Object-oriented Model-based Extensions of Robot Control Languages

Armin Müller, Alexandra Kirsch, Michael Beetz

Object-oriented Model-based Extensions of Robot Control Languages

Armin Müller, Alexandra Kirsch, Michael Beetz

Informatik IX, Technische Universität München

Abstract. More than a decade after mobile robots arrived in many research labs it is still difficult to find plan-based autonomous robot controllers that perform, beyond doubt, better than they possibly could without applying AI methods. One of the main reason for this situation is abstraction. AI based control techniques typically abstract away from the mechanisms that generate the physical behavior and refuse the use of control structures that have proven to be necessary for producing flexible and reliable robot behavior. The consequence is: AI-based control mechanisms can neither explain and diagnose how a certain behavior resulted from a given plan nor can they revise the plans to improve its physical performance.

In our view, a substantial improvement on this situation is not possible without having a new generation of robot control languages. These languages must, on the one hand, be expressive enough for specifying and producing high performance robot behavior and, on the other hand, be transparent and explicit enough to enable execution time inference mechanisms to reason about, and manipulate these control programs. This paper reports on aspects of the design of RPL-II, which we propose as such a next generation control language. We describe the nuts and bolts of extending our existing language RPL to support explicit models of physical systems, and object-oriented modeling of control tasks and programs. We show the application of these concepts in the context of autonomous robot soccer.

1 Introduction

Robot control languages have an enormous impact on the performance of AI-based robot controllers. The languages allow for explicit and transparent representation of behavior specifi cations for reasoning and execution time program manipulation, and they provide the control structures for making the robot behavior flexible, reliable, and responsive. Despite their importance research on the design of robot control languages that enable intelligent robot control is largely neglected in AI — primarily for historical reasons.

As the predominant software architecture for autonomous robot control most researchers have used layered architectures, most notably the 3T architectures [15]. Characteristic for these layered software architectures is the use of multiple control languages: a programming language for the low-level reactive control and a very simple high-level language for strategic planning. This way the planner can still nurture the illusion of plans being sequences of plan steps and many existing planning techniques

carry over to robot control more or less the way they are. To bridge the gap between the partially ordered sets of actions (goal steps) and the low-level feedback control routines most software architecture use an intermediate control layer. In this intermediate layer an interpreter for a reactive plan language, such as RAP [7] or PRS [9], takes the high-level plan steps, selects methods for carrying them out based on sensor data, and executes them in a robust manner.

Unfortunately, this layered abstraction of robot control comes at high cost. The planning mechanisms cannot diagnose the behavior produced by a given plan because the behavior producing mechanism is much more sophisticated than assumed by the planning mechanisms. In addition, the planning mechanisms cannot exploit the variety of control structures offered by reactive plan languages to produce better behavior.

Let us illustrate this point using the following example taken from the autonomous robot soccer domain. A robot is to score a goal. An AI planner would typically produce a simple two step plan: (1) get the ball; (2) dribble it into the goal, because ball possession is a precondition for scoring. The navigation and the dribbling actions are considered as atomic black boxes. Unfortunately, these mechanisms do not allow for the generation of high performance plans with high scoring probability such as the one depicted in Figure 1(right). To compute such a high performance plan planning mechanisms have to tailor the parameterizations of the individual actions using accurate causal and physical models of the control routines they use.

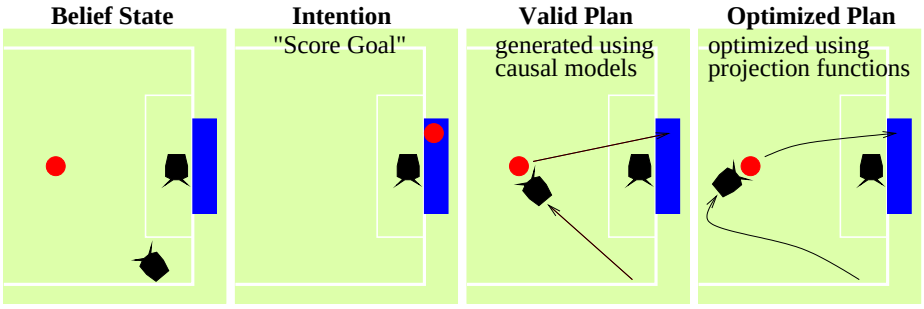

Fig. 1. Chaining of actions.

As plan-based robot control systems come of age and are applied to real world tasks a new generation of software architectures arises that are capable of dealing with these problems. These second generation software architectures share a number of important principles. These shared principles include (1) lightweight reasoning is embedded into the robot control languages; (2) the architectures invite programmers to specify models of the robot and its environment explicitly within the program code; (3) have much richer languages for the specifi cation of goals in terms of constraints on the values of state variables.

In our research group we are currently working on the next generation of the robot control/plan language RPL [12] called RPL-II. RPL-II allows for the explicit specifi - cation and representation of robot learning problems [5], for the specifi cation of explicit

robot and environment models, is object-oriented and supports the specification of specialization hierarchies for control tasks and routines and reasoning about them. RPL-II is an industrial strength robot control/plan language. It is implemented unlike its predecessor on a public domain CommonLisp using state of the art software tools such as Corba, UFFI, etc. RPL-II is applied to a variety of control tasks with different characteristics including mid-size robot soccer, a simulated household robot, and a robot assistant in an intelligent camera-equipped office environment.

This paper focuses on a particular aspect of the design of RPL-II, namely the representation and specification of the system that the controller controls and the control tasks that it performs. In a companion paper [5] we have described the new language features that support experience-based learning. The main contributions of the paper are the means for specifying state variables and their management, control tasks and control routines, and object oriented programming.

We demonstrate the application of these mechanisms to the implementation of our next generation controller for autonomous robot soccer. The implementation results in control programs that make extensive and explicit use of learning mechanisms and routines that can be much better reasoned about and transformed. This is primarily achieved through the explicit representation of physical entities and control tasks and the reasoning about them.

The remainder of the paper is organized as follows. In section 2 we describe the problems we encountered with our former control program for the AGILO soccer robots and sketch the ways we want to solve them. Section 3 introduces the basic concepts for model-based reasoning about physical control tasks. The use of these concepts is then demonstrated in the sections 4 and 5. We conclude with our next intended extensions of RPL-II, a discussion of related work, and our conclusions.

2 Languages at the Reactive Layer

The reasons why we want to use a model-based, object-oriented approach in structuring our control programs, are our experiences with former controllers of the AGILO soccer robots. [6] Let us therefore sketch how the controllers worked, which problems occurred and the conclusions we draw from them.

The control program ran in a low-level control loop that (1) updates the world model (section 2.1) and (2) chooses a command (section 2.2) in every time step (every 0.01 sec).

2.1 State representation

We need a set of variables containing information about the most likely current state of the world. We call this information “belief state”. The belief state is represented in a class “world model”, that provides variables and functions for any value the robot might want to know. It is not possible to differentiate between

- constant values (that would actually deserve the name “world model”),
- values taken from the percept or belief state vector, and

– values calculated from the belief state and world model.

So there is a representation of the state of the world, but with several drawbacks. The variables representing the state are not related to any physical values. In different parts of the program variables with different names, but the same physical meaning can occur. The orientation of the robot might in one place be called phi, in another place phi-deg. On the other hand does the same variable name not necessarily denote the same physical value. The variable pos-x can at one time express the robot's position, at another time the position of the ball.

Besides, the robot and the environment are not represented explicitly. When starting the robot in a certain environment the right configuration file has to be loaded and constant values are set. So we have just variables filled with values that have apparently nothing to do with the outside world.

Another problem is the heterogeneity of measuring units. This is especially hard when it comes to angles. Sometimes the value is given in degrees, sometimes in radian measure. In the action selector it is quite save to assume degrees, but there is no standard if degrees range from $0°$ to $360°$ or from $-180°$ to $+180°$.

2.2 Action Selection

Figure 2 shows a simplified extract of our old action selection routine. As is easily visible, there is only procedural knowledge in the controller. The information exchange between calling and called procedures is done by passing a variable param. The value(s) of this parameter sometimes denote the goal, sometimes a parameterization of the called function.

Furthermore the structure of the code seems very arbitrary. The purpose of the first three if-conditions is to trap failures. Then a more interesting part follows which decides what to do whenever the robot has the ball. Then again we have two failure conditions testing if the robot is in one of the penalty areas. Here we can see another problem. The reaction of being in the own penalty area or in that of the opponent team is almost the same and could be done by the same or a related function.

So in the end only three of eight cases build up the real controller code. The rest is only there for trapping failures. Even worse, the interesting parts are spread over the code and intercepted by failure testing. It is hopeless to reason about the best action, when there is no difference between failure trapping and real action selection.

Also the granularity of decisions seems ill-founded in our old controller code. When the robot has the ball, all we want to do is getting the ball somehow into the goal. Here the controller already decides how to do this (by dribbling the ball into the goal, kicking it or passing it to another player).

Another nuisance of our former controller is that it works in single time steps only and the decisions are purely reactive.

3 Key Concepts of RPL-II

After we have given a summary of the necessity of building a model-based, object-oriented system in section 2, we now have a closer look at the concepts we want to employ. These concepts are

```
function run-soccer-agent(worldmodel)
  var param := null
  var command
  if (not worldmodel.on-field) then command := NO-OP
  elseif (worldmodel.stuck-time > MIN-STUCK-TIME) then command := STUCK
  elseif (not worldmodel.localized) then command := RELOCALIZE
  elseif worldmodel.ball-is-in-guiderail then
    // do a lot of calculations and decide whether to call
    // PASS2POS, SHOOT2GOAL or DRIBBLE (with the goal as destination)
    param := ⟨x_dest, y_dest⟩
    command := DRIBBLE // for example
  elseif (worldmodel.time-in-opp-penalty-area > MAX-PA-TIME) then
    param := voronoi-pathplanning
    command := LEAVE-OPP-PENALTY-AREA
  elseif (worldmodel.time-in-own-penalty-area > MAX-PA-TIME) then
    param := voronoi-pathplanning
    command := LEAVE-OWN-PENALTY-AREA
  elseif (worldmodel.nearest-to-ball = my-robot-no) then
    param := voronoi-pathplanning
    command := GO2BALL
  else command := FACE-BALL
  execute(command, worldmodel, param)
```

Fig. 2. Code extract of our old AGILO controller.

1. state representation with globally accessible state variables
2. goal representation as constraints over state variables
3. control tasks and control routines arranged in an object hierarchy

We use the robot control language RPL (section 3.1), that provides constructs for monitoring failures while performing an action and parallel execution of processes. So now we think more in terms of actions than in terms of low-level commands and control loops.

For the representation of the robot's belief state we use globally known state variables that are described in more detail in section 3.2. Every variable corresponds to a physical value. We have not yet approached the representation of measuring units, although it should not be difficult within our framework.

The goal is now specified explicitly, tightly coupled to the state variables (section 3.3). We regard a goal as an intention how the world should be changed.

Finally, we require means of how to reach a given goal from a certain belief state. Our control procedures are structured along two lines, an object-oriented inheritance hierarchy and a calling hierarchy involving two classes of control procedures (section 3.4). We represent procedures as first-class objects, which allows for the specification of relevant calling parameters. Thus we can maintain a uniform calling mechanisms for all procedures. Inheritance is also an important factor when it comes to representing similarities between procedures. This makes the implementation very structured and concise.

To structure the calling hierarchy we introduce two classes of procedures, *control tasks* and *control routines*. A skill like "get-ball-into-goal" is implemented as a control

task. The different possibilities to fulfill the job like "dribble-ball-into-goal" or "kick-ball-into-goal" are represented as control routines. Control tasks and routines are called alternatively. The success and failure testing is completely done in the control task, as well as the choice of the appropriate routine in the current situation.

3.1 The Reactive Plan Language RPL

The robot's plans are implemented in RPL (Reactive Plan Language) [12], which has been successfully employed in different projects [4, 3, 2]. RPL provides conditionals, loops, program variables, processes, and subroutines as well as high-level constructs (interrupts, monitors) for synchronizing parallel actions. To make plans reactive and robust, it incorporates sensing and monitoring actions, and reactions triggered by observed events.

Connecting Control Routines to "Sensors" Successful interaction with the environment requires robots to respond to events and asynchronously process sensor data and feedback arriving from the control processes. RPL provides *fluents*, registers or program variables that signal changes of their values. Fluents are used to store events, sensor reports and feedback generated by low-level control modules. Moreover, since fluents can be set by sensing processes, physical control routines or by assignment statements, they are also used to trigger and guard the execution of high-level control routines.

Fluents can also be combined into digital circuits that compute derived events or states such as the robot's current distance to the ball. That fluent would be updated every time the position of the robot or the ball changes, since it is calculated out of the respective fluents.

Fluents are best understood in conjunction with the RPL statements that respond to changes of fluent values. The RPL statement **whenever** *F B* is an endless loop that executes *B* whenever the fluent *F* gets the value "true." Besides **whenever**, **wait for**(*F*) is another control abstraction that makes use of fluents. It blocks a thread of control until *F* becomes true.

Behavior Composition sources use control structures for reacting to asynchronous events, coordinating concurrent control processes, and using feedback from control processes to make the behavior robust and efficient. RPL provides several control structures to specify the interactions between concurrent control processes (figure 3). The control structures differ in how they synchronize processes and how they deal with failures.

The **in parallel do**-construct runs a set of processes in parallel and fails if any of the processes fails. The second construct, **try in parallel**, can be used to run alternative methods in parallel. The compound statement succeeds if one of the processes succeeds. Upon success, the remaining processes are terminated. Similarly **try in order** executes the alternatives in the given order. It succeeds when one process terminates successfully, it fails when all alternatives fail. **with policy** *P B* means "execute the primary activity *B* such that the execution satisfies the policy *P*." Policies are concurrent processes that run while the primary activity is active and interrupt the primary if necessary. Additional concepts for the synchronization of concurrent processes include semaphores and priorities.

in parallel do $p_1 \ldots p_n$	**in parallel do** navigate(⟨1.3, 2.0⟩) face-ball()
try in parallel $p_1 \ldots p_n$	**try in parallel** calculate-position-with-odometry() calculate-position-with-camera()
try in order $p_1 \ldots p_n$	**try in order** score-goal() distract-opponent()
with policy p b	**with policy** check-holding-ball() dribble(⟨4.2, 1.9⟩)

Fig. 3. Some RPL control structures and their usage.

3.2 State Representation

We represent the state of the world by globally declared variables. Figure 4 shows the class hierarchy of these variables.

Every value that is globally known throughout the system is called a *global value*. Every variable has a name and a (current) value. The world consists of values changing over time and values that remain constant. *Constants* are initialized when the system is started and represent the world model (section 4).

More interesting are *state variables*. Their value is represented as an RPL fluent, because it changes over time. Apart from being informed when a state variable has changed, it is often necessary to have access to former values. A *recording state variable* keeps a history of its past values. The history values can be accessed by the same function get-value that is used to obtain the current value of a state variable by specifying an additional parameter giving the number of steps we want to look back in time.

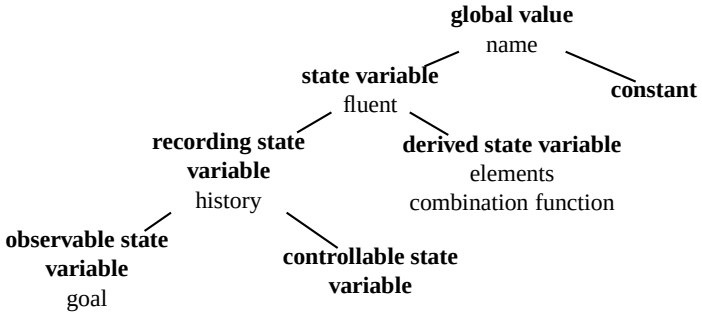

Fig. 4. Class hierarchy of globally known variables.

With these specifications we can now define *observable state variables* that represent our percept or belief state and *controllable state variables* representing the command. An observable state variable has an additional parameter for setting a goal value. This is explained in more detail in section 3.3.

The representation of the plain percept as state variables is usually not sufficient to make adequate decisions. For example we might want to react if a player is leaving the boundary of the soccer field. Or maybe we want to know if the robot has been stuck over a longer period of time. Therefore we introduced the concept of *derived state*

variables. From the outside, derived state variables are accessed just like observable state variables. But instead of keeping a history we remember the components and the function that produces the value of the derived state variable taking the component state variables as input. When a past value is accessed it is calculated from the history elements of the components.

As an example we have a look at the state variables in our soccer robots. The observable state variables include pos-x, pos-y and phi-deg which represent the x- and y-coordinates of the robot and its orientation as well as ball-x and ball-y denoting the position of the ball. The controllable state variables are c-rotation, c-translation, which set the robot's rotational and translational velocities, and kick, a boolean variable indicating whether to use the kicking device. Now we can define a derived state variable denoting the robot's distance to the ball:

make-instance derived-state-var
  name: distance-to-ball
  elementary fluents: *pos-x, pos-y, ball-x, ball-y*
  combination function: $\sqrt{(\textit{pos-x} - \textit{ball-x})^2 + (\textit{pos-y} - \textit{ball-y})^2}$

The fluent of the variable *distance-to-ball* depends on the state variables given in elementary fluents. The combination function calculates the current distance of the robot and the ball.

In order to test whether the robot is approaching the ball, we only need to check whether the distance to the ball has decreased:

fluent approaching-ball
  (get-value(distance-to-ball, t) $<$ get-value(distance-to-ball, t-1))

Since *distance-to-ball* is not a *recording state variable*, it does not have a history of its own. As the components and the function for obtaining the value are known and the components have a history, older values of *distance-to-ball* can be calculated.

3.3 Goals

Up to now our controller follows the very simple policy of our former control program described in section 2. In the future we would like to use a belief-desire-intention structure for the representation of top level goals or intentions (section 6). On the lower level we have to address the issue of how to tell a routine what to do.

There are two points of view for representing goals. First, we could order a routine to do something for us like "go to position $\langle$1.0, -1.5$\rangle$". So we have to pass a data structure that the routine must know how to interpret. The drawback of this idea is that there is no explicit relationship to the state variables *pos-x* and *pos-y*. The routine just knows that when these two state variables have the value of the goal specification the work is done.

Now the situation can also be seen as follows. Our control program wants to alter the world in a certain way, it might for example want to be in a state where the robot is at position $\langle$1.0, -1.5$\rangle$. It can now tell the corresponding state variables that it would like to have them changed to a different value. Then the controller calls a routine that is

best fi t to produce the desired state from the current situation. The called routine looks up the goal values of the state variables and tries to reach them.

3.4 Procedures

To support an explicit representation of the agent program, we describe procedures as fi rst class objects, so that we can reason about aspects such as performance measures or we can fi nd out if a procedure has yet to be learned.

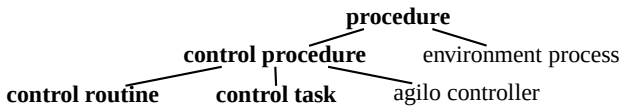

Fig. 5. Class hierarchy of procedures.

Figure 5 shows the basic class hierarchy of procedures. A *procedure* is any kind of function. At the moment, the for us most interesting subclass of *procedure* is a *control procedure*. A *control procedure* maps recent percepts to a command. Other kinds of procedures like environment processes, that map a command to a world state, might play a larger role in the future when we will model environment and perception processes.

For a good robot behavior we found it necessary to introduce two concepts of control procedures, *control tasks* and *control routines*. A robot should have certain skills, in the robot soccer domain we need skills such as dribbling or scoring a goal. These skills are called *control tasks*. Usually there are different ways to perform a skill. For example, in order to score a goal the robot might use its kicking device or dribble the ball into the goal. These implementations are called *control routines*. The job of the control task is to decide which control routine should be called in the current situation. This decision is based on models of the control routines that predict the time needed to fulfi ll a task or the probability of being successful.

Figure 6 shows a typical pattern of how control tasks and routines call each other. The task of scoring a goal can be achieved by two routines. One of them kicks the ball into the goal, the other one dribbles the ball to a point inside the goal. This second routine can be implemented like this:

```
control routine dribble-ball-into-goal
  p := find-goal-point()
  adjust-goal(pos-x ← p.x, pos-y ← p.y)
  execute(dribble)
```

We see that we need the task of dribbling in order to fulfi ll our goal. Therefore the control task *dribble* is executed, which can again be implemented by different control routines.

The “procedures” we are talking about are actually objects, the function that is really running is the generic function execute. With an object oriented approach we use inheritance mechanisms to get compact and concise implementations of the execute methods. This object oriented approach is especially useful in the domain of robot soccer where

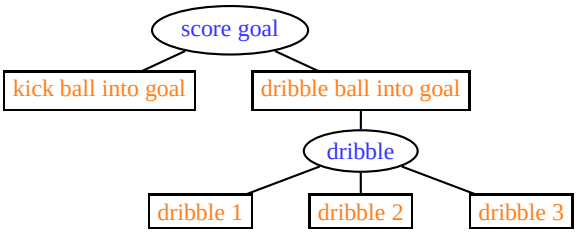

Fig. 6. Typical calling pattern of control tasks and control routines

almost every action comes down to navigation. So by using an object hierarchy we save a lot of work and redundancy.

Control Tasks Every skill is modeled as a *control task*. It should know when it has succeeded and when a failure has occurred and either react to it or abort execution. A special case of a failure is the exceeding of time resources. Since a control task's job is to choose the best control routine, it must know which control routines are available. All this information is given in the class definition:

```
class control task
  success
  failure
  time-out
  available routines
```

In most cases the control task will choose a control routine and check for failures or success during the execution. Such a method is shown in figure 7. What remains to do is the specification of the method choose-control-routine, which has to be implemented for each control task using models provided by the control routines.

```
method execute (ct of class control-task)
  r := choose-control-routine (ct)
  with-policy
    in parallel do
      whenever ct.failure fail("general failure")
      seq
        wait time ct.time-out
        fail("time-out")
    try in parallel
      execute(r)
      wait for ct.success
```

Fig. 7. Method execute for class *control task*

Control Routines are implementations of a *control tasks*. Since a control routine is always called by a control task, we don't have to worry about success or failure conditions. In both cases, the routine is interrupted by the control task. If there are errors the control task cannot detect, we can check them in the execute function of the control routine and return a fail command.

A control routine should not only reach a given goal state, it should also be able to predict how long it will take to reach this goal, what the accuracy of the solution will be or the probability of being successful at all. Thus, a control routine requires not only an execute method, but also methods providing information about its behavior.

4 Description of the Agent and the Environment

An intelligent robotic agent is more than just a program. It is a complex system that interacts with an environment through percepts and commands. We use this model to describe the agent, the environment and their interactions.

4.1 Declaration of the System Components

Fundamentally our system consists of two parts: an agent and an environment as described in [14]. These two components are absolutely independent, an agent can run in different environments and an environment can be the home of different agents. Therefore our first step is to state which agent should run in which environment. These declarations are principally used to declare state variables and constants (see section 3.2).

Figure 8 shows the parts we have to specify and how this information is used in global variables. Our *agent* consists of a *body*, an *architecture* and a *program*. The program is a control procedure that is called when the agent is given the command to run. The body describes physical properties of the agent like its dimensions. The architecture provides the connection to the environment, it describes which features the agent can receive as a percept and what kind of command can be given. The *environment* the agent acts in has certain properties that remain unchanged over time.

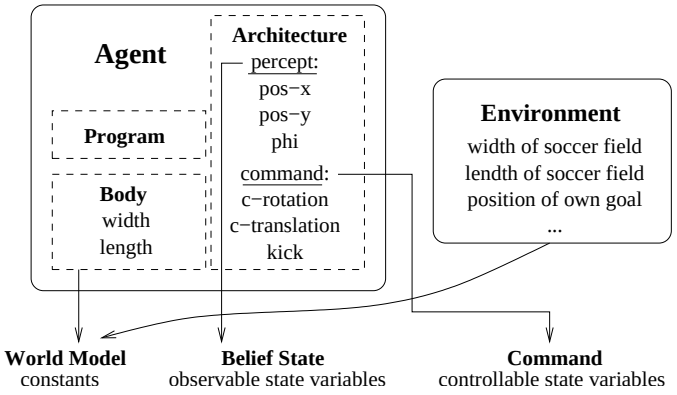

Fig. 8. Initialization of constants and state variables.

To summarize the information given by the agent and the environment we have three kinds of information, which is then provided by the global variables described in section 3.2: (1) a constant world model, (2) a belief state changing over time, and (3) a command.

4.2 Running the System

Now that we have declared an agent and an environment, we can run the agent. To do this, we call the function run-agent, which calls a method boot and starts a process update, both depending on the *agent-architecture* and the *environment*, and starts an RPL process that runs the *agent-program*. After initialization by the boot method agent and environment don't interact directly. The communication is done by the process update that gives the command in the controllable state variables to the environment and receives the percept, which it writes to the observable state variables. The state variables are set and read by a different process called RPL process which is running the agent program (figure 9).

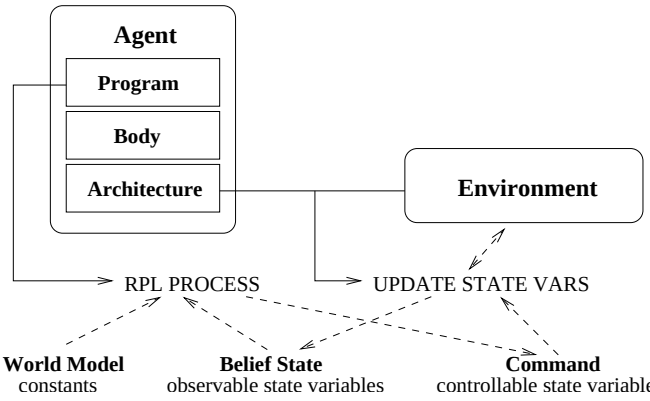

Fig. 9. The system at work.

When the agent has finished its job or when we want to stop the agent from the outside, the function kill-agent is called to stop the RPL and update processes and to call a method shutdown that specializes over the *architecture* and the *environment*.

5 The Agilo Controller

Using the concepts described in the previous sections we have implemented a very simple controller for our soccer robots. The controller consists of two processes: a monitoring and a controlling process. For this purpose the RPL construct with-policy can be used very effectively as shown in figure 10. The action selection is now concentrated in one loop, whereas the failure testing is done outside.

Of course, this is a very simple controller that has to be enhanced. We are planning to use a belief-desire-intention architecture to make sophisticated decisions (see also section 6).

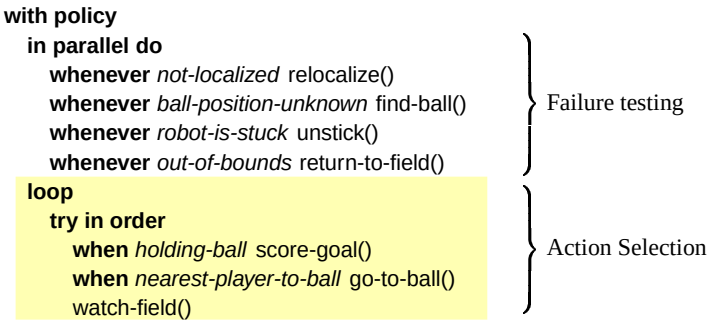

Fig. 10. Controller of our soccer robots

6 Research Agenda for RPL-II

We are developing RPL-II, the next generation of the robot control and plan language RPL. RPL-II supports the specification of high performance robot control programs by combining the expressiveness of RPL with respect to behavior specifications with advanced concepts that enable AI based control mechanisms to better reason about and manipulate control programs during their execution.

The extensions we have realized so far include the support of specifying explicit models of the physical systems to be controlled and object-oriented modeling of control tasks and routines. In two companion papers we have described extensions for the explicit representation of learning tasks in experience-based learning [5] and the tighter integration of programming and learning [10].

Still, these research results present only initial steps of the development of RPL-II, as a second generation AI-based robot control language. So let us briefly sketch the next steps on our research agenda: (1) comprehensive mechanisms for goal management, (2) improved physical system modeling, and (3) bootstrapping learning mechanisms for complex application tasks.

Comprehensive Goal Management. At the moment we only have the notion of low-level goals that are essentially constraints on state variables. So far RPL-II does not support goal selection that is consistent with the robot beliefs and other intentions of the robot. To support these goal management mechanisms we will add “desires” and “intentions” in addition to the current concept “goals” as first class objects in RPL-II and provide the respective reasoning mechanisms. This will give us the possibility of a much better action selection than the rule-based policy we are using now.

Deep models of state variables. While our current extensions make state variables explicit in the program code they still do not specify their physical meaning. We plan to provide such mechanisms by requiring programmers to specify the physical meaning in an explicit domain model formalized in a description logic. We believe that a concept taxonomy for a wide range can be provided and only small modifications for the individual application is needed. The slightly increased modeling effort will pay off immensely because using the domain model automated reasoning processes will be capable of solving much harder reasoning problems. For example, that all the conditions of a behavior trigger are observable or that two control routines will not interfere because the state variables they change are independent of each other.

Bootstrapping Learning Mechanisms. Finally, in RPL-II it will be possible to run partially specified control programs. The interpreter will then detect control tasks that the robot has no control routines for and acquire them by solving the associated learning tasks. This way a control program can complete itself or adapt itself to new environments and tasks by means of bootstrap learning.

7 Related Work

Model-based programming has been a major issue in several space exploration projects like Remote Agent [13], Livingstone [18], the Mission Data System Project [16], Reactive Model-based Programming Language [17] and others [1, 8, 11]. All of these projects represent the physical behavior of very complex systems in an explicit manner. This gives them the power to use lightweight reasoning techniques. The systems in these projects have to work very reliably. Vital parts of the physical systems are redundant, so that in the case of failure the system can be reconfigured. For this purpose the properties of the physical system parts have to be known by the controller. In our case reliability is not such an important issue. However, the environment our soccer robots have to deal with is much more dynamical. So we are interested in a robot that can adapt its control program to changing situations in its environment.

8 Conclusions

In this paper we introduce model-based concepts for the programming of robot controllers. The representation of state knowledge is done by state variables that are known throughout the system. Tightly coupled to the state variables is the representation of low-level goals. Those goals are achieved by control procedures arranged in two hierarchies: an object hierarchy that exploits inheritance mechanisms and a calling hierarchy including control tasks and control routines.

These concepts enable us to describe the robot and its environment declaratively. Using the robot control language RPL we can build a highly structured control program, where failure handling and action selection are separated.

On this basis we plan to include learning mechanisms as well as lightweight reasoning techniques. We still need to implement higher-level concepts like a belief-desire-intention architecture or logic representations to facilitate reasoning in the controller.

References

1. A. Barrett. Domain compilation for embedded real-time planning. In *Proceedings of the ICAPS'03 Workshop on Plan Execution*, 2003.
2. M. Beetz. Structured Reactive Controllers. *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Best Papers of the International Conference on Autonomous Agents '99*, 4:25–55, March/June 2001.
3. M. Beetz. *Plan-based Control of Robotic Agents*, volume LNAI 2554 of *Lecture Notes in Artificial Intelligence*. Springer Publishers, 2002.

4. M. Beetz, T. Arbuckle, M. Bennewitz, W. Burgard, A. Cremers, D. Fox, H. Grosskreutz, D. Hähnel, and D. Schulz. Integrated plan-based control of autonomous service robots in human environments. *IEEE Intelligent Systems*, 16(5):56–65, 2001.
5. M. Beetz, A. Kirsch, and A. Müller. Rpl-learn: Extending an autonomous robot control language to perform experience-based learning. In *3rd International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS)*, 2004.
6. M. Beetz, T. Schmitt, R. Hanek, S. Buck, F. Stulp, D. Schröter, and B. Radig. The AGILO robot soccer team - experience-based learning and probabilistic reasoning in autonomous robot control. *Autonomous Robots*, 2004. accepted for publication.
7. J. Firby. *Adaptive Execution in Complex Dynamic Worlds*. Technical report 672, Yale University, Department of Computer Science, January 1989.
8. M. Ingham, R. Ragno, and B. C. Williams. A reactive model-based programming language for robotic space explorers. In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, Montreal, Canada, 2001.
9. F. Ingrand, M. Georgeff, and A. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
10. A. Kirsch, A. Müller, and M. Beetz. Programming robot controllers that learn. submitted to International Conference on Intelligent Robots and Systems (IROS), 2004.
11. R. Knight, S. Chien, and G. Rabideau. Extending the representational power of model-based systems using generalized timelines. In *The 6th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, Montreal, Canada, 2001.
12. D. McDermott. A Reactive Plan Language. Research Report YALEU/DCS/RR-864, Yale University, 1991.
13. N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.
14. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
15. S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schmidt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, R. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, Cambridge, MA, 1998.
16. R. Volpe and S. Peters. Rover technology development and infusion for the 2009 mars science laboratory mission. In *Proceedings of 7th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, 2003.
17. B. C. Williams, M. Ingham, S. H. Chung, and P. H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, 9(1):212–237, January 2003.
18. B. C. Williams and P. P. Nayak. Livingstone: Onboard model-based configuration and health management. In *Proceedings of AAAI-96*, 1996.