A Lua-based Behavior Engine
for Controlling the Humanoid Robot Nao

Tim Niemüller[1], Alexander Ferrein[1,2], and Gerhard Lakemeyer[1]

[1] Knowledge-based Systems Group
RWTH Aachen University, Aachen, Germany
{niemueller,gerhard}@kbsg.rwth-aachen.de
[2] Robotics and Agents Research Lab
University of Cape Town, Cape Town, South Africa
alexander.ferrein@uct.ac.za

Abstract. The high-level decision making process of an autonomous robot can be seen as an hierarchically organised entity, where strategical decisions are made on the topmost layer, while the bottom layer serves as driver for the hardware. In between is a layer with monitoring and reporting functionality. In this paper we propose a behaviour engine for this middle layer which, based on formalism of hybrid state machines (HSMs), bridges the gap between high-level strategic decision making and low-level actuator control. The behaviour engine has to execute and monitor behaviours and reports status information back to the higher level. To be able to call the behaviours or skills hierarchically, we extend the model of HSMs with dependencies and sub-skills. These Skill-HSMs are implemented in the lightweight but expressive Lua scripting language which is well-suited to implement the behaviour engine on our target platform, the humanoid robot Nao.

1 Introduction

Typically, the control software of an autonomous robot is hierarchically organised with software modules that communicate directly to the hardware on the lowest level, some more elaborated entities for, say, localisation or the object detection on a middle layer, and an action selection mechanism on top. On each of these levels the time constraints are different, meaning that modules on lower levels have shorter decision cycles and need to be more reactive than those on the higher levels. The reason is that usually, the level of abstraction increases with each layer of the software. (See e.g. [1,2] for textbooks on "classical" 3-tier architectures). The same holds for the high-level action selection. From that viewpoint basic or primitive actions are selected in coordination with the teammates and the team strategy; these are broken down to actuator commands on the lowest level over several levels of software abstraction. The term basic or primitive action hides the fact that these actions are usually on a high level of abstraction. Examples for those basic actions are dribble or attack-over-the-left-wing. Many different approaches exist for how these primitive actions will be selected. These range

from full AI planning to simply reactively couple sensor values to these actions or behaviours. Between the high-level action selection and the driver modules for the servo motors of the robot, a middle layer is required to formulate complex actions and report success or failure to the high-level control.

In this paper, we address this middle-layer and show a possibility how the gap between high-level control and the rest of the robot system can be bridged. Independent from the high-level scheme, the middle control layer for behaviours, which we call *behaviour engine*, needs to be expressive enough to hide details from the high-level control while having all the information needed to order and monitor the execution of basic actions in the low-level execution layer. This behaviour engine must thus provide control structures for monitoring the execution and facilities to hierarchically call sub-tasks etc. Moreover, it needs to be lightweight enough not to waste resources which should either be spent for the high-level decision making or for tasks like localisation. Hence we need a computationally inexpensive framework which allows for the needed kind of expressiveness. As our target platform is the standard platform Nao which has limited computational resources, the lightweight of the behaviour engine is even more important. We propose a behaviour engine which matches these criteria. The building block for our behaviour engine is a *skill*. We formalise our skills as extended Hybrid State Machines (HSMs) [3] which allow for using state machines hierarchically. Our implementation of these *Skill-HSMs* is based on the scripting language Lua [4], which is a lightweight interpreter language that was successfully used before for numerous applications ranging from hazardous gas detection systems for the NASA space shuttle to specifying opponent behaviour in computer games. We show that the combination of Lua and HSMs provide a powerful system for the specification, execution and monitoring of skills. However, by choosing a general purpose language like Lua, we do not preclude the possibility to later extend the behaviour engine. It is therefore also possible to use other skill specifications in parallel to our hybrid state machines.

The paper is organised as follows. In Sect. 2 we present the software framework FAWKES which we use for our Nao robot. In particular, we show the different control modules and how the communication between the different sub-systems takes place. One important issue is that the original motion patterns from NaoQi can be easily integrated, if desired. In Sect. 3 we define skills in terms of extended HSMs, called Skill Hybrid State Machines (SHSMs), to yield a hierarchy of skills, and distinguish between actions, behaviours, and motion patterns. Section 4 addresses the implementation of SHSMs in Lua. In Section 5 we show an example state machine for the stand-up motion. We conclude with Section 6.

2   Fawkes and the Nao

In this section we briefly introduce the ideas behind and key components of the FAWKES software framework and describe our target platform, the Nao. We then go over to describing our instantiation of FAWKES on the Nao platform.

2.1   The Humanoid Robot Nao

The Nao [5] is the new biped platform for the STANDARD PLATFORM LEAGUE by the French company Aldebaran. The league is the successor of the Sony Aibo league. The 58 cm tall robot has 21 degrees of freedom, and is equipped with two CMOS cameras providing 30 frames per second in VGA resolution, it has some force sensitive resistors, a gyroscope and an accelerometer as well as sonar sensors. The CPU is an AMD Geode running at 500 MHz accompanied by 256 MB of RAM. To communicate with the platform, Wi-Fi (IEEE 802.11b/g) and Ethernet are available. It comes with a closed source software framework called NaoQi which is the only way to access the robots actuators and sensory. It also provides basic components for image acquisition, walking, and actuator pattern execution.

2.2   Fawkes in a Nutshell

The FAWKES robot software framework [6] provides the infrastructure to run a number of plug-ins which fulfil specific tasks. Each plug-in consist of one or more threads. The application runs a main loop which is subdivided into certain stages. Threads can be executed either concurrently or synchronised with a central main loop to operate in one of the stages. All threads registered for the same stage are woken up and run concurrently. Unlike other frameworks such as Player or Carmen we pursue a more integrated approach where plugins employ threads running in a single process exchanging data via shared memory instead of message passing. Currently, we use the software in the ROBOCUP@HOME LEAGUE for domestic service robots as well as in the MIDDLE SIZE LEAGUE and STANDARD PLATFORM LEAGUE for soccer robots. The framework will soon be released as Open Source Software.

Blackboard. All data extracted and produced by the system and marked for sharing is stored in a central blackboard. It contains specified groups of values which are accessed with a unified interface. An example is an object position interface, which provides access to position information of an object, like the position of the robot itself or the ball on the field. These interfaces can be read by any other plug-in to get access to this information. Commands are sent to the writer via messages. Message passing eliminates a possible writer conflict. Opposed to IPC (see `http://www.cs.cmu.edu/~{}ipc/`) as used by Carmen data is provided via shared memory, while messaging is used only for commands.

Component-based Design. FAWKES follows a component-based approach for defining different functional blocks. A component is defined as a binary unit of deployment that implements one or more well-defined interfaces to provide access to an inter-related set of functionality configurable without access to the source code [7,8]. With the blackboard as a communication infrastructure, system components can be defined by a set of input and output interfaces. This
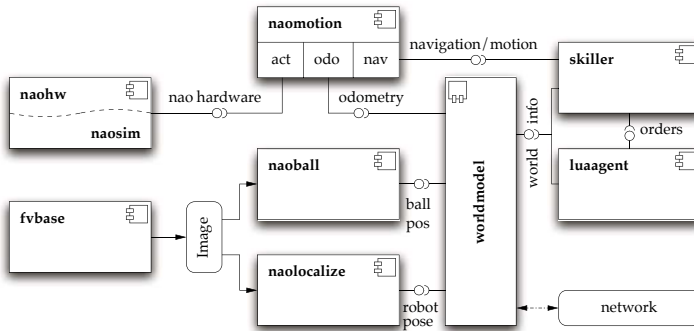
Fig. 1. Component configuration for FAWKES on the Nao robot

allows for easily replacing a component as long as the replacement component requires and provides the same sets of interfaces.

2.3 Running Fawkes on the Nao

Figure 1 shows the component configuration that has been implemented for the Nao robot platform. On the left-hand side one can find the *naohw/naosim* component. The *naohw* and *naosim* plug-ins both provide access to the underlying robot hardware, on the real robot and in a simulation environment. The *fvbase* plug-in provides access to the camera and distributes the acquired image via a shared memory segment. The *naoball* and *naolocalize* plug-ins use this image to extract information about the robot and ball position. This data together with other acquired information is processed in the *worldmodel* component. This component merges different sources of information (local extraction components and information received from other robots via Wi-Fi) to provide a unified world model to all higher level components. The *naomotion* component deals with robot locomotion and odometry calculation. It also includes a navigation component for path planning. The unified world model is used by the skill execution run-time (*skiller*) and the *luaagent* component which we are going to describe in more detail below.

NaoQi Integration. Given the current closed nature of the Nao robot platform it is essential to integrate FAWKES with NaoQi to gain access to the hardware. Beyond plain hardware access it is desirable to provide access to other NaoQi functionality. An integration module exists for this purpose that integrates FAWKES with NaoQi. For instance the actuator sub-component of *naomotion* can be provided via the integration of Aldebaran's NaoQi motion engine.

3 The Behaviour Engine and Skill Hybrid State Machines

In this section we define the behaviour engine. As stated in the introduction, the whole behaviour system can be seen as a layered system, just like a hierarchically

structured system for the overall control of a robot. In Sect. 3.1 we therefore distinguish between *low-level control*, *behaviours*, and the *agent* as different levels of the behaviour system, before we define our behaviour engine in Sect. 3.2.

3.1   Low-Level Control, Behaviours and Agents

To model a robot's behaviour multiple different levels of control can be distinguished. On the lowest level we have tight control loops which have to run under real-time or close-to-real-time constraints, for instance, for generating joint patterns to make the robot walk. On the highest level we have an agent which takes decisions on the overall game-play or on the team strategy, possibly communicating and coordinating actions with other robots.

Especially when more elaborated approaches for designing an agent like planning or learning are used, it is beneficial to not only have the very low-level actions like "walk-a-step", but also more elaborate reactive behaviours like "search-the-ball" or "tackle-over-the-right-wing". This reduces the computational burden for the agent tremendously. Additionally it is easier to develop and debug small behaviour components. In the following we will clarify what we understand with skills and show the different levels of behaviours as three tiers.

Definition 1 (Behaviour levels)

Level 0: Low-level Control Loops *On this level modules run real-time or close-to-real-time control loops for tasks like motion pattern generation or path-planning and driving.*

Level 1: Skills *Skills are used as reactive basic behaviours that can be used by the agent as primitive actions.*

Level 2: Agent *An agent is the top-most decision-making component of the robot that makes the global decisions about what the robot is doing and the strategic direction.*

At each level, behaviours can only be called by other behaviours which are from the same or a higher level.

According to this understanding of a tiered behaviour system, the FAWKES software framework was designed. Figure 2 shows the organisation of the FAWKES software stack. On the lowest level modules for sensor acquisition, data extraction, and other low-level control programs are located. On the top, the agent is making the overall decision on the behaviour. In between lies a reactive layer which provides basic actions to the agent. For this it uses information of the low-level modules to make local decisions for a specific behaviour, e.g. when approaching a ball the walking direction might need to be adjusted for ball movements. From that it creates commands for the low-level actuator driving components like locomotion.

3.2   Behaviour Engine

Against the background of a deliberative approach, one has specific expectations what the behaviour engine has to provide, which nevertheless can be applied for
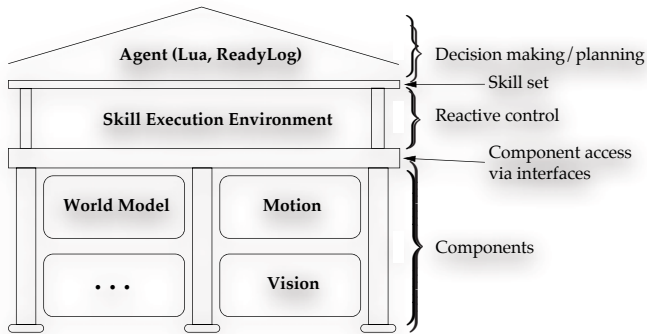
Fig. 2. The Fawkes Software Stack

reactive decision making. On the higher level strategic planning, we need primitive actions that we can model and use for plan generation. For efficiency reasons these primitive actions – skills – are written following a reactive paradigm. The skills are small reactive execution entities which have a well-defined purpose, like "go-to-position (x, y)" or "intercept-ball". When a skill is called it will try to fulfil its purpose, or report a failure if that is not possible, e.g. the ball is no longer visible while intercepting it. A skill cannot deliberately switch to another skill. This decision is the task of the higher level agent program. However, it can call another skill as part of itself, for instance the *intercept* skill has to call the *goto* skill for the movement towards the ball. But the intercept would not decide to change the behaviour to *search-ball* when the ball is not visible. While changing the active skill could make sense for a field player, a defender might better go back to the defending position. Therefore the decision to switch skills should be made by the agent controlling the overall behaviour. Skills can thus be seen as execution entities which make only local decisions, not global game-play decisions. These skills need a particular programming and run-time environment – the behaviour engine. According to Definition 1 the behaviour engine is located at level 1 in our behaviour hierarchy. From the initial proposition that skills are reactive execution entities which accomplish simple task, i.e. the primitive actions from a higher level perspective, state machines are an obvious choice for modelling the behaviour. As we need continuous transitions between states, we selected hybrid state machines (HSMs) as the model of our choice. A HSM is a finite state machine which allows for state transitions following differential equations (flow conditions) on the one side, and logical expressions (jump conditions), on the other side. Jump conditions are represented by the inequalities, while the flow conditions are stated by the differential equations (see e.g. [3]).

UML state charts combined with hybrid automata have been used by Stolzenburg et al. [9,10] to model higher level reactive behaviours. The focus of their work, however, was on level 2 of our behaviour hierarchy, where they coordinated

the multi-agent behaviour with HSMs. Here, we concentrate on a formal model
for hierarchical skills on level 1.

We want to use HSMs to model skills as reactive execution entities. More
complex skills often use simpler skills. Therefore we want to provide an efficient
way for re-using a skill, which avoids the construction of a state machine that
includes both, the complex skill behaviour and all of the internal details of the
included simple skill. To achieve this we extend the HSMs [3]. The behavior en-
gine should be applicable to multiple platforms and domains. We expect that the
available skills will depend on the combination of a particular platform and do-
main. Therefore we give the following definitions. The domain describes the area
of operation and determines the tasks to be executed. The platform describes
the used robot system.

Definition 2 (Skill Space). *The combination of a platform $\mathcal{P}$ and a domain
$\mathcal{D}$ with regard to skills is called* skill space $(\mathcal{P}, \mathcal{D})$ *for platform $\mathcal{P}$ and domain $\mathcal{D}$.*

Definition 3 (Set of Skills). *The set $\mathcal{K}_{(\mathcal{P},\mathcal{D})}$ is called the* set of skills *for the
skill space $(\mathcal{P}, \mathcal{D})$. It denotes the set of available skills for a particular skill space.*

Definition 4 (Skill Hybrid State Machine (SHSM))

$$S = (G, X, D, A, jump, flow, exec, \mathcal{K}_{(\mathcal{P},\mathcal{D})}) \tag{1}$$

Final and failure state. *The graph $G = (Q, E)$ has only two valid exit states
$Q_{\mathrm{exit}} = \{q_{\mathrm{final}}, q_{\mathrm{failure}}\}$.*

Control graph. *A finite directed multi-graph $G = (Q, T)$, where $Q = Q_U \cup Q_{\mathrm{exit}}$
are the states (vertices) with Q_U being the user defined states and T are the
transitions (edges).*

Dependencies. *For hierarchical definition of a skill existing skills can be re-used.
These used skills are called* dependencies *of skill S. Skills that are used in
the current skill are called direct dependencies, skills that are used in direct
dependencies or their dependencies are called indirect dependencies. A skill
may not depend directly or indirectly on itself. For this we define a set $D \subseteq
\mathcal{K}_{(\mathcal{P},\mathcal{D})} \setminus S$ of dependencies. Let $D_S \subseteq D$ be the set of skills that the skill
S directly depends on. Then the function $\delta : \mathcal{K}_{(\mathcal{P},\mathcal{D})} \to \wp(\mathcal{K}_{(\mathcal{P},\mathcal{D})} \setminus S)$ with
$\delta(S) = D_S \cup \{\delta(d) \mid d \in D_S\}$ gives a set of all direct and indirect dependencies
of S and $S \notin \delta(S)$. This can be represented as a dependency graph.*

Execution Function. *A skill is executed with the* exec *function. It assigns values
to some variables $x \in X$ and runs the state machine by evaluation of the
jump conditions of the current state, possibly leading to a state change. It
is defined as $\mathrm{exec}(x_1, \ldots, x_n) \to \{\mathrm{final}, \mathrm{running}, \mathrm{failure}\}$ with $x_i \in X$. The
return value depends on the current state after the evaluation.*

Actions. *For the execution of lower-level behaviors and other SHSMs we de-
fine a set A of actions. An action $a \in A'$ is a function $a(x_1, \ldots, x_n) \to
\{\mathrm{running}, \mathrm{final}, \mathrm{failure}\}$ with $x_i \in X$ that executes a lower-level system be-
havior (on a lower behavior level). The set $K = \{\mathrm{exec}_d \mid d \in D\}$ is the set of
execution functions of dependency skills (on the same behavior level). The
set of actions is then defined as $A = A' \cup K$.*

Action Execution. *For each state $q \in Q$ we define a set $E_q \subseteq A$ of actions. Each action is executed when the state is evaluated. The set E_q may be empty.*

Before we illustrate the definition of skill hybrid state machines in Sect. 5 with an example skill from the humanoid robot Nao in detail, we address the implementation of SHSMs in Lua in the next section. For now, it is sufficient to note that skills are special hybrid state machines that can hierarchically be called by other skills. We hence model actions and sub-skills as a way to interact with the actuators of the robot and to easily call other skills. To avoid specification overhead sub-skills are defined as functions. Rather then integrating another skill's state machine when it is called into the current caller's state machine it is sufficient to call the encapsulating k-function and define appropriate jump conditions based on the outcome. Dependencies are defined as a way to avoid cyclic call graphs when a skill calls another skill.

4   Implementing Skill Hybrid State Machines in Lua

4.1   Lua

Lua [4] is a scripting language designed to be fast, lightweight, and embeddable into other applications. These features make it particularly interesting for the Nao platform. The whole binary package takes less then 200 KB of storage. When loaded, it takes only a very small amount of RAM. This is particularly important on the constrained Nao platform and the reason Lua was chosen over other scripting languages, that are usually more than an order of magnitude larger [11]. In an independent comparison Lua has turned out to be one of the fastest interpreted programming languages [11,12]. Besides that Lua is an elegant, easy-to-learn language [13] that should allow newcomers to start developing behaviours quickly. Another advantage of Lua is that it can interact easily with C/C++. As most robot software is written in C/C++, there exists an easy way to make Lua available for a particular control software.

Other approaches like XABSL [14] mandate a domain specific language designed for instance for the specification of state machines. This solves to some extent the same problems, easy integration, low memory foot print and easy development. But it also imposes restrictions in terms of the expressiveness of the language. By using a general purpose language one can easily experiment with different approaches for behaviour formulations. Using parsing expression grammars implemented in Lua (LPEG [15]), XABSL files could even be read and executed within the Lua behaviour engine.

Lua is popular in many applications. Especially in the computer game sector Lua has found a niche where it is the dominant scripting language. It has been used to develop game AI and extension modules for games. And even in RoboCup applications, Lua showed its strength as programming language before [16,17].

Integration of Lua into Fawkes. As FAWKES uses a plug-in strategy for integrating software modules, it was particularly easy to develop a Lua plug-in

for FAWKES making use of the C/C++ interface. As on the level of the behaviour engine required information from the low-level control system are stored in the blackboard, access to the blackboard from Lua needed to be guaranteed. Wrappers for accessing C++ code from Lua can be generated automatically via tolua++ (cf. http://www.codenix.com/tolua/ for the reference manual). Since interfaces are generated from XML descriptions input for automated wrapper generation by means of tolua++ can easily be created. With this, data can be read from and written to the blackboard, and messages can be created and sent from Lua. With this access to all the robot's information about the current world situation is accessible and commands to any component in the framework can be sent.

The agent calls skills by forming a Lua string which calls the skills as functions. The Lua integration plug-in will create a sandbox and execute the Lua string in that sandbox. The sandbox is an environment with a limited set of functions to be able to apply policies on the executed string. This could be preventing access to system functions that could damage the robot or only providing certain behaviours.

4.2   Implementing Skill Hybrid State Machines

We chose SHSMs to model the robot's different behaviours. Each skill is designed as an HSM. The core of the implementation is the Skill-HSM to which states are added. Each state has a number of transitions, each with a target state and a jump condition. If the jump condition holds the transition is executed and the target state becomes the active state. The HSM has a start state which is the first active state. When a skill is finished or stopped by the agent, the state machine is reset to the start state.

The state machine is executed interleaved. That means that in an iteration of the main loop all transitions of the active state are checked. If a jump condition holds the appropriate transition is executed. In this case transitions of the successor state are immediately checked and possibly further transitions are executed. A maximum number of transitions is executed after which the execution is stopped for this iteration. If either a state is reached where no jump condition fires or the maximum number of transitions is reached, the execution is stopped and continued only in the next iteration, usually with fresh sensor data and thus with possibly different decisions of the jump conditions.

4.3   Tools for Developing Behaviours

An often underestimated aspect is the need for tool support when developing behaviours. Lua has the great advantage that it comes with automatic memory management and debugging utilities. Hence, we can focus on the development of skills in the behaviour engine. Instead of going along the lines of programming the behaviours graphically (as RoboCupers usually are experienced programmers), skills need to be coded. Nevertheless, we support the debugging process with visual tools, displaying the state transitions and execution traces on-line.
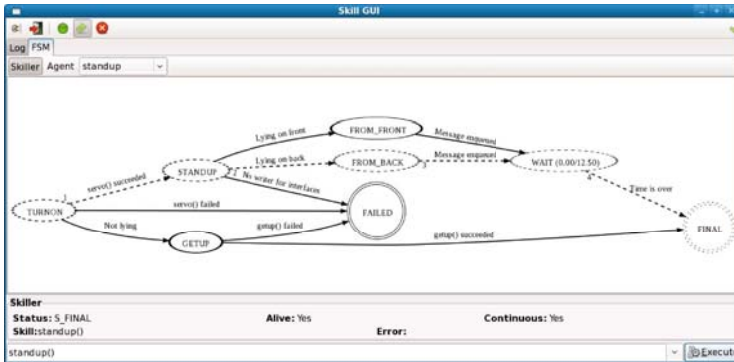
Fig. 3. SkillGUI: a GUI to execute, visualise and debug skills

The behaviour is visualised as a graph with states as nodes and transitions as edges. The developed implementation in Lua allows to create the HSM graph at run-time. From the formulation of the behaviour as HSM in Lua the graph can be directly generated. The Graphviz [18] library is used to generate the graphical representation and display to the user. In Figure 3 the visualisation of the state machine of a stand-up behaviour is shown. The dotted node marks the currently active node, the dashed lines are states and transitions which have been passed in the current run of the state machine. The numbers give the order of the transition sequence, the trace. By this trace one can follow what happened in the state machine even if the transitions happen very fast.

5   The Stand-Up State Machine in Lua

To illustrate the definition of SHSM, we show an example state machine of a stand-up behaviour of the Nao. In Listing 1.1 an excerpt from the code of the stand-up skill is given. The complete skill is visualised in Figure 3. The code shows the state machine for standing up from lying on the back. Of particular interest are line 2, where the dependencies are defined, and the following lines, which define the required blackboard interfaces. Lines 8-10 instantiate the states, line 10 specifically adds a sub-skill state which will execute and monitor the getup skill (which makes the robot stand from a sitting position). In Lines 12-13 a transition is added which is executed when the robot is standing or sitting on its feet. Line 15 adds a transition that is executed if the robot is lying on its back. Lines 17-20 show the code to order an execution in the low-level base system via the blackboard. Parts like the execution of stand-up from lying on the front and waiting for the action to finish have been omitted for space reasons.

In our experiments, besides verifying the correctness of the state machine and its monitoring, we analysed the run-time of the system. As we stated in the introduction, a behaviour engine must be lightweight, not wasting any resources. On average, the state machine takes 1 ms till the next command is executed, 3 ms with the debugging graph visualised.

Listing 1.1. Standup Skill HSM in Lua (excerpt)

```
1   fsm                 = SkillHSM:new{name=name, start="STANDUP"}
2   depends_skills      = {"servo", "getup"}
3   depends_interfaces = {
4      {v = "naomotion", type = "HumanoidMotionInterface"},
5      {v = "naohw",     type = "NaoHardwareInterface"}
6   }
7
8   fsm:new_jump_state("STANDUP")
9   fsm:new_jump_state("FROM_BACK")
10  fsm:new_jump_state("GETUP", getup, FINAL, FAILED)
11
12  STANDUP:add_transition(GETUP,
13          "naohw:accel_x() >= -35 and naohw:accel_x() <= 35")
14
15  STANDUP:add_transition(FROM_BACK, "naohw:accel_x() < -35")
16
17  function FROM_BACK:init()
18     naomotion:msgq_enqueue_copy(
19       naomotion.StandupMessage:new(naomotion.STANDUP_BACK))
20  end
```

Other skills that have been implemented include walking to a certain global or relative position, controlling servos (e.g. to move the head while walking), searching for the ball, tracking the ball with the head or intercepting the ball.

In addition to the behaviour engine we have used extended HSMs and Lua to implement a simple agent program to control the overall game-play (*luaagent* component in Figure 1). In this configuration the basic actions for the HSM are the skills provided by the behaviour engine (*skiller* component). By keeping these two layers separate and not writing the agent as a top-level skill the agent could easily be replaced, for instance by a more powerful planning component.

6   Conclusion

In this paper we proposed a behaviour engine for the humanoid standard platform Nao. For this purpose, we introduced a three tier architecture for the behaviour system. The behaviour engine we propose here is located in the middle layer and interfaces between the high-level decision component and the execution of behaviour patterns. This means that the behaviour engine particularly has the task to monitor the execution of behaviours, and report success or failure to the high-level decision maker. Central for our behaviour engine is the skill, which is a piece of code which monitors the low-level execution and reports to the high-level agent. As a formal model for skills we decided for hybrid state machines. We extended these state machines to allow for skill hierarchies. A skill is therefore formalised by a skill hybrid state machine. The implementation of the behaviour engine was done in the lightweight scripting language Lua, which can be easily integrated into a robot control architecture. We use this behaviour engine successfully on the new standard biped platform Nao. In particular it is important to note that the motion patterns via NaoQi can be integrated easily. This makes our approach interesting as an extension of the Nao software architecture. For future work, we want to use the inherent features like events and

multi-graphs, which come with HSMs and could be used to model multi-agent behaviour, for example for cooperative team play on the agent level as well.

References

1. Bekey, G.A.: Autonomous Robots: From Biological Inspiration to Implementation and Control. MIT Press, Cambridge (2005)
2. Murphy, R.R.: Introduction to AI Robotics. The MIT Press, Cambridge (2000)
3. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings Logic in Computer Science 1996, July 1996, pp. 278–292. IEEE, Los Alamitos (1996)
4. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua - An Extensible Extension Language. Software: Practice and Experience 26(6), 635–652 (1999)
5. Aldebaran Robotics (2008), http://www.aldebaran-robotics.com/
6. Niemueller, T.: Developing A Behavior Engine for the Fawkes Robot-Control Software and its Adaptation to the Humanoid Platform Nao. Master's thesis, Knowledge-Based Systems Group, RWTH Aachen University (to appear, 2009)
7. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Orebäck, A.: Towards Component-Based Robotics. In: Proc. IROS 2006, August 2005, pp. 163–168 (2005)
8. Collins-Cope, M.: Component Based Development and Advanced OO Design. Whitepaper, Ratio Group Ltd. (2001)
9. Furbach, U., Murray, J., Stolzenburg, F.: Hybrid Multiagent Systems with Timed Synchronization – Specification and Model Checking. In: Dastani, M.M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (eds.) ProMAS 2007. LNCS (LNAI), vol. 4908, pp. 205–220. Springer, Heidelberg (2008)
10. Arai, T., Stolzenburg, F.: Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. In: Proc. AAMAS 2002. ACM Press, New York (2002)
11. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: The Evolution of Lua. In: Proceedings of History of Programming Languages III, pp. 2-1–2-26. ACM, New York (2007)
12. The Debian Project: The Computer Language Benchmarks Game, http://shootout.alioth.debian.org/ (retrieved January 30, 2009)
13. Hirschi, A.: Traveling Light, the Lua Way. IEEE Software 24(5), 31–38 (2007)
14. Loetzsch, M., Risler, M., Jungel, M.: XABSL - A Pragmatic Approach to Behavior Engineering. In: Proc. IROS 2006, pp. 5124–5129 (2006)
15. Medeiro, S., Ierusalimschy, R.: A parsing machine for PEGs. In: Proceedings of the 2008 Symposium on Dynamic Languages, pp. 1–12. ACM, New York (2008)
16. Kobayashi, H., Ishino, A., Shinohara, A.: A framework for advanced robot programming in the robocup domain - using plug-in system and scripting language. In: Proc. IAS-9, pp. 660–667 (2006)
17. Hester, T., Quinlan, M., Stone, P.: UT Austin Villa 2008: Standing On Two Legs. Technical report, Department of Computer Sciences, The University of Texas, Austin (2008)
18. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz – Open Source Graph Drawing Tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 483–597. Springer, Heidelberg (2002)