Integrating golog++ and ROS for Practical and Portable High-level Control

Maximillian Kirsch, Victor Mataré[a], Alexander Ferrein[b] and Stefan Schiffer[c]
Mobile Autonomous Systems and Cognitive Robotics Institute (MASCOR),
FH Aachen University of Applied Sciences, Eupener Str. 70, 52066 Aachen, Germany

Keywords: Reasoning about Action and Change, GOLOG, Robot Operating System, ROS.

Abstract: The field of Cognitive Robotics aims at intelligent decision making of autonomous robots. It has matured over the last 25 or so years quite a bit. That is, a number of high-level control languages and architectures have emerged from the field. One concern in this regard is the action language GOLOG. GOLOG has been used in a rather large number of applications as a high-level control language ranging from intelligent service robots to soccer robots. For the lower level robot software, the Robot Operating System (ROS) has been around for more than a decade now and it has developed into the standard middleware for robot applications. ROS provides a large number of packages for standard tasks in robotics like localisation, navigation, and object recognition. Interestingly enough, only little work within ROS has gone into the high-level control of robots. In this paper, we describe our approach to marry the GOLOG action language with ROS. In particular, we present our architecture on integrating golog++, which is based on the GOLOG dialect Readylog, with the Robot Operating System. With an example application on the Pepper service robot, we show how primitive actions can be easily mapped to the ROS ActionLib framework and present our control architecture in detail.

1 INTRODUCTION

In mobile robotics, the recent years have been marked by notable progress in recognition, manipulation and planning algorithms. The advent of robotics frameworks and middlewares has caused a consolidation of software stacks, with the Robot Operating System (ROS) emerging as the de-facto standard. The most noteworthy achievement of ROS is the growth of a lively community that shares re-usable components with well-defined interfaces.

The field of high-level control, however, has not seen the same amount of technical consolidation, over time leading to a disconnect between the ROS world and the Cognitive Robotics community. Of the many existing high-level action and planning languages, there exists only the ROSPlan package (Cashmore et al., 2015) in ROS which uses PDDL (McDermott et al., 1998; McDermott, 2000) as the declarative input language. While having PDDL readily available is certainly very valuable, we do believe that a purely planning-based approach is not necessarily the best choice for every real-world robotics problem. However, within the ROS ecosystem, the only other options for high-level control are based on finite automata (Bohren and Cousins, 2010; Brunner et al., 2016), which makes them more of a complement to PDDL than an alternative.

One approach that sits precisely in-between purely declarative problem description (planning) and imperative behavior prescription (like finite automata) is GOLOG (Levesque et al., 1997; Reiter, 2001). It allows for combining classical imperative programming constructs with more exotic constructs like the non-deterministic choice of action and argument. One feasible architecture is, for example, having a GOLOG agent play the role of an executive below a PDDL-based planner. Another option with some dialects of GOLOG is to have decision-theoretic planning integrated in the GOLOG interpreter itself. Different practical applications in robotics have shown that really any permutation of the three approaches can be suitable, depending on the domain requirements. High-level controllers based on Golog have been used for various domains such as robotic soccer (Ferrein and Lakemeyer, 2008) or domestic service robotics (Schiffer et al., 2012).

[a] https://orcid.org/0000-0003-4606-4758
[b] https://orcid.org/0000-0002-0643-5422
[c] https://orcid.org/0000-0003-1343-7140

Kirsch, M., Mataré, V., Ferrein, A. and Schiffer, S.
Integrating golog++ and ROS for Practical and Portable High-level Control.
DOI: 10.5220/0008984406920699
In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART 2020) - Volume 2*, pages 692-699
ISBN: 978-989-758-395-7; ISSN: 2184-433X

In this paper, we want to focus on the GOLOG language family and propose an interface that makes it a first-class citizen of the ROS world. Since our golog++-ROS interface aims to do for GOLOG what ROSPlan does for PDDL, there are some functional similarities, but also significant differences in non-functional requirements and the technical realization.

We start with some background, history and motivation for the development of golog++ in Section 2 before we review related approaches in other high-level languages in Section 3. Next, we give an overview of the central ideas behind golog++ in Section 4. In Section 5 we describe the interface between ROS and golog++ both from an abstract and an implementation-oriented perspective. We conclude with a discussion of the results and future work in Section 6.

2 BACKGROUND

Interfacing to some kind of execution platform is obviously an issue that all high-level agent languages have to deal with. In the following, we will give an overview over the most important concepts and the history of the GOLOG language family. Since we want to focus on ROS in this paper, we continue with a short introduction to its most relevant ideas and concepts, in particular those that cater to agent languages.

2.1 The GOLOG Language Family

All GOLOG dialects, including the original one (Levesque et al., 1997) have in common that they have a formal semantics based on the Situation Calculus (McCarthy, 1963; McCarthy and Hayes, 1969). The Situation Calculus is a second order logical language with equality which allows for reasoning about actions and their effects. The world evolves from situation to situation due to actions, starting in an initial situation S_0. Possible world histories are represented by sequences of actions. Properties of the world which can change from situation to situation are stored in so-called *fluents*, predicates or functions with a situation term as their last argument. Each action has preconditions and effects that formalize when it is possible to execute the action and how this action will change the world with regard to fluents. A basic action theory (BAT) contains axioms about the initial situation, action precondition axioms and successor state axioms as well as unique names and some additional foundational axioms.

GOLOG combines the declarative approach of the Situation Calculus with imperative programming, offering constructs such as loop and conditionals known from imperative programming languages. Many dialects also support less standard constructs such as concurrent program execution, nondeterministic choice or decision-theoretic planning. The purpose of the nondeterministic constructs is to generate the search space for a planning operator which can thus be used not only to search the entire action space, but to search for an executable parameterization of a nondeterministic program. Additionally, an agent must be able to perform both active and passive sensing, i.e. to actively execute an action to acquire new knowledge about its environment (Lakemeyer, 1999) (e.g. using sensors), and to passively receive new knowledge (i.e. react to uncontrollable events like the press of a button on the robot) (De Giacomo et al., 1997). Noteworthy about active sensing (i.e. *sensing actions*) is that their sensed effect is unknown during planning—only when the action is actually being executed will the agent's knowledge be updated. Passive sensing—a background update of fluents whenever a sensor value has changed in the world—is of course also unplannable by definition, and is usually modelled by so-called *exogenous actions*. These are actions that are beyond the control of the agent, but update its internal knowledge about the world.

In practical robotics, actions are not normally completed instantaneously. This is reflected in the concept of durative actions (Reiter, 1996), which introduces instantaneous actions that mark the begin and end of an ongoing activity. This also allows the agent to do other things (e.g. plan) while a physical action is in progress.

Many different GOLOG dialects have been proposed. CONGOLOG (De Giacomo et al., 1997) implements exogenous actions, interrupts and the ability execute actions concurrently, and INDIGOLOG (De Giacomo and Levesque, 1999; De Giacomo et al., 2009) expands on that allowing for incremental planning. DTGOLOG (Boutilier et al., 2000) introduces decision-theoretic planning, i.e. the ability to generate plans that are optimized with regard to some reward function. READYLOG (Ferrein, 2010b; Ferrein and Lakemeyer, 2008) builds upon that base and integrates features from CCGOLOG and PGOLOG (Grosskreutz, 2002; Grosskreutz and Lakemeyer, 2000) to handle actions with uncertain outcomes and fluents that change continuously.

While most of the GOLOG-related research deals with language semantics, practical use has shown that the issue of interfacing with and executing actions on a real robot is also not entirely trivial. GOLEX (Hähnel et al., 1998) is one of the first developments that identified some important interfacing issues: The

Level of Abstraction in GOLOG is generally too high to directly map primitive GOLOG actions to directives for robot actuators. *Execution Monitoring* is needed, since in reality actions can fail for many reasons beyond the control and/or scope of the agent program. To enable *Sensing and interaction*, the language semantics actually had to be extended (Lakemeyer, 1999). It was also determined that some resource management system is required "since certain resources such as motors cannot be shared simultaneously". At that time, GOLEX did not support parallel actions; so it was not able to react to user inputs, say, while the robot was travelling.

2.2 The Robot Operating System (ROS)

ROS (Quigley et al., 2009) is an open-source middleware designed for robotic application development. It offers fundamental concepts and tools to manage distributed robot applications, as well as ready-to-use packages for common problems such as localization, navigation, vision, motion etc.

Its fundamental architecture is a loosely coordinated, network-transparent multi-process system. Individual components are called *nodes*, and they communicate asynchronously by publishing messages under a so-called *topic*. Topics have a specific message type and are organized within a namespace. To receive messages, a node subscribes to a certain topic, after which it will receive all messages published under that topic. To support synchronous communication, there are so-called *services* which employ a request/response pattern.

This fundamental architecture has caused a leap forward in robotics research by fostering a community that can easily exchange software components with clearly defined interfaces.

2.3 ROS ActionLib

The ROS ActionLib[1] is an execution monitoring interface in the sense of (Hähnel et al., 1998). Client and server communicate over action-specific messages that define a *goal*,[2] a *feedback* and a *result*. To execute an action with certain arguments, an *ActionClient* sends a *GoalMessage* to a specific *ActionServer*. The action server is then responsible for executing the action with the given goal, tracking its progress (possibly giving *feedback*), and finally notifying the client if and when the goal has *succeeded* or was *aborted*, possibly along with some data as a *result*. For example, an action server can provide the action to move the robot from position A to B. The goal parameters could be the (x, y) coordinates the robot has to reach. As feedback, the action server could then, for instance, periodically notify the client about the current position, or it could simply give an estimated percentage driven of the planned trajectory. While the robot is on its way, the state of the goal is *running*. A running goal can be canceled, whereupon the action server stops the execution and the status of the goal is set to *preempted*. When the movement stops, the action server notifies the client with the final state (*succeeded*, *preempted* or *aborted*), possibly along with the final position.

All actions can be started in blocking or non-blocking mode. In the former case, the action client simply performs a function call which returns when the action server is done with execution, or in the latter case, the function dispatching a goal returns immediately, and the end of action execution is signalled with a callback.

3 OTHER RELATED WORK

ROSoClingo (Andres et al., 2013; Gebser et al., 2011) is one example of an action language that leverages the ROS ActionLib. It interfaces to the ROS ActionLib through two topics, one for output (i.e. executing actions) and one for input (i.e. action feedback/results and sensing). These two topics have to be subscribed to by special adapter nodes that translate the *ROSoClingo*-specific messages into ROS ActionLib goals.

Another noteworthy work is the framework ROSPlan (Cashmore et al., 2015) which embeds the action language PDDL (Fox and Long, 2011) into the ROS system. ROSPlan provides an interface to execute the planned actions of PDDL on a robot running ROS. Therefore, a dispatcher component holds the information of the next planned PDDL action to be executed, but it is up to the developer how the planned action should be executed in ROS. One of the reasons for this is that ROSPlan integrates several solvers that can solve a planning problem. A ROSPlan GitHub software repository demonstrates the execution of the planned action on a robot platform with the ROS Actionlib.

The so-called *Cognitive Robot Abstract Machine*[3] (CRAM) (Beetz et al., 2010) is another agent development framework that uses the Actionlib to interface with ROS. It targets robots in everyday (e.g.

[1] http://wiki.ros.org/actionlib

[2] Note that the term *goal* here has nothing to do with the *goal* concept in the context of a planning system. No reasoning is involved at this level.

[3] https://ias.in.tum.de/research/cram

domestic) environments, with a focus on integrating common-sense knowledge from multiple sources, sharing (updated) knowledge between robots and learning from past experience.

EXPCOG (Eppe et al., 2014) integrates several independent logic-based action calculi in one framework. Among others, a STRIPS-like planning system implemented in SWI-Prolog, an ASP-based on OCLINGO and a situation calculus-based on GOLOG and INDIGOLOG. However to our current knowledge there is no straightforward ROS integration.

The work of (Adam et al., 2017) transforms declarative expert domain models into PDDL problems. For this purpose, a Java interface must be implemented for each robot that calls the functions of the connected middleware. At the time of writing this paper, there was however no publicly available ROS integration.

4 THE golog++ FRAMEWORK

Most research about high-level control focuses on language semantics, while architectural aspects such as code reusability, portability and maintainability as well as usability and ergonomics receive less attention. The golog++ framework offers an alternative approach over other existing implementations that takes such aspects into account. It extends previous work on Golog such as *YAGI* or *golog.lua* (Ferrein, 2010a; Ferrein and Steinbauer, 2010; Ferrein et al., 2012; Ferrein et al., 2016).

The main purpose of golog++ (Mataré et al., 2018) is to factor the concerns of execution/platform-interfacing and static semantics out of the runtime semantics. This clear separation of concerns helps to significantly improve language usability, safety and maintainability.

The runtime semantics can be implemented in different ways, e.g. by embedding an existing GOLOG interpreter. With the *ReadylogSemantics*, golog++ supports both active and passive sensing, decision-theoretic planning and concurrency. Furthermore golog++ offers some unique features that are not found in other GOLOG dialects:

- Code model: The C++ object model that forms the core of golog++ supports all features required for an integrated development environment, i.e. full code introspection with reference resolution.
- Type safety: Everything has a type, and inconsistent use of types generates a specific error message. Compound key-value types and list types can be freely defined and nested.

```
symbol domain objects = {
  stop_sign, bottle, traffic_light
}
symbol domain position = {
  left, middle, right
}
string fluent sought_object() {
  initially: () = null;
}
number fluent found_position() {
  initially: () = null;
}
action dialog(string question) {
  senses: sought_object()
}
action detect_position(symbol obj) {
  senses: found_position()
}
action say(string say_str) {}
```

Figure 1: Exemplary BAT (defines a world and available actions).

```
procedure main() {
  dialog("What shall I search?");
  if (sought_object() != null) {
    start(say("Give me a second."));
    detect_position(sought_object());
    if (found_position() != null) {
      say("The " + sought_object() + " is at the "
              + found_position() + " position.");
      say("Thanks for your attention");
    }
    else
      say("Sorry, I couldn't find the "
              + sought_object());
  }
  else
    say("Sorry, I didn't understand.");
}
```

Figure 2: Exemplary imperative procedure that implements a simple behaviour on top of the BAT in fig. 1. Some details (like animations) are left out for conciseness.

- Syntax checking: golog++ comes with its own syntax and a special parser, so the syntax is strict, specific, and errors are pinpointed precisely.

In golog++, when a programmer defines an action $A(\mathbf{x})$, it is implicitly a durative action. That means that in terms of classical GOLOG, $A(\mathbf{x})$ actually defines a primitive action $start_A(\mathbf{x})$ that begins some activity, and another primitive action $end_A(\mathbf{x})$ that waits for it to end.

The golog++ example program shown in Figures 1 & 2 demonstrates the use of durative actions and active/passive sensing actions. The scenario can be seen in Figure 3: The robot Pepper (Pandey and Gelin, 2018) is standing in front of a table with a traffic light, a bottle and a stop sign on it. Spectators can swap the objects' positions around, and then ask Pepper for the position of either one. This example does not leverage golog++'s planning capabilities, but it does serve to show that a simple AI demo application can be written up with minimal effort.

In the next section, we will outline the underlying infrastructure that enables this kind of high-level behaviour development.

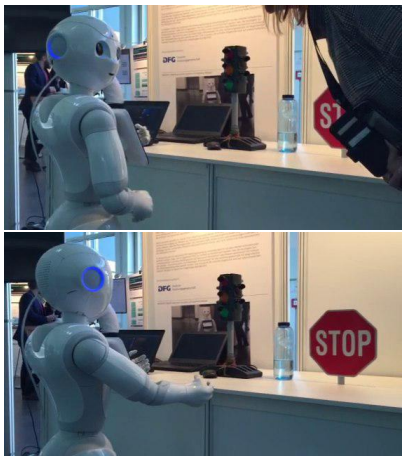

Figure 3: Interactive demo with Pepper at an event. Top: Pepper listens to a spectator who names an object that Pepper should look for. Bottom: Pepper looks for the object on the table and points out its position.

5 INTEGRATING golog++ AND ROS

Since golog++ is designed to work with different platforms, the *PlatformBackend* is an abstract interface that requires a platform-specific implementation. Its job is to map the durative actions, exogenous and sensing actions in a golog++ program onto a certain robot platform. Figure 4 gives an overview which functions the *RosPlatformBackend* has to implement. As the names suggests, the function *execute_activity* is called when the agent decides to begin some activity, and *preempt_activity* is called when the agent wants to cancel a currently ongoing activity. The function *update_activity* is already implemented in the *PlatformBackend* and must be called by the *RosPlatformBackend* to set the state *finish*, *fail* or *preempted*, and to (possibly) pass a sensing result when a running activity terminates.

In the following, we describe how the requirements of executing actions concurrently and active/-passive sensing can be satisfied with the ActionLib stack and ROS core functionality.

Action Execution. Obviously, the action concept of the ROS ActionLib is quite similar to that of golog++. As described in Section 2.3, the action server can finish in the states *succeeded*, *aborted* and *preempted*.

The *RosPlatformBackend* implements the *execute_activity* method simply by dispatching a goal to the appropriate action server in non-blocking mode. When the action is done, the callback then calls *update_activity*, informing golog++ of the action's success/failure. This ultimately allows the agent to execute the primitive action $end_A(\mathbf{x})$ and to decide whether/which effect has to be applied.

Another ROS concept that maps easily to golog++ actions is executing ROS *service* calls. A service call could be to e.g. enable face tracking on the Pepper platform (cf. Fig. 3). Since service calls are synchronous by nature, they have to be started in an own thread, so that they don't block the execution of the golog++ agent program. The thread then waits until the service reply has been received, and calls *update_activity* with *finish* or *fail* depending on whether the service call was successful or not.

Active Sensing. As described in Section 2.3, both an action server and a service call can return a specific result. The *RosPlatformBackend* translates the result into a golog++ *Value* and passes it to the *update_activity* function as a sensing result, together with the outcoming action state. So active sensing is realized like a normal golog++ action with the only difference that the action server (or service) has to provide a result in addition to the success/failure outcome.

Exogenous Actions. Exogenous actions are event-based by nature, which precisely matches the core message-passing architecture of ROS. Thus implementing an exogenous action comes down to simply subscribing to a topic where a message is published when the desired event occurs. The message is then converted into an *ExogEvent* object to be handled by the golog++ event loop.

5.1 Design of the RosPlatformBackend

One problem we did not mention so far is the fact that all ROS message types are realized as generated C++ headers with a strict typing discipline. Since every ActionLib action uses specific messages for the communication between ActionServer and ActionClient, both have to be either hand-written or generated in C++. Currently, the ROS ActionLib can generate ActionClients, but it does so without making use of inheritance, so there is no abstraction that would allow handling an ActionClient in a generic manner.

That is why, as shown in Figure 4, the *ActionManager*, *ExogManager* and *ServiceManager* classes exist, and why they must all be template classes. Their template argument is the action/service type for the *ActionManager* and *ServiceManager* respectively,

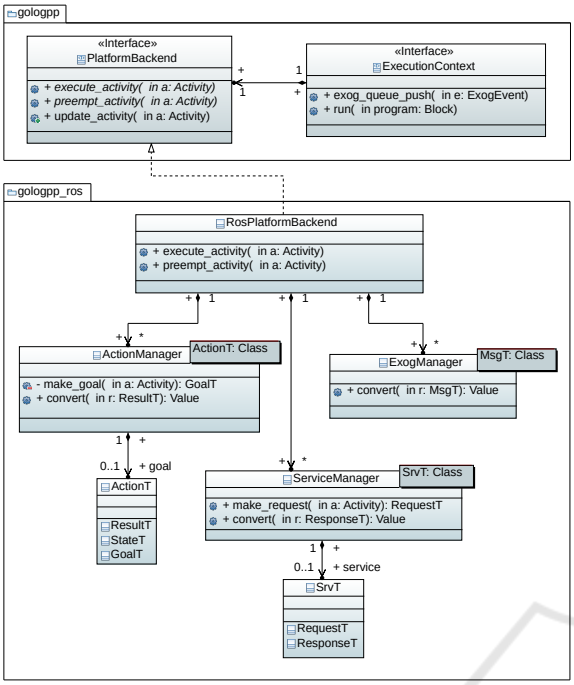

Figure 4: A UML class diagram that shows the most important properties of the golog++ *PlatformBackend* and some implementation details of the *RosPlatformBackend*.

```
template<>
darknet::obj_detectionGoal
ActionManager <
  darknet::obj_detectionAction
>::make_goal(const gpp::Activity &a)
{
  darknet::obj_detectionGoal goal;
  goal.obj_to_detect = std::string(
      a.mapped_arg_value("obj"));
  return goal;
}
```

Figure 5: Specialization of the *make_goal* method, which assigns the argument of an activity to a goal member for the action server. The string argument to the *mapped_arg_value* method specifies the name of the golog++ action's parameter, which can optionally be remapped from within the golog++ code (hence the method's name).

and the plain message type of the subscribed topic for the *ExogManager*. With this design, we are able to keep almost all of the glue code generic and leverage the code generation of the C++ template system to significantly reduce the amount of work that needs to be done to make a specific ROS action, service or topic accessible to golog++.

The only thing that remains to be written specifically for an action/service/topic is the actual mapping of data fields. As an example, consider the specialization of if the *make_goal* method in Figure 5. It simply maps the arguments of a golog++ action onto a goal for the DarkNet[4] ActionServer that does the object recognition used in Figures 1 and 2. The rest is taken care of by generic template code that is instantiated by the C++ compiler when it sees this specialization. If/when an outcome from the action server should be delivered as a sensing result, that is handled by another template specialization which translates that specific result type to a golog++ value.

6 DISCUSSION & OUTLOOK

The most obvious benchmark to compare the golog++ *RosPlatformBackend* to is the ROSPlan PDDL interface. ROSPlan interfaces to the outer world by publishing generic *ActionDispatch* messages that have to be received by another ROS node which then somehow implements the desired behaviour. Most commonly, that is done by executing some ActionLib action, just as in golog++. Unlike in golog++, in ROSPlan the programmer also has the freedom to not use ActionLib and write a free-form ROS node that can do anything. This freedom however can also be viewed as a lack of interface rigidity, and it comes with the cost of large amounts of boilerplate code. Having such a fundamentally unrestricted behaviour node also significantly increases the risk of programmer errors, especially when considering that it requires high-level developers to read into low-level ROS concepts.

In contrast, the golog++ *RosPlatformBackend* defines an extremely rigid interface for the platform integrator to implement. The behaviour required of the *convert*, *make_goal* and *make_request* method templates is completely specified by their respective input and output types (cf. Fig. 5). Most importantly, a golog++ platform integrator needs only to be proficient in C++, no further knowledge about ROS concepts (or even golog++ concepts for that matter) is required except what can be gleaned immediately from the types in the method's signature. Should action-specific quirks be required at other points in the control flow, the *RosPlatformBackend* also supports that by deriving from the *ActionManager*, *ExogManager* or *ServiceManager* class templates.

The difference between the two approaches also becomes apparent in the number of lines of code that are required to e.g. make a MoveBase action available. In golog++ that amounts to ∼12 LoC to translate action arguments into a MoveBaseAction goal. In ROSPlan, an entire ROSNode needs to be written for the same job, which weighs in around 50 LoC, most

[4]https://github.com/pjreddie/darknet

of which deals with managing the node itself as well as ActionClient objects, callbacks and other accessorial data structures which are hidden from the user in `golog++`.

Practical tests on the Pepper platform (Fig. 3) have shown that `golog++` in combination with ROS yields an easy-to-use and robust high-level agent architecture. Since Pepper is a platform directly targeted at human-robot interaction, implemented tasks revolved around social robotics. That is, Pepper acts as a simple tour guide in the rooms of the MASCOR Institute, or demonstrates simple speech-based human-robot interaction and neural network-based object recognition at science fairs and other events. `golog++`'s concise syntax and interpreted nature turned out to be very helpful in such scenarios by making it easy to adapt a demo application on the spot, e.g. to address questions from the audience.

Preliminary user studies have also shown that, given some rudimentary documentation and example code, users with varying backgrounds in robotics and computer science were able to interface their first new action to the `golog++` *RosPlatformBackend* in about one hour. Subsequent action interfaces were then done within mere minutes.

In the future, we plan on releasing `golog++` with the *RosPlatformBackend* as a ROS package to make it as accessible as possible. The next development step will be to hook into the ROS message generation infrastructure to completely automate the process of interfacing a ROS action to `golog++` so that no additional C++ coding will be necessary.

We are also following the development of ROS2 and will consider a port as soon as important functionalities like the ActionLib are sufficiently developed.

ACKNOWLEDGEMENTS

This work was supported by the German National Science Foundation (DFG) under grant number FE 1077/4-1

REFERENCES

Adam, K., Butting, A., Kautz, O., Rumpe, B., and Wortmann, A. (2017). Executing robot task models in dynamic environments. In *Proceedings of the 3rd International Workshop on Executable Modeling (EXE) held at ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 95–101.

Andres, B., Obermeier, P., Sabuncu, O., Schaub, T., and Rajaratnam, D. (2013). ROSoClingo: A ROS package for ASP-based robot control. *CoRR*, abs/1307.7398.

Beetz, M., Mösenlechner, L., and Tenorth, M. (2010). CRAM A Cognitive Robot Abstract Machine for everyday manipulation in human environments. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1012–1017.

Bohren, J. and Cousins, S. (2010). The smach high-level executive [ros news]. *IEEE Robotics & Automation Magazine*, 17(4):18–20.

Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S., et al. (2000). Decision-theoretic, high-level agent programming in the situation calculus. *AAAI/IAAI*, 2000:355–362.

Brunner, S. G., Steinmetz, F., Belder, R., and Dömel, A. (2016). Rafcon: A graphical tool for engineering complex, robotic tasks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3283–3290. IEEE.

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carreraa, A., Palomeras, N., Hurtós, N., and Carrerasa, M. (2015). ROSPlan: Planning in the Robot Operating System. In *Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'15, pages 333–341. AAAI Press.

De Giacomo, G., Lespérance, Y., and Levesque, H. J. (1997). Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *IJCAI*, volume 97, pages 1221–1226.

De Giacomo, G., Lespérance, Y., Levesque, H. J., and Sardina, S. (2009). Indigolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming*, pages 31–72. Springer.

De Giacomo, G. and Levesque, H. J. (1999). An incremental interpreter for high-level programs with sensing. In *Logical foundations for cognitive agents*, pages 86–102. Springer.

Eppe, M., Bhatt, M., Suchan, J., and Tietzen, B. (2014). ExpCog: Experiments in commonsense cognitive robotics. In *Proceeding of the 9th International Workshop on Cognitive Robotics (CogRob) held at the European Conference on Artificial Intelligence (ECAI 2014)*.

Ferrein, A. (2010a). golog.lua: Towards a non-prolog implementation of GOLOG for embedded systems. In Hoffmann, G., editor, *Proceedings of the AAAI Spring Symposium on Embedded Reasoning*, (SS-10-04), pages 20–28. AAAI Press.

Ferrein, A. (2010b). Robot controllers for highly dynamic environments with real-time constraints. *KI - Künstliche Intelligenz*, 24(2):175–178.

Ferrein, A. and Lakemeyer, G. (2008). Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56(11):980–991.

Ferrein, A., Maier, C., Mühlbacher, C., Niemueller, T., Steinbauer, G., and Vassos, S. (2016). Controlling logistics robots with the action-based language YAGI. In *International Conference on Intelligent Robotics and Applications*, pages 525–537. Springer.

Ferrein, A. and Steinbauer, G. (2010). On the way to high-level programming for resource-limited embedded systems with GOLOG. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 229–240. Springer.

Ferrein, A., Steinbauer, G., and Vassos, S. (2012). Action-based imperative programming with YAGI. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Fox, M. and Long, D. (2011). PDDL2.1: an extension to PDDL for expressing temporal planning domains. *CoRR*, abs/1106.4561.

Gebser, M., Grote, T., Kaminski, R., and Schaub, T. (2011). Reactive answer set programming. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 54–66. Springer.

Grosskreutz, H. (2002). *Towards more realistic logic-based robot controllers in the GOLOG framework*. PhD thesis, Bibliothek der RWTH Aachen.

Grosskreutz, H. and Lakemeyer, G. (2000). Turning high-level plans into robot programs in uncertain domains. In *ECAI*, pages 548–552.

Hähnel, D., Burgard, W., and Lakemeyer, G. (1998). GOLEX: Bridging the gap between logic (GOLOG) and a real robot. In Herzog, O. and Günter, A., editors, *KI-98: Advances in Artificial Intelligence*, volume 1504 of *Lecture Notes in Computer Science*, pages 165–176. Springer Berlin Heidelberg.

Lakemeyer, G. (1999). On sensing and off-line interpreting in golog. In *Logical Foundations for Cognitive Agents*, pages 173–189. Springer.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1997). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–84.

Mataré, V., Schiffer, S., and Ferrein, A. (2018). `golog++`: An integrative system design. In *Proceedings of the 11th Cognitive Robotics Workshop 2018 (CogRob@KR 2018), co-located with 16th International Conference on Principles of Knowledge Representation and Reasoning*, pages 29–36.

McCarthy, J. (1963). Situations, actions, and causal laws. Technical Report Memo 2, AI Lab, Stanford University, California, USA. Published in Semantic Information Processing, ed. Marvin Minsky. Cambridge, MA: The MIT Press, 1968.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press. reprinted in McC90.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL – the planning domain definition language.

McDermott, D. M. (2000). The 1998 ai planning systems competition. *AI magazine*, 21(2):35–35.

Pandey, A. K. and Gelin, R. (2018). A mass-produced sociable humanoid robot: Pepper: The first machine of its kind. *IEEE Robotics Automation Magazine*, 25(3):40–48.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

Reiter, R. (1996). Natural actions, concurrency and continuous time in the situation calculus. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, KR'96, pages 2–13, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Reiter, R. (2001). *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, Massachusetts.

Schiffer, S., Ferrein, A., and Lakemeyer, G. (2012). CAESAR: An intelligent domestic service robot. *Intelligent Service Robotics*, 5:259–273.