reference manual. Technical Report MEMO 92-05, Vrije Universiteit Brussel, AI-Laboratory, 1992.

[Ste94a] Luc Steels. The artificial life roots of artificial intelligence. *Artificial Life Journal*, 1(1), 1994.

[Ste94b] Luc Steels. A case study in the behavior-oriented design of autonomous agents. In Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 3. Proc. of the Third International Conference on Simulation of Adaptive Behavior.* The MIT Press/Bradford Books, Cambridge, 1994.

[Ste96a] Luc Steels. Discovering the competitors. *Journal of Adaptive Behavior 4(2)*, 1996.

[Ste96b] Luc Steels. A selectionist mechanism for autonomous behavior acquisition. *Journal of Robotics and Autonomous Systems 16*, 1996.

[You82] SJ Young. *Real Time Languages.* Ellis Horwood, 1982.