A Scripting-based Approach to Robot Behavior Engineering Using Hierarchical Generators

Thijs Jeffry de Haas, Tim Laue and Thomas Röfer

Abstract— **When developing software for autonomous robots, the aspect of behavior engineering is, among tasks such as sensing, state estimation, and motion control, of major importance. Current solutions range from basic behavior-based approaches to sophisticated reasoning systems, in each case depending on the complexity of the robot's task as well as the available amount of computing time. In this paper, we present a behavior specification language, which is called *b-script*, to describe hierarchical agent behaviors using the programming concept of generators. We show that this is a convenient approach to realize complex robot behaviors in an intuitive and clean way that can be used in large-scale. Furthermore, the actual implementation of this language is in particular suited to be used on resource-restricted embedded systems. This is shown in different examples of a Nao robot in a robot soccer scenario.**

I. Introduction

Each robot that operates (at least partially) autonomously needs an action selection (aka behavior) component to map sensor input or world models to actions. For some trivial tasks, this can already be achieved by sensor actuator coupling, but in general, more sophisticated mechanisms are necessary. These range from behavior-based approaches [1] over pragmatically hand-coded finite state machines [2] to complex planning systems such as [3] or [4]. Amongst others, the applied approach might depend on the complexity of the task, the number of different sub-tasks, as well as of the necessary amount of computing time that is available on the used robot platform. The actual task of defining a robot's action selection, which is often named as behavior engineering, is, in general, supported by various description languages or logics.

In this work, we focus on the development of behaviors for robots with limited computational resources in dynamic scenarios that require acting in realtime and that cannot be handled by behavior-based approaches due to their complexity or a strictly specified workflow that might conflict with emergent behaviors. Typical examples for such scenarios are office delivery, (semi-)autonomous transportation of humans, or industrial pick and place tasks. In such domains, it is a common practice to preprogram decisions explicitly and to avoid any computationally expensive inference. This is often realized by implementing state machines, either by using special description languages or by directly using some kind of common programming language.

Thijs Jeffry de Haas is with Department of Mathematics and Computer Science, Universität Bremen, Bremen, Germany. `jeffry@informatik.uni-bremen.de`

Tim Laue and Thomas Röfer are with Cyber-Physical Systems, DFKI, Germany. {`Tim.Laue, Thomas. Roefer`}`@dfki.de`

The contribution of this paper is a domain-specific language called *b-script* to describe hierarchical agent behaviors using the programming concept of generators. We show that it is a convenient way to use hierarchical continuation-based routines to realize complex robot behaviors in an intuitive and clean way and that it can be used in large-scale applications. This is achieved by using a specialized generator implementation, similar to Python's generators, including implicit context maintenance. The language is in particular suited to be used on resource restricted embedded systems, such as the Nao robot [5], and was already successfully applied in actual competitions of the RoboCup [6] Standard Platform League [7]. Furthermore, *b-script* has a strong type system and a scripting-like look and feel that can be compiled to C++.

The development was guided by different domain-specific requirements, in particular:

- light-weight due to resource restricted hardware
- allow execution in realtime
- easily embeddable into existing robot architectures (usually written in C/C++)
- consider long and costly edit-compile-deploy-test cycle
- "runtime safe" (no runtime exceptions)

In theory, the basic functionality of this approach could also be realized by using several established high-level languages, such as Python, Lua, Ruby, or C#. But in practice, these languages have different drawbacks, as they are either dynamic languages (conflicts with "runtime safe" and the long edit-compile-deploy-test cycle) or need extensive runtime libraries such as Microsoft's Common Language Runtime (conflicts with light-weight). In addition, for modeling robot behavior, the full expressiveness of general-purpose languages is not needed. In contrast, it can even be a drawback when developing in large teams, because there is always the risk of the introduction of hacks, i. e. constructs that seriously impede the maintainability of the whole system. Therefore, with *b-script* we specifically designed a language for modeling robot behavior. As such, it excludes concepts that are not needed for its purpose and bear the risk of resulting in confusing code (e. g. global variables) and it introduces new ones that are very beneficial in its domain (e. g. tasks).

This paper is organized as follows: Section II provides an overview of related work in general and the basic concept of generators in particular. Their use in this work is explained in Sect. III, followed by a description of their actual application for specifying complex behaviors Sect. IV. The *b-script* language details are presented in Sect. V. Finally, recent applications of *b-script* are shown along with an evaluation in Sect. VI.

II. Related Work

As aforementioned, finite-state machines are a common choice for describing a robot's behavior. Although a direct implementation is possible in almost any programming language, it is a common practice to use domain specific languages. Some recent examples include XABSL [2], XRobots [8], and the use of UML state charts [9], [10]. These implementations have in common that they are not limited to the specification of single, possibly very huge state machines but that they also allow structuring complex behaviors by splitting them into a hierarchy (such as the example in Fig. 1b) of smaller state machines.

Other behavior description systems are COLBERT [11], which includes finite state machines as one major concept, as well as TDL (Task Description Language) [12], an extension to C++. Also the known URBI framework [13] includes its own, event-based behavior scripting language called UrbiScript. All three approaches are based on concurrent code blocks to realize the decomposition of problems, a concept that makes the detailed interplay of different tasks less comprehensible and – in case of bugs or necessary fine-tuning – harder to debug. In addition, COLBERT as well as URBIScript are both embedded in specific control frameworks – Sapphira and URBI respectively – and cannot be considered as light-weight regarding a usage inside other frameworks or control concepts.

Most behavior engineering approaches use programming paradigms that differ from the one that is, in general, used for most other software components inside a robot system: imperative programming[1]. Huge problems are decomposed into small bits and assembled in a hierarchical structure. Each component in the hierarchy is usually a function that solves a sub-problem (e. g. finding the brightest pixel in an image). This allows creating complex programs that are well structured and relatively easy to understand. Almost all programmers are familiar with this programming concept.

Considering robot behaviors on an abstract level, there is actually only a minor difference from conventional programming problems: the (sub)problems cannot be solved immediately but need to be approached iteratively in discrete steps (e. g. approaching an object and manipulating it). However, usual blocking functions are not adequate to decompose these problems hierarchically. By using the programming concept of generators, we can create function-like routines that provide exactly the behavior required. To put it simple, they allow creating routines that can be suspended (using the *yield* statement) at any point in execution and be resumed later. When suspended, the context (program counter and local variables) of the generator will be saved. When resumed, the generator will continue its execution from the saved context. Certain programming problems can be modeled very elegantly with generators. In particular, problems that are typical use cases for finite state machines, such as parsers or communication protocols, can be described efficiently, as demonstrated by [14]. A basic example that computes a Fibonacci sequence using the *b-script* generator implementation is shown in Fig. 1a.

Jouvin introduced an approach to use continuations – the functionality underlying generators [15] – to specify behaviors of conversational agents [16]. It is shown that continuations can be used to describe agent behaviors in an elegant and intuitive way, however, no applications to robot behaviors are considered. Interestingly, it seems that this approach was not further investigated. Furthermore, Bruce and McMillen discussed the possibility of implementing a robot soccer goalkeeper using Python generators [17]. They showed a rudimentary example but apparently did not apply the approach to real robots.

III. Tasks – Generators for Robot Behaviors

The core of our approach are so-called *tasks*. These are specialized generators that provide a handy way to describe hierarchical robot behaviors. In this section, the characteristics compared to generators are described.

A. Context Maintenance

When using common generator implementations, there is the need to handle the context of a generator explicitly. One can create multiple instances of the same generator. For specifying robot behaviors, this is not necessary since each component of the hierarchy represents a *skill* and could be treated as some kind of singleton (like a function). Therefore, each *task* is associated with a single context instance, which is stored in the *b-script* runtime environment and is retrieved when the *task* gets called (cf. Fig. 1a).

B. Task Parameters

Regular generators evaluate their parameters only once when the context is created. When used in a hierarchical orchestra controlling a robot, this usually is not very helpful. Consider a problem from the robot soccer domain: the *task* *Motion::walkTo(position)* in Fig. 1c that is called by a *task* that tries to approach the ball. The relative position of the ball will change while approaching it and hence the target position passed to *walkTo* needs to be updated in every frame. Therefore, *b-script* evaluates the parameters of a *task* at each call.

C. Task States

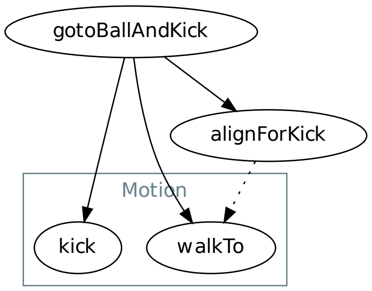
The context of a common generator usually has discrete states. If not in execution, the state is either *suspended* or *dead*. A *suspended* context can be resumed whereas a *dead* context indicates that the generator terminated and can not be resumed. In contrast, in *b-script* a *task* can be in a nearly arbitrary number of states. If a *task* is *running* (any task state > 0), the *task* is suspended and will resume its context on the next call. A *task* is *done* when it terminated with the previous

[1]Even when using object-oriented languages such as C++, major parts of the programs consist of imperative blocks.

a)

```
requires "io"

task fib()
  int a = 0
  int b = 1

  for(; true; yield)
    io::printi(b)
    int t = b
    b += a
    a = t

func main()
  for(int i = 0; i < 20; i+=1)
    fib()
```

b)



c)

```
task alignForKick()
  #.....

task gotoBallAndKick()
  func kickPoseReached() : bool
    return #determine aligned for kick

  for(; input.ball.position.abs() > 150; yield)
    Motion::walkTo(input.ball.position)

  for(; !kickPoseReached(); yield)
    if(input.ball.position.abs() > 250)
      yield TaskState::failure
    alignForKick()

  for(; true; yield)
    Motion::kick()
    if(taskState(Motion::kick) == TaskState::done)
      break
```

Fig. 1. a) Using a *b-script task* to generate the Fibonacci sequence. b) A simple hierarchy of behaviors which serves as comprehensible main example for the description of *b-script*. The main task of playing a ball is decomposed into simpler tasks such as kicking and walking. Each task can be considered as a single behavior. c) The according *b-script* code to this example.

call and will be restarted with the next call. A *task* can also be in *failure* state (any number < 0), which also results it to be restarted with the next call. Besides the implicit *task* state handling (termination $\rightarrow$ *done*, yield $\rightarrow$ $running$), the *yield* statement has an optional parameter that allows explicitly setting the task state, e. g. *yield TaskState::failure*. The state of a *task* can by retrieved by its caller (or any other *task* or function) with the expression *taskState(<taskIdentifier>)*.

Thus, *task states* can be used to indicate the internal state of a *task* – e. g. termination or errors – that can be retrieved by other *tasks* and functions in order to react on them (cf. Fig. 1c).

D. Implicit Context Handling

Hierarchical finite state machine implementations, such as XABSL [2], use implicit context management. The basic "scheduling" algorithm used in XABSL is to reset the context of a state machine if the state machine transitioned to a target state in the previous call, or it was not called at all during the previous execution cycle. Otherwise, the previous context will be continued. *b-script* uses the same approach for the *task* contexts. As mentioned in Sect. III-C, *tasks* will reset their context at a call if their task state is either *TaskState::done* or *TaskState::failure*. Furthermore the *task* will reset its context if it was not called in the previous execution cycle. A nice side effect of this behavior is that a *task* even continues its context if it was called from different callers consecutively, i. e. when the same sub-behavior is required by different successive higher level behaviors, it is simply continued.

These specializations provide a handy and clean way to handle *tasks* without any "overhead". From a callers point of view, they behave exactly like functions. In fact, one can turn a *b-script* function into a *task* without changing the invocation (assuming the parameters are the same). The only difference to a regular function is the use of the *yield* statement. Figure 1c depicts how to specify a simple *gotoBallAndKick* skill using a *task*.

IV. Composing Tasks to Describe Behaviors

The *tasks* introduced in Sect. III can be used to describe reactive low-level behaviors as well as composed and deliberatetive higher level behaviors in an intuitive way. In the domain of robot soccer, executing a certain kick or head control (to control the robot's gaze) are common low-level *tasks*, whereas *gotoBallAndKick* is a typical composed high level *task*. Figure 2 shows a more complete implementation of a *gotoBallAndKick* skill including the two low-level *tasks* mentioned.

In general, the *tasks* will be called like functions and execute one step, i. e. generate the next action based on the actual state, and suspend again. In terms of usual generators, this could be seen as a dynamic iterator executing an action instead of *yielding* a value. As Fig. 2 illustrates, this can be used to easily describe behavior components in a clean way, where in particular the connectedness of the code fragments is obvious because it follows the common procedural control flow.

In particular in the domain of robotics, special cases are often necessary to realize certain features. Using finite state machines implementing a special case always results in adding a new state. This often leads to multiple states that only differ slightly from each other and are inconvenient to maintain. Using *tasks*, we can easily add special cases. Most likely this can be realized using an *if...else...* statement at a certain position in the code. Such a special case is, e. g., shown in Fig. 2c in lines 10-13. A finite state machine would have to have two distinct states to realize that the head control is depending on the distance to the ball.

Obviously, this approach is suited best to specify more or less sequential behaviors. The practical use of this system showed that most non-sequential situations can be decomposed into hierarchical components, which can be described properly with this approach. As in all procedural languages, we can also emulate finite state machines if they are actually needed. Since we can use usual procedural programming features, the control flow, and local variables to maintain the "state", we can easily specify customized state-based general automata. In addition, we can integrate further features such as, e. g., the use of utility functions, neural networks, or other high as well as low-level decision making components.

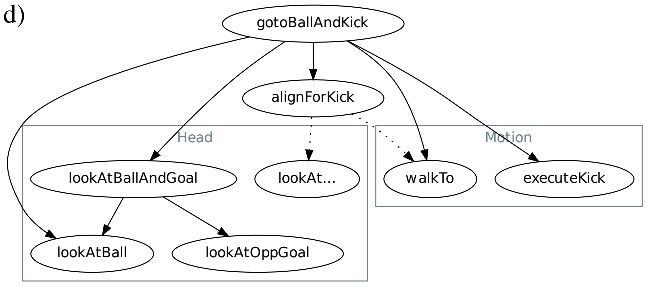
a)
```
task executeKick()
  #set output signal until motion agent reacts
  for(; input.executedMotion != MotionType::kick; yield)
    output.motion = MotionType::kick

  #reset output signal and wait until kick is finished
  for(; input.executedMotion == MotionType::kick; yield)
    output.motion = MotionType::stand
```
b)
```
task lookAtBallAndGoal()
  float ballRatio = 0.75

  int startTime = input.time
  for(; true; yield)
    float progress = ((input.time - startTime) % 2000) / 2000.0

    if(progress < ballRatio)
      lookAtBall()
    else
      lookAtOppGoal()
```
c)
```
task gotoBallAndKick()
  func kickPoseReached() : bool
    return #determine aligned for kick

  int ballNearDist = 100
  int ballFarDist = 300

  for(; input.ball.position.abs() > ballNearDist; yield)
    Motion::walkTo(input.ball.position)
    if(input.ball.position.abs() < ballFarDist)
      Head::lookAtBall()
    else
      Head::lookAtBallAndGoal()

  for(; !kickPoseReached(); yield)
    if(input.ball.position.abs() > ballFarDist)
      yield TaskState::failure
    alignForKick()

  for(; true; yield)
    Motion::executeKick()
    Head::lookAtBall()
    if(taskState(Motion::executeKick) == TaskState::done)
      break
```
d)


Fig. 2. A cross section of a gotoBallAndKick skill: a) & b) executeKick from the *Motion* module and lookAtBallAndGoal from the *Head* module are common low level tasks c) gotoBallAndKick is a common composed high level task, *Motion::walkTo* includes obstacle avoidance d) shows the visualized hierarchy of the behaviors used to describe the gotoBallAndKick *task*.

```
#include "../Util/b-script/src/bsNativeModule.h"
#include "bs_Math.h"

BS_MODULE(Math,
  registerClass<Vector2<> >(module, "Vector2f", "Vector2<>")
    .var("x", &Vector2<>::x)
    .var("y", &Vector2<>::y)
    .func("abs", &Vector2<>::abs)
    .func("_plus_", &Vector2<>::operator+, "operator+")
    .func("_min_", (Vector2<> (Vector2<>::*)(const Vector2<>&) const)
                  &Vector2<>::operator-, "operator-")
    ;

  registerFunction(module, "fabs", (float(*)(float))&fabs, "fabs");
```

Fig. 3. A code snippet declaring a C++ *b-script* module. The C++ class *Vector2<>* gets registered as *Math::Vector2f* including its member variables x and y as well as a member function and two operators. Furthermore a function gets registered. The types and signatures of the registered classes and functions will be deduced automatically. Each *register...* call takes the predefined *module* variable as well as the *b-script* identifer, the C++ pointer and optionally a C++ identifier (for the C++ code generation) as parameters.

V. B-SCRIPT - LANGUAGE & FEATURE OVERVIEW

A. Syntax

The syntax of *b-script* is mainly a mixture of Python and C++ syntax. It uses pythonic indentation-based block delimiting while using a C++ like syntax for *if* and *for* statements. This provides a common compact high level syntax with good code readability, which should be familiar and intuitive to most programmers.

Many domain-specific languages use special syntactical and semantic constructs (e. g. Colbert [11] and UrbiScript [13]). To ensure easy migration, *b-script* does not use any specialized constructs. At the same time, this should encourage to exploit the capabilities of usual procedural programming constructs extensively to gain specialized solutions.

B. Language Overview

In *b-script*, each file declares a module similar to Python or Lua. Dependencies to other modules can be declared using the *requires* statement at the top of a module. Each module consists of *functions* and *tasks*. This makes the language very modular and enables comfortable parallel development of huge behaviors in teams without interfering with each other.

b-script implements a minimalistic but sufficient subset of common procedural programming statements. This makes the language easy to learn and understandable for non-experts (the examples in this paper demonstrate almost all features).

C. C++ Integration

b-script includes an object-oriented interface to integrate C++ functions and classes similar to *Boost.Python* [18]. This makes embedding *b-script* into existing robot architectures easy and comfortable. Furthermore it provides a powerful way to extend the language itself since modules can be written in C++ and used in *b-script*. Notable is that *b-script* does not have any support for declaring objects. All objects and types available are C++ objects registered through the C++ interface (this even includes basic data types such as integers, floats, and Booleans). Figure 3 shows an example declaration of a C++ *b-script* module.

Scripts written in *b-script* can either be executed using an interpreter or be compiled to C++ code. The generated C++ code defines a C++ *b-script* module, which can be loaded by the *b-script*Engine and be executed very efficiently.

D. System Overview - Input/Output - Communication

In general, behavior systems are using input values provided by the robot software and produce output values that will be returned to the robot software. The input values usually are pre-processed sensor readings and world models built on top of these sensor readings. The output values normally are actuator commands or high-level commands that will be post-processed by the robot software to generate actuator commands.

As such a behavior system, *b-script* can be integrated into any robot control architecture that provides the world model

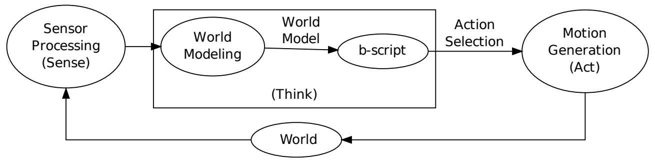


Fig. 4. System overview of a typical robot control software following the Sense-Think-Act cycle using *b-script*. The surrounding robot software provides the world model as input and receives the action selection in return.

Application	#Tasks	#Func.	#regClasses	#SLOC
RE GO2011	37	20	16	1543
RE RC2011	56	35	16	2615
BH RC2011	46	33	26	1630

Fig. 5. Statistics of RoboCup behaviors written in *b-script*. Legend: RE = RoboEireann, BH = B-Human, GO2011 = GermanOpen2011, RC2011 = RoboCup 2011, regClasses = registered classes, SLOC = source lines of code

as input data and generates actuator commands from *b-script*'s action selection. Figure 4 depicts a system overview of a robot control software using *b-script* as behavior system.

To access the input and output structures, *b-script* has two global variables declared and assigned through the *b-script*Engine. These variables are arbitrary data structures registered through the C++ interface. Both variables are accessible globally in *b-script* modules through the identifiers *input* and *output*.

Intentionally, general communication between *tasks* is prohibited. This is done to avoid obscure communication channels (e. g. global variables), which would break the clean structure and modularity of *b-script* behaviors. Therefore, *input* is a read-only variable, *output* is a write-only variable, and no further global variables can be declared. The only communication between *tasks* are their parameters and the *task states*.

VI. Application and Evaluation

A. Application

So far, *b-script* was applied in RoboCup competitions in the Standard Platform League. The RoboCup provides several complex and dynamic benchmark environments for all kinds of robot software and hardware systems. In the Standard Platform League, the humanoid *Nao* robot [5] by *Aldebaran Robotics* is used. Teams consisting of four autonomous robots each play soccer games against each other on a $6 \times 4 m^2$ color-coded pitch as depicted in Fig. 6. The robots mainly use their cameras with an diagonal opening angle of $58°$ to perceive the world and build world models on top of the resulting percepts. The robots use two-legged walking algorithms for locomotion. Due to the limited field of view and the bipedal locomotion, the perception, world models as well as motions are subjects of high uncertainty and noise.

The Standard Platform League team RoboEireann [19] used *b-script* to develop the behavior system for the German Open 2011 and the RoboCup 2011. Especially the behavior system used on the German Open 2011 performed very successful. RoboEireann achieved the fourth place and defeated

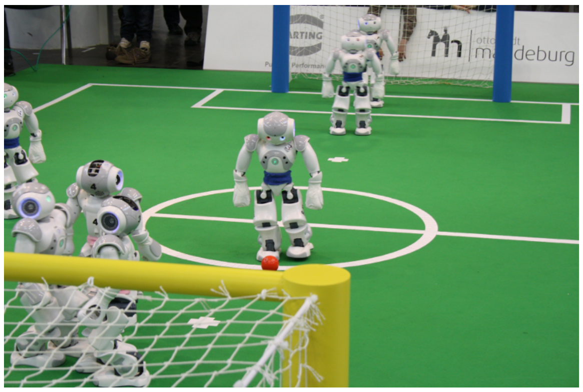


Fig. 6. RoboCup Standard Platform League: Nao robots autonomously playing a 4 on 4 soccer match in a color-coded environment.

the later vice world champion in a penalty shootout. Here, *b-script* allowed rapid and easy behavior development and adaptation to new situations even in short match timeouts without circumstances. It definitely fulfilled its contribution to the successful participation.

The reigning world champion in the Standard Platform League B-Human [20] used *b-script* to develop the penalty behavior for the RoboCup 2011. (Un)fortunately, B-Human did not have to participate in an official penalty shootout in the course of the competition. A complete integration into the B-Human software system was implemented providing full access to all sensor and world model inputs and to all actuator outputs as well as a set of *tasks* providing high level access to all outputs, such as those depicted in Fig. 2a. This base system was also used to implement test match opponents. It met the demands of the B-Human software standards and performed well.

Figure 5 illustrates some statistics about the complexity of the *b-script* behavior systems developed by RoboEireann and B-Human.

B. Evaluation

Since many autonomous robots are running restricted hardware systems and must meet real time constraints, the runtime performance is a crucial property of behavior specification systems. To evaluate the runtime performance of *b-script*, the penalty behavior developed for the RoboCup 2011 by the team B-Human was examined. The behavior was compiled to *b-script* C++ modules and loaded as dynamic library into the *b-script*Engine. In Fig. 7, the runtime per frame is plotted. The average runtime is approximately 0.8ms when running complex behaviors. Since the system's cognitive main loop (including vision, world modeling, and the behavior) is running at 30Hz, this meets the real time requirement and leaves enough computing time for other, computationally more expensive software components.

The test was run on a *Nao V3.3* robot running the regular B-Human RoboCup code. The *Nao V3.3* runs a Linux system on a x86 AMD GEODE 500MHz CPU and 256MB SDRAM.

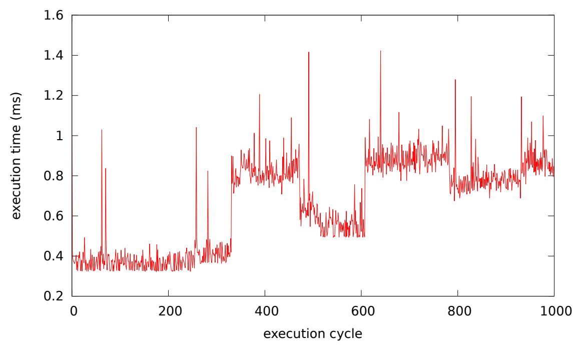


Fig. 7. The timings of the B-Human 2011 RoboCup penalty behavior running in a Nao robot. The different levels are caused by the different complexities of the actual behaviors (i. e. cycles 0-350 set state: stand & look around, cycles 350-450 approaching the ball for a short kick, cycles 450-600 executing a short kick, cycles 600-780 approaching the ball for the final kick, 780-900 executing the final kick). The short peaks are artifacts of the time measurement method on the Nao robot.

VII. Conclusions and Future Works

A. Conclusions

In this paper, we presented *b-script*, a domain specific language based on hierarchical generators to describe complex robot behaviors in an intuitive and clean way. Through its procedural design it is very flexible, powerful, and allows specifying robot behaviors very precisely. The object-oriented C++ interface provides a handy way to easily extend the language and embed it into existing robot software systems. The scripting-like design of the language gives a commonly known look and feel and makes the migration for programmers very easy. Furthermore the modular structure allows working in large teams and provides the opportunity to reuse code in different applications. Thus it seems to be an elegant alternative to commonly used hierarchical finite state machines to script robot behaviors.

The application in the RoboCup Standard Platform League proved that it allows convenient behavior development in large-scale and that it is suitable to be used on robots with restricted hardware resources. Even though *b-script* was developed to be used in the RoboCup, it contains no domain specific features and can be used in other scenarios as well.

The source code as well as the complete *b-script* B-Human 2011 penalty behavior and further documentation is available at `http://www.informatik.uni-bremen.de/~jeffry/b-script`.

B. Future Works

The actual *b-script* implementation distinguishes between *functions* and *tasks*, where *tasks* do not support return values. Since *functions* are actually specialized *tasks* (i. e. *tasks* that do not *yield* and thus terminate on each call), these two concept could be unified. Therefore, a *task* would need to support return values. Furthermore, adding return values to *tasks* would give new possibilities to use *tasks*, namely to outsource decisions (and their states) and thus simplifying the caller's code, resulting in an even more modular design of behaviors.

VIII. Acknowledgments

The authors of this paper would like to especially thank Sonja Stüdli for working with *b-script* from the beginning and giving great feedback and bug reports as well as the B-Human team and the RoboEireann team for providing the software base for the applications in RoboCup. This work was partially funded by the SFB/TR 8 Spatial Cognition Research Center of the German Research Foundation (DFG).

References

[1] R. Arkin, *Behavior-based robotics*. The MIT Press, 1998.

[2] M. Lötzsch, M. Risler, and M. Jüngel, "XABSL - A pragmatic approach to behavior engineering," in *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, Beijing, China, 2006, pp. 5124–5129.

[3] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research*, vol. 20, pp. 379–404, 2003.

[4] C. Dornhege, M. Gissler, M. Teschner, and B. Nebel, "Integrating symbolic and geometric planning for mobile manipulation." in *Proceedings of the 2009 IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR)*, Denver, CO, USA, 2009.

[5] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier, "The NAO humanoid: a combination of performance and affordability," *CoRR*, vol. abs/0807.3223, 2008.

[6] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara, "RoboCup: A challenge problem for ai," *AI Magazine*, vol. 18, no. 1, pp. 73–85, 1997.

[7] "RoboCup Standard Platform League web site," 2011, http://www.tzi.de/spl/.

[8] S. Tousignant, E. Van Wyk, and M. Gini, "An overview of XRobots: A hierarchical state machine-based language," in *Proceedings of The 4th Workshop on Software Development and Integration in Robotics (SBIR-IV)*, Shanghai, China, May 2011.

[9] O. Obst, "Specifying rational agents with statecharts and utility functions," in *RoboCup 2001: Robot Soccer World Cup V*, ser. Lecture Notes in Computer Science, vol. 2377. Springer, 2002, pp. 173–182.

[10] J. Murray, "Specifying agent behaviors with UML statecharts and StatEdit," in *RoboCup 2003: Robot Soccer World Cup VII*, ser. Lecture Notes in Computer Science, vol. 3020. Springer, 2004, pp. 145–156.

[11] K. Konolige, "Colbert: A language for reactive control in Sapphira," in *In KI-97: Advances in Artificial Intelligence, 21st Annual German Conference on Artificial Intelligence*, ser. Lecture Notes in Artificial Intelligence, vol. 1303. Springer, 1997, pp. 31 – 52.

[12] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 1998)*, Victoria, B.C., Canada, 1998, pp. 1931 – 1937.

[13] J.-C. Baillie, "URBI: towards a universal robotic low-level programming language," in *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005)*, Edmonton, Alberta, Canada, 2005, pp. 820 – 825.

[14] E. Bendersky, "Co-routines as an alternative to state machines," 2011. [Online]. Available: http://eli.thegreenplace.net/2009/08/29/co-routines-as-an-alternative-to-state-machines/

[15] L. Allison, "Continuations implement generators and streams," *The Computer Journal*, vol. 33, no. 5, pp. 460–465, 1990.

[16] D. Jouvin, "Continuations and behavior components engineering in multi-agent systems," in *Multiagent System Technologies*, ser. Lecture Notes in Computer Science, vol. 4196. Springer, 2006, pp. 147–158.

[17] C. McMillen, "Creating robot behaviors with python generators," 2011, not available anymore, previous address: http://colinm.org/blog/creating-robot-behaviors-with-python-generators.

[18] D. Abrahams and R. Grosse-Kunstleve, "Building hybrid systems with Boost.Python," *C/C++ Users Journal*, vol. 21, pp. 29–36, 2003.

[19] "RoboEireann::Robot Soccer Team::National University of Ireland, Maynooth," 2011. [Online]. Available: http://www.eeng.nuim.ie/robocup/

[20] "B-Human — RoboCup Standard Platform League," 2011. [Online]. Available: http://www.b-human.de