Controlling Logistics Robots with the Action-Based Language YAGI

Alexander Ferrein[1(✉)], Christopher Maier[2], Clemens Mühlbacher[2], Tim Niemueller[3], Gerald Steinbauer[2], and Stavros Vassos[4]

[1] Mobile Autonomous Systems and Cognitive Robotics Institute, FH Aachen University of Applied Sciences, Aachen, Germany
ferrein@fh-aachen.de

[2] Institute for Software Technology, Graz University of Technology, Graz, Austria
{muehlbacher,steinbauer,maier}@ist.tugraz.at

[3] Knowledge-Based Systems Group, RWTH Aachen University of Technology, Aachen, Germany
niemueller@kbsg.rwth-aachen.de

[4] Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Rome, Italy
vassos@dis.uniroma1.it

Abstract. To achieve any meaningful tasks, a robot needs some form of task-level executive which acquires knowledge, reasons or plans, and performs and monitors actions. A formal approach for such agent programming is the Golog agent programming language. Golog is based on a first-order logic representation, and a drawback of common implementations is that in order to program agents, also knowledge of Prolog functionality is typically needed. In this paper, we present a prototype implementation of YAGI, a language rooted in Golog that offers a practical subset of the rich Golog framework in a more familiar syntax. Bridging imperative-style programming with an action-based specification, YAGI is more accessible to developers and provides a better ground for robot task-level executives. Moreover, we developed bindings for popular robotics frameworks such as ROS and Fawkes. As a proof of concept we present a YAGI-based agent for the RoboCup Logistics League which shows the expressiveness and the possibility to easily embed YAGI into robot applications.

1 Introduction

For a mobile robot to fulfill its tasks, some high-level decision making strategy is needed. There is a variety of paradigms for encoding and handling the high-level tasks of a robot. For instance, state machines or decision trees are often used to decide which action to take depending on sensor inputs, e.g., [13], and Belief-Desire-Intention architectures are used as the basis for high-level control languages such as 3APL [8] and PRS [9]. Another possibility to realize high-level control is classical task planning using standardized description languages such as the planning domain definition language (PDDL) [5]. Similarly,

© Springer International Publishing Switzerland 2016
N. Kubota et al. (Eds.): ICIRA 2016, Part I, LNAI 9834, pp. 525–537, 2016.
DOI: 10.1007/978-3-319-43506-0_46

approaches based on hierarchical tasks networks (HTNs) have been used [22]. Another expressive approach for high-level control that is based on a rich logical framework is Golog [11], along with its many descendants. Golog has a formal semantics based on the situation calculus [21] and has shown its usefulness in prototype applications ranging from educational robotics [12] and robot soccer [2] to domestic service robots [3].

While the Golog family languages have many advantages, e.g., advanced expressiveness, a formal underlying semantics, and the capability to combine imperative-style programming along with logical reasoning and planning, nonetheless the existing implementations have some drawbacks: (1) nearly all implementations are based on Prolog, which makes the integration of a Golog interpreter into a typical robot or agent architecture a less straightforward task; (2) Golog interpreters are often bounded by features of the underlying Prolog interpreter; (3) when writing agent programs, the distinction between Golog features and Prolog functionality is often not quite clear for most Golog implementations. Moreover, we see that Prolog is not the first choice for a roboticist making it less familiar than other programming environments. There are approaches to address these shortcomings, for example an interpreter based on Lua [4], and YAGI (Yet Another Golog Interpreter) [1]. YAGI is a language specification for a subset of Golog that was designed to reach out to a larger user group appealing to a more familiar imperative-style programming syntax. The state reported in [1] only concerned a first version of a user-friendly syntax and a semantic bridge to the situation calculus and Golog. A fully implemented YAGI interpreter and interfaces to robotics systems though was not available and the evaluation of YAGI (and Golog) in wider robotics scenarios was not possible.

In this paper, we present a fully-functional interpreter for the YAGI language that has been further extended with concepts necessary to control a robot system like sensing and exogenous events. Moreover, we show how robotics frameworks like ROS [20] and Fawkes [16] can be hooked up easily through a plug-in system. As a proof of concept we programmed YAGI agents for the RoboCup Logistics League [15,19] in simulation [23], where a team of robots has to fulfill logistics tasks in a virtual factory environment. YAGI overcomes the tight and confusing coupling between syntax and semantics of previous Prolog implementations of Golog and even allows to use different back-end algorithms for interpreting the language. We are convinced that tools like YAGI are a good step to motivate roboticist to use deliberative concepts like action-based programming to control their robots.

2 The YAGI Framework

2.1 Situation Calculus and Golog Prerequisites

YAGI (Yet Another Golog Interpreter) is an action-based robot and agent programming language, has its roots in the Golog language family [11] and its formal foundations in the situation calculus [21]. The situation calculus is a first-order logic language with equality (and some limited second-order features) which allows for reasoning about actions and their effects [21]. The major concepts are *situations*, *fluents*, and *primitive actions*. Situations reflect the evolution of the world and are represented as sequence of executed actions rooted in an initial situation S_0. Properties of the world are represented by fluents, which are situation-dependent predicates. Actions have preconditions, which are logical formulas describing if an action is executable in a situation. Successor state axioms define whether a fluent holds after the execution of an action. The precondition and successor state axioms together with some domain-independent foundational axioms form the so-called basic action theory (BAT) describing a domain.

Golog [11] is an action-based agent programming language based on the situation calculus. Besides primitive actions it also provides *complex actions* that can be defined through programming constructs for control (e.g. while loops) and constructs for non-deterministic choice (e.g. non-deterministic choice between two actions). This allows to combine iterative and declarative programming easily. The program execution is based on the semantics of the situation calculus and querying the basic action theory whenever the Golog program requires the evaluation of fluents.

2.2 Design Aims

Our work to provide a Golog-like interpreter with YAGI aims at overcoming some of the shortcomings of existing Prolog-based implementations, in order to make the action-based programming concepts used in Golog accessible to a larger community. A major problem of the existing implementations and also a reason for limited outreach is that it is hard for non-expert users to distinguish Golog from Prolog concepts. As a basic action theory and Golog are essentially defined in first-order logic, there is in fact no syntax specification for the language used in the implemented systems which typically use a mix of Prolog functionality with Golog-defined constructs. The YAGI syntax on the other hand, aims at a specification that is independent of the underlying implementation, and using concepts that are similar to familiar programming languages in order to make it easy to use. The YAGI syntax and semantics should follow the action-based framework of Golog, also providing constructs for realizing other important tasks for robotics, such as sensing or exogenous events. Finally, a YAGI interpreter should allow for an easy integration into other programming languages or robotics frameworks using clear and lean interfaces.

2.3 System Architecture

In order to achieve the above design aims, the YAGI framework uses the 3-tier architecture. The architecture comprises (1) a front-end, (2) a back-end and (3) a system interface. (Please refer to [14] for a detailed description.) This architecture allows a clear separation between the YAGI language and its syntax (see Sect. 2.4) and the implementation of the interpreter. The front-end provides the user-interface allowing queries and invocation of actions and programs and checks the syntactical correctness of queries, statements or programs. Currently the front-end is realized by an interactive console that allows to query fluents and to execute single statements or entire programs. The interface to the back-end is realized using an abstract syntax tree (AST) and a string-based callback. Therefore, in contrast to Prolog-based implementations the user has not to be aware of the concrete implementation of the system and different implementations of the back-end are easily possible. The back-end interprets YAGI statements and maintains a representation of the YAGI domain theory, the program to execute, and the state of the world. Moreover, it is able to execute imperative parts of programs as well as to plan over declarative sections. The back-end also incorporates information coming from the system interface such as result to sensing actions or exogenous events. The communication to the system interface is realized through an open string-based signal mechanism. The system interface provides the coupling of the back-end and the remaining robot system. It is responsible for the execution of actions triggered by the back-end and the collection and interpretation of feedback from the robot system. Using this system interface allows to hook-up different robot frameworks easily.

2.4 The YAGI Language

The aim of the definition of YAGI's syntax was to provide a clear definition of the language that allows to specify domain theories and control programs in same language while staying with the well-elaborated concepts of the situation calculus and Golog, but providing a more common style. The language definition is given in BNF and is therefore independent of any implementation. The full language definition can be found in [14].

The language provides statements to build up a domain theory with fluents, actions, and procedures. *Fluents* are used to represent the state of the world. While in the situation calculus fluents are first-order predicates that depend on a situation, in YAGI a fluent is represented as a set of tuples. The structure of the tuple defines the signature of a fluent. The interpretation of a fluent is that a fluent holds for a parameter tuple if the related tuple is in the set. Currently, the possible domain for parameters is a finite set of strings. For instance, the fluent $is\_at(p, l)$ represents a puck p being at a location l. Listing 1.1 depicts the fluent declaration with finite domains for pucks and locations. The definition is similar to well-known associative arrays. The set-based representation of fluents allows for an intuitive syntax for manipulation, queries and iteration over fluents. The assignment `is_at += <"Pk2","M3">`, which describes the effect on a fluent, looks similar to statements in common programming languages and express that the fluent additionally holds for the pair `"Pk2"` and `"M3"`. Similar statements exist for deleting pairs, assigning fluents, and more advanced assignments such as iterative or conditional assignments. *Query formulas* may include logical connectives and quantifiers similar to first-order formulas. For example, a statement such as `exists <$x,"M2"> in  is_at` – where `$x` denotes a variable named x – asks if any puck `$x` is at machine `"M2"`.

```
fluent is_at[{"Pk1",...,"Pk18"}]
             [{"D1",...,"D6","M1",...,"M24","R1","R2"}];
```

Listing 1.1. Definition of a binary fluent representing the location of a puck.

```
action goto($place) external ($status)
effect:
  skill_status = {<$status>};
signal:
 "skill-exec-wait ppgoto{place='"+ $place +"'}";
end action
```

Listing 1.2. Action that calls the Behavior Engine to move to a specified place.

YAGI further provides the concept of primitive *actions* that mostly follows the situation calculus definition by Reiter [21]. Listings 1.2 and 1.3 show two examples of action definition. The action `read_light` consists of four parts. The head defines the action's name plus a list of internal and external variables (l. 1). The internal variables are the regular parameters to the action. If external variables are defined, the action becomes a setting action. Such actions are a way to integrate sensing. We follow here the pragmatic way of directly set a fluent based on sensing result rather than reasoning about incomplete knowledge and alternative models [2]. Besides exogenous events, setting actions are extensions made to YAGI during this work in order to be able to reflect the behavior of a robot system. External variables are bound by the system interface after the action execution. The precondition (ll. 2–3) is a YAGI formula and denotes if an action is actually executable. In contrast to Reiter where successor state axioms are used to describe the effect of actions to fluents, in YAGI the effect on fluents (ll. 4–5) is directly expressed using fluent assignments. Please note that effects can comprise a sequence of simple as well as complex statements such as iterative and conditional assignments. Finally, the signal block (ll. 6–7) represents the communication to the system interface. Once the action is triggered for execution, the related string is sent to the system interface. Variables will be bound before transmission.

Statements to define *procedures* are also provided by YAGI. Procedures work the same way as in GOLOG. Listing 1.5 depicts a procedure definition comprising a head with the procedure's name and parameters and a body. The body of a procedure comprises primitive and complex actions. Complex actions are

```
1 action read_light() external ($red, $yellow, $green)
2 precondition:
3   blackboard_connected == {<"true">};
4 effect:
5   light_state = {<$red, $yellow, $green>};
6 signal:
7  "bb-get RobotinoLightInterface::/machine-signal";
8 end action
```

Listing 1.3. Action that reads the light signal in front of the robot and stores the outcome into the fluent `light_state`.

```
proc production()
  while not (exists <$M> in machines) do
    sleep_ms("1000");
  end while
  while phase == {<"PRODUCTION">} do
    get_raw_material("Ins1");
    pick <$M,"T5"> from machine_types such
      perform_production_at($M,"T5");
    end pick
    deliver("deliver1");
  end while
end proc
```

Listing 1.4. Procedure that implements the production main control loop.

```
proc exploration()
  while not (exists <$M> in expl_machines) do
    sleep_ms("1000");
  end while
  while phase == {<"EXPLORATION">} do
    if (exists <$M> in expl_machines) then
      pick <$M> from expl_machines such
        explore($M);
      end pick
    end if
  end while
end proc

proc explore($M)
  goto($M);
  read_light();
  exploration_report($M);
  mark_explored($M);
end proc
```

Listing 1.5. Procedure that implements the exploration main control loop.

for and while loops, conditionals and non-deterministic statements for picking arguments or statements. For example, the procedure in Listing 1.5 uses a `pick` statement to non-deterministically choose a tuple where a fluent holds. Please note that pattern matching with variables can be used. For instance `<$p,"R1"> in is_at` would select a puck `$p` which is at recycling machine `"R1"`. Also, a non-deterministic selection between actions is possible. For instance, the statement `choose A1() or A2()` chooses non-deterministically between the actions `A1` and `A2`. These non-deterministic statements play a major role if *planning* is used in YAGI. By default, YAGI performs on-line execution where the next best action is derived and executed immediately. No backtracking is done and the program is aborted if the selected action cannot be executed because the precondition does not hold. YAGI provides a `search` statement where parts of a program are executed off-line analogous to the original GOLOG semantics. If this statement is used, first a complete valid action sequence through the program is sought and once one is found, the sequence is executed. Therefore, YAGI allows to combine iterative programming and planning.

2.5 The Database Back-End

The two major tasks of the back-end are the representation of the YAGI domain theory including definitions of all fluents, actions, and procedures as well as the program execution.

For the first task, internal data structures are maintained and updated if the AST of the YAGI program contains domain-related statements like action or procedure definitions. The second task is tackled in the current implementation using a database representation for fluents [7]. Fluents are represented as tables in a relational database. Rows in a table represent the tuples the fluent holds for while the columns represent the fluent's parameter. If a fluent is defined, the according table is added to the database. Using a relational database allows for quick updates of fluents as well as efficient queries. The realization of the back-end using a database is the main reason for finite domains concerning fluents. It also imposes a closed-world assumption. The current implementation uses SQLite. YAGI formulas are evaluated using these tables and the usual SQL semantics.

For the execution of a YAGI program, its AST is traversed according to the YAGI semantics. The on-line execution of YAGI programs (basically the main procedure) follows the on-line transition semantics defined for IndiGolog [6]. This semantics uses the predicates *Trans* and *Final*. The predicate $\mathit{Trans}(y, d, y', d')$ denotes if a YAGI program y with the database d legally can lead to a remaining program y' and an updated database d'. The predicate $\mathit{Final}(y, d)$ denotes if a YAGI program y with a database d can terminate legally. In the on-line execution, once a legal transition is present this transition is executed. If the transition is final the program is terminated. During on-line semantics non-deterministic statements like pick and choose are random.

For the search statement with an off-line execution semantics, a complete trace through the AST is determined before any actions get executed. The current implementation uses a simple but complete breadth-first search algorithm. Once a legal transition within the AST is found, the remaining program is considered a valid state in the search space. For each state a database is kept which is updated according to the transitions leading to that respective state. The search is continued until a state with legal termination is found. Then, the actions along the path from the root node of the AST are executed in sequence.

The back-end is also responsible for the interaction with the system interface. If the back-end selected an action for execution, the according string is sent to the system interface including the proper binding of variables. Once the action was executed, the back-end updates the database according to the action definition's effects. In the case of a setting action, the external binding of variables by the system interface is considered. Please note that setting actions are not allowed within a search statement as the actions are not executed during the search and therefore no results from the system interface are available. Moreover, the back-end handles exogenous event in an asynchronous way. Exogenous events reported by the system interface are stored in a queue. After a transition step is finished all stored events are processed in a first-come-first-serve fashion. For each exogenous event, the variables are bound and the database is updated according to the statements in the event definition. An event definition is similar to a setting action definition, but has no precondition section.

2.6 System Interface

The system interface is realized as a C++ plug-in system allowing for an easy connection to different robot systems. Only two functions have to be reimplemented. The first function is called once an action (primitive or setting) is triggered. The function only needs to call the appropriate action within the robot system and to report the result. The second function handles the exogenous events reported by the robot system and adds them to the event queue of the back-end. Currently, implementations for ROS [20] and Fawkes [16] exists and are available as open source projects, see http://yagi.ist.tugraz.at. The Fawkes system interface used in this work is detailed in Sect. 4.

3 The RoboCup Logistics League in Simulation

RoboCup [10] is an international initiative to foster research in the field of robotics and artificial intelligence. It serves as a common testbed for comparing research results in the robotics field. RoboCup is particularly well-known for its various soccer leagues. In the past few years application-oriented leagues received increasing attention. In 2012, the new industry-oriented Logistics League Sponsored by Festo (LLSF) – renamed to RoboCup Logistics League (RCLL) in 2015 – was founded to tackle the problem of production logistics. Groups of three robots have to plan, execute, and optimize the material flow in a smart factory scenario and deliver products according to dynamic orders. Therefore, the challenge consists of creating and adjusting a production plan and coordinate the group of robots [15].

In 2014, the *LLSF competition* took place on a field of $11.2\,\text{m} \times 5.6\,\text{m}$ surrounded by walls (Fig. 1). Two teams are playing at the same time competing for points, (travel) space and time. Each team has an exclusive input storage (blue areas) and delivery zone (green area). Machines are represented by RFID readers with signal lights on top indicating the machine state. The lights indicate the current status of a machine, such as “ready”, “producing” and “out-of-order”. There are three delivery gates, one recycling machine, and twelve production machines per team. Material is represented by orange pucks with an RFID tag. In the beginning, all pucks (representing the products) are in raw material state and located in the input storage (blue zones).

The game is controlled by the *referee box (refbox)*, a software component which keeps track of puck states, instructs the light signals, and posts orders to the teams. After the game is started, no manual interference is allowed, robots receive information only from the refbox.

The game is split into two major phases. During the *exploration phase* the machines will show a light pattern to indicate a specific type. At game start,

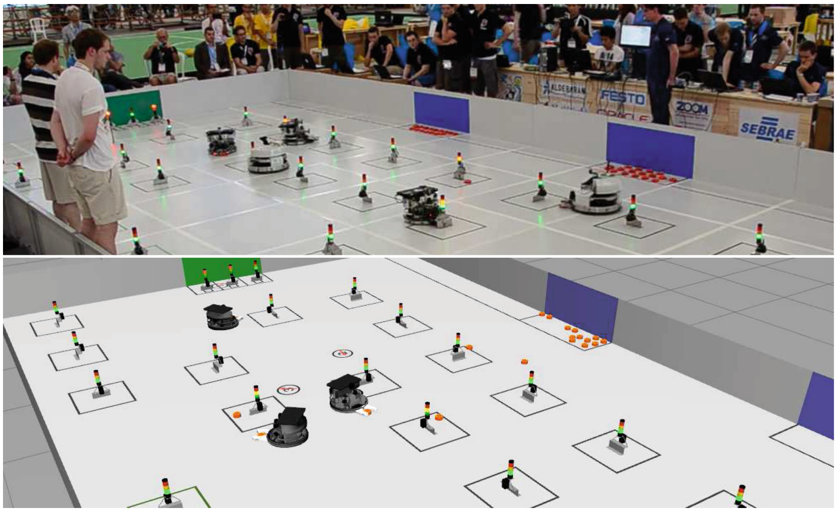
Fig. 1. Top: LLSF finals at RoboCup 2014. Bottom: The simulation of the LLSF in Gazebo. (Color figure online)

the refbox randomly determines the machine types, light pattern, and team assignment and announces it to the robots. Then, the robots have to explore the machines and report them back to the referee box. After four minutes, the refbox switches to the *production phase* during which teams win points for delivering ordered products, producing complex products, and recycling. The initial raw materials must be refined through several stages to final products using the production machines. Finished products must then be taken to the active gate in the delivery zone. Machines can be unavailable ("broken") for a limited time.

We use a *simulation of the LLSF* based on Gazebo [23] shown in Fig. 1 for development and testing. The simulation provides a 3D environment with optional physics engines and a variety of already supported sensors and actuators. The integrated system and many of the components are available as open source (http://carologistics.org). A noteworthy property of the LLSF is that the referee box provides *accountable agency to the environment*. The simulation uses the same game controller as in real games. Additionally, the simulation allows to provide different *levels of abstraction*. Here, we use mixed higher-level (signal lights directly from simulation instead of image processing) and lower-level abstractions (laser data for self-localization and navigation).

4 Implementing a Logistics Agent with YAGI

As we build on the Carologistics system, we use the Fawkes [16] robot software framework for integration. It provides the necessary functional components and integration with the simulation. We have developed a Fawkes-specific YAGI system interface which provides generic access to the Fawkes middleware as well as communication with the refbox. The basic skills, e.g. to grab a product from the input storage, are provided through the Lua-based Behavior Engine [17]. The skills perform limited execution monitoring and recover from local failures if possible. Failures that require a change of strategy or deliberation are reported to the YAGI program to deal with it. YAGI interprets the high-level control program and executes actions by calling skills in Fawkes. Information like the light signal states are read via setting actions, i.e. data is read explicitly at certain points in the program (upper left arrows). The communication with the referee box is provided through signal and exogenous events (upper right arrows). Note that there is no direct interaction between YAGI system interface and the simulation. Perception and actuation is performed through Fawkes only. This allows for a simple deployment on a real robot running the Fawkes framework as the interfaces remain the same. Integrating YAGI with additional frameworks only requires an appropriate system interface.

We have implemented a local, incremental, and in principal distributed agent program [19]. It is local in that its scope is a single robot, and not the group as a whole. Because it does not plan ahead the whole game but commits to a course of actions at certain points in time, we call the agent incremental. In principal, the agent could coordinate with other robots for a more efficient production without a centralized instance, making it a distributed system. However, at this time we limit our effort to the single robot case.

The `exploration` program (Listing 1.5) reads the machine mapping from the referee box (ll. 2–4), and then explores the machines by picking one machines after another calling the `explore` procedure for each (ll. 6–10). This procedure drives to the machine, reads the light signal, and reports it (ll. 14–19). Listing 1.2 shows the action declaration that calls the Behavior Engine to move to a specified machine in a blocking fashion which returns only after the skill has been finished (ll. 4–5). It then binds the status value to the `skill_status` fluent (ll. 1–3) to indicate success or failure of the skill execution. Listing 1.3 shows a setting action which gathers data from the blackboard. If a blackboard connection has been established (ll. 3–4), it calls to the system interface (ll. 6–7), data is returned in the external variables (l. 1), and stored in a fluent (ll. 4–5). The program consistently scores about half of the possible 48 points in simulation. To improve this, the pick could be modified to choose the closest unexplored machine. To achieve more points, an efficient use of multiple robots is necessary.

The `production` program (Listing 1.4) repeatedly completes the production chain for P_3 products, which require a single refinement step at a specific machine. Once machine information has been received (ll. 2–4), it retrieves a raw material from the input store (l. 6) and picks an appropriate machine to perform the production (ll. 7–9). After this production step, the raw material will have been converted to a P_3 product and can be delivered (l. 10). The score in the production phase has a greater variance because of the greater influence of travel distances and posted orders. The top score achieved was 55 points (out of possible 60 with P_3 products alone).

5 Discussion

The integration of the YAGI framework with the existing Fawkes-based system and the coding of an agent control program was achieved in about a week. The Fawkes system interface exploits middleware introspection to provide a generic integration. Other systems lacking this capability, like ROS, may not be able to offer a generic integration, but require a platform and domain specific one. The restriction to finite domains for fluents prevent the use of arbitrary numerics or user pointers as opaque data structures for closer C++ system integration. This, however, is not a principal issue, but can be improved in later revisions. Compared to a more elaborated CLIPS-based agent system [18], on-line YAGI emphasizes imperative constructs, while an event or rule-based system allows for easier revising a robot's goal. Integrating IndiGolog's interrupts or using YAGI's search capabilities could remedy this disadvantage. The particular benefit of YAGI is that it combines the strengths of two worlds: for one, it is based on the well-studied and well-understood foundations of Golog. This provides a sound semantics to the language and its interpretation. For another, it overcomes concerns prevalent in the robotics community by providing a more familiar and easy to use syntax for robot task-level executives. Our experiments implementing an agent program for the RoboCup Logistics League have shown that already the current prototype is a viable platform for such medium complex domains. Some of the current limits will be addressed in YAGI's vivid development process shortly. Finally, due to the fact that YAGI is based on Golog's formal logical semantics over programs and their executions, it is possible to perform much more advanced reasoning tasks, gaining all the benefits that come with the research that is being done in the situation calculus community. For example, it is not difficult to imagine future versions of a YAGI interpreter to be able to evaluate whether properties hold over future executions of programs or verify programs over different domains and scenarios, following current state-of-the-art research in verifying properties for Golog programs.

Acknowledgments. T. Niemueller was supported by the German National Science Foundation (DFG) research unit *FOR 1513* on Hybrid Reasoning for Intelligent Systems (http://www.hybrid-reasoning.org). C. Mühlbacher was supported by the Austrian Research Promotion Agency (FFG) with the grant Guaranteeing Service Robot Dependability During the Entire Life Cycle (GUARD). We thank the anonymous reviewers for their helpful comments.

References

1. Ferrein, A., Steinbauer, G., Vassos, S.: Action-based imperative programming with YAGI. In: Cognitive Robtics Workshop. AAAI Press (2012)
2. Ferrein, A., Lakemeyer, G.: Logic-based robot control in highly dynamic domains. J. Robot. Auton. Syst. **56**(11), 980–991 (2008)

3. Ferrein, A., Niemueller, T., Schiffer, S., Lakemeyer, G.: Lessons learnt from developing the embodied AI platform CAESAR for domestic service robotics. In: Proceedings of AAAI Spring Symposium. AAAI Technical Report, vol. SS-13-04. AAAI (2013)
4. Ferrein, A., Steinbauer, G.: On the way to high-level programming for resource-limited embedded systems with Golog. In: Ando, N., Balakirsky, S., Hemker, T., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2010. LNCS, vol. 6472, pp. 229–240. Springer, Heidelberg (2010)
5. Fox, M., Long, D.: PDDL2. 1: an extension to PDDL for expressing temporal planning domains. J. Artif. Intell. Res. (JAIR) **20**, 61–124 (2003)
6. Giacomo, G.D., Lespérance, Y., Levesque, H.J., Sardina, S.: IndiGolog: a high-level programming language for embedded reasoning agents. In: Multi-Agent Programming: Languages, Tools and Applications. Springer, US (2009)
7. Giacomo, G.D., Palatta, F.: Exploiting a relational DBMS for reasoning about actions. In: Cognitive Robotics Workshop (2000)
8. Hindriks, K., de Boer, F., van der Hoek, W., Meyer, J.J.: Agent programming in 3APL. Auton. Agent. Multi-Agent Syst. **4**(2), 357–401 (1999)
9. Ingrand, F., Chatila, R., Alami, R., Robert, F.: PRS: a high level supervision and control language for autonomous mobile robots. In: IEEE International Conference on Robotics and Automation (ICRA), vol. 1 (1996)
10. Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E.: RoboCup: the robot world cup initiative. In: Proceedings of 1st International Conference on Autonomous Agents (1997)
11. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: Golog: a logic programming language for dynamic domains. J. Logic Program. **31**(1–3), 59–83 (1997)
12. Levesque, H., Pagnucco, M.: LeGolog: inexpensive experiments in cognitive robotics. In: Cognitive Robotics Workshop (2000)
13. Loetzsch, M., Risler, M., Jungel, M.: XABSL - a pragmatic approach to behavior engineering. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (2006)
14. Maier, C.: YAGI - An Easy and Light-Weighted Action-Programming Language for Education and Research in Artificial Intelligence and Robotics. Master's thesis, Faculty of Computer Science, Graz University of Technology (2015)
15. Niemueller, T., Ewert, D., Reuter, S., Ferrein, A., Jeschke, S., Lakemeyer, G.: RoboCup logistics league sponsored by festo: a competitive factory automation testbed. In: Behnke, S., Veloso, M., Visser, A., Xiong, R. (eds.) RoboCup 2013. LNCS, vol. 8371, pp. 336–347. Springer, Heidelberg (2014)
16. Niemueller, T., Ferrein, A., Beck, D., Lakemeyer, G.: Design principles of the component-based robot software framework fawkes. In: Ando, N., Balakirsky, S., Hemker, T., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2010. LNCS, vol. 6472, pp. 300–311. Springer, Heidelberg (2010)
17. Niemüller, T., Ferrein, A., Lakemeyer, G.: A lua-based behavior engine for controlling the humanoid robot nao. In: Baltes, J., Lagoudakis, M.G., Naruse, T., Ghidary, S.S. (eds.) RoboCup 2009. LNCS, vol. 5949, pp. 240–251. Springer, Heidelberg (2010)
18. Niemueller, T., Lakemeyer, G., Ferrein, A.: Incremental task-level reasoning in a competitive factory automation scenario. In: Proceedings of AAAI Spring Symposium 2013 - Designing Intelligent Robots: Reintegrating AI (2013)

19. Niemueller, T., Lakemeyer, G., Ferrein, A.: The RoboCup logistics league as a benchmark for planning in robotics. In: Workshop on Planning and Robotics (PlanRob) at ICAPS-15 Scheduling (ICAPS) (2015)
20. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software (2009)
21. Reiter, R.: Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge (2001)
22. Sacerdoti, E.: Planning in a hierarchy of abstraction spaces. Artif. Intell. **5**(2), 115–135 (1974)
23. Zwilling, F., Niemueller, T., Lakemeyer, G.: Simulation for the RoboCup logistics league with real-world environment agency and multi-level abstraction. In: Bianchi, R.A.C., Akin, H.L., Ramamoorthy, S., Sugiura, K. (eds.) RoboCup 2014. LNCS, vol. 8992, pp. 220–232. Springer, Heidelberg (2015)