golog++: An Integrative System Design

Victor Mataré Stefan Schiffer Alexander Ferrein

Mobile Autonomous Systems and Cognitive Robotics
FH Aachen University of Applied Sciences
52066 Aachen, Germany

Abstract

Golog is a language family with great untapped potential. We argue that it could become a practical and widely usable high-level control language, if only it had an implementation that is usable in a production environment. In this paper, we do not specify another Golog interpreter, but an extensible C++ framework that defines a coherent grammar, developer tool support, internal/external consistency checking with clean error handling, and a simple, portable platform interface. The framework specifically *does not* implement language semantics. For this purpose we can simply hook into any of the many existing implementations that do very well in implementing language semantics, but fall short in regards to interfacing, portability, usability and practicality in general.

Introduction

As of 2018, we can safely say that the Golog [11] language family is still far from realizing its potential as a practical high-level robot control language. Theoretically, the idea of freely interleaving planning with traditional imperative programming should be at the core of how complex robot behavior is implemented today, but for practical reasons, most high-level control is still either finite state machines or specialized imperative code.

The reasons for this are manifold. In an academic context, the criticism is often leveled at Golog's runtime complexity and supposed obsoleteness due to advances in planning. The response then usually has to justify it as a worthwhile endeavour since there is no reason (yet) to just drop the middle ground between planning and programming. Golog's verifiability, generality, extensibility and transparency (e.g. w.r.t. accountability) are also often mentioned as redeeming features.

In our view however, the reasons why Golog is not really gaining the practical impact it deserves, are beyond the scope of most Golog-related research. From a robot manufacturer's point of view, Golog is simply not a feasible option due to numerous usability and engineering issues.

The Status quo and related work

There is a wealth of different implementations, and although diversity can be a good thing, the Golog family is missing technical coherence beyond the theory. Each implementation sports unique and valuable features, strengths and weaknesses. Documentation (aside from theoretical papers) on any of those is, however, sparse or nonexistent. To even figure out which one might fit a project's requirements, a programmer needs to be educated in formal logic to sift through the pertinent academic literature. Porting features between implementations can be tedious and error-prone since there are no explicit extension points and no internal consistency checks. In fact, most implementations do not even have any *external* consistency checking: Calling an action whose precondition is unsatisfied, for instance, yields the same result as calling an undefined action. While the former is a perfectly valid programming technique, the latter constitutes a malformed program which could easily be rejected with an explicit error message pinpointing the exact cause of the error. In general, the complete absence of such error handling in most Golog implementations makes debugging a larger codebase impossible within a reasonable time budget.

All of the Prolog-based implementations also tend to blur the line between the language and its imple-

In: G. Steinbauer, A. Ferrein (eds.): Proceedings of the 11th International Workshop on Cognitive Robotics, Tempe, AZ, USA, 27-Oct-2018, published at http://ceur-ws.org

mentation. This issue has been noted before, e.g. by [5]. Here, we want to emphasize that the Prolog implementation-level is not just *sometimes abused* for dirty hacks. On the contrary, it is even deliberately exposed as a quasi-legitimate *meta-language* to encode domain restrictions or to enumerate program elements. While this can certainly be a powerful tool to the expert programmer, it can also make any code unreadable if used irresponsibly. To the language learner, this metalanguage proves a conundrum, precisely because it is also the implementation language (i.e. it is effectively undefined!) and can be used to manipulate anything, *including the language semantics*.

The last major issue that cannot go unmentioned is interfacing with the real world. Usually, it boils down to some Prolog clause like

```
execute(Action, History) :- impl(Action).
```

where `impl(Action)` is some Prolog code that takes care of triggering the real action, e.g. in the behavioral layer of some robotics framework. One notable exception to this otherwise minimalistic approach is IndiGolog [1], which comes with a more elaborate interfacing called the *Environment Manager*. Its use case is however for one IndiGolog agent to control N disparate execution environments via TCP/IP. Today, this is obsoleted by robotics frameworks that provide a coherent, component-based view on a robot [15, 14]. Such frameworks typically offer network-transparent component APIs [16], so that Indigolog's $1 : N$ over TCP capability just adds unnecessary complexity, on top of the cruft of embedding Prolog on the remote end to basically just unmarshal a TCP packet into an action call.

In general, we can see that the interfacing logic of most Golog implementations doesn't go any further than the execution system specified by [6]. For N robot platforms and M Golog implementations, $N \times M$ platform interfaces need to be written since none of the interfacing code is portable between either robotics frameworks or Golog implementations.

All of these problems combined keep Golog from gaining a critical mass of supporters that could sustain a community that is interested in a good implementation.

Outside of the Golog community, we see more diverse approaches to high-level interfacing. The Semantic Robot Description Language (SRDL) [10] implements a framework that describes a robot platform with its components, and how they can be used to realize certain actions. As such, it serves a different purpose, namely mapping an abstract action concept to a realization strategy on a particular robot platform. To the framework we envision here, actions are opaque units, so a system like SRDL could be used as an action execution backend.

PRS/OpenPRS [9] and the ecosystem around them are worth noting because they also serve the purpose of a high-level control language, but found much wider use than any Golog dialect. However *PRS does not set itself apart through better theoretical foundations or through greater expressivity. The issue where *PRS clearly has the lead on any Golog dialect is usability, developer support, integration, i.e. *tool support* in general.

The same applies to the C-Language Integrated Programming System [18, CLIPS, cf.]. From a theoretical view, the language should be less expressive and less practical for high-level control application than Golog. But nonetheless, it is used much more widely[1]. While some of CLIPS' lead on Golog could be explained by the more general, rule-based language paradigm, its clear advantages are still a coherent language specification and well-designed, stable C/C++ bindings.

It is not like Golog's potential had not been widely noticed. It played a prominent part in the Semantic Web hype (cf. e.g. [12] with thousands of citations), and it has been used for high-level control of soccer playing robots [3] as well as for domestic service robots [17]. However unlike OWL [7], it never made the leap out of the academic laboratories. Again, observe how OWL is supported by a rich set of interoperable tools that help with visualization, error checking and debugging, while Golog is not.

How to fix it

We can summarize the issues outlined above as a general lack of tool support and usability which makes Golog infeasible as a practical high-level control language. So what features should an implementation have to make it useful in actual robotics scenarios? Important groundwork on this question has been done by [5]. Here, we want to expand on that while shifting the focus towards a more condensed and flexible architecture.

First of all, we have to differentiate two major user roles: The *language user* (e.g. an application developer), and the *language developer* (e.g. a maintainer or an extension developer). The non-functional requirements that follow are relevant to these roles in varying degrees. The requirements **Q1** through **Q4** follow directly from general guidelines of usability, as described e.g. by [2]. They apply mainly to the role of the *language user*. **Q5** to **Q8** on the other hand are most relevant to the *language developer*, and they

[1] See also 418 questions tagged "clips" on stackoverflow (`stackoverflow.com/questions/tagged/clips`) vs. 2 questions tagged "golog" (`stackoverflow.com/questions/tagged/golog`).

correspond directly to the fundamentals of software engineering.

Non-functional requirements

Q1 Familiarity Ensure that commonly known language constructs do what the programmer intuitively expects.

Q2 Readability Be syntactically "easy on the eye", i.e. let the visual structure reflect the syntactic structure (to help with skimming code).

Q3 Learnability Actively support the user in developing an understanding of the language's syntax and semantics. Make errors traceable to their cause, give a clear and specific hint at what is wrong.

Q4 Visibility Support implementation of visual editing frontends, i.e. code browsing, semantic code highlighting and interactive debugging.

Q5 Extensibility Define a clear path to introducing new language features and robot interfaces while ensuring internal consistency as well as possible.

Q6 Minimalism Don't bloat. Don't reinvent the wheel. Implement what is missing, re-use good code. Use minimal indirection, especially at runtime.

Q7 Separation of concerns Eliminate internal dependencies where possible. Concentrate external dependencies in few, well-specified interfaces.

Q8 Maintainability Keep external interfaces compatible. Be OpenSource. Be contributor friendly through good nomenclature, readable code and concise, instructive documentation.

The YAGI interpreter [5] went a long way towards **Q1**, **Q2** and **Q3** by defining a specialized language syntax and implementing it in a proper parser. **Q5** was considered in regard to robot interfaces, but in regard to the language itself, the focus was on specifying *and* implementing the full language semantics, effectively spawning yet another golog dialect (hence the name). This of course conflicts with **Q6**. Thought was also put into **Q7** and **Q8**, mainly by employing a clean coding style and using appropriate design patterns.

We have to embrace the fact that the majority of Golog-related research deals with semantic variants and extensions. Another significant push in expressive power can be expected from the convergence of machine learning with knowledge-based systems. So what we need at this time is not another Golog implementation, but a language design and interfacing framework that puts the main emphasis on extensibility and practicality. With that, the resulting functional requirements are significantly slimmed down in comparison to [5]:

Functional requirements

F1 Parse Golog programs written in a specifically designed syntax.

F2 Represent the parsed program in a concise, type-safe object model. Reflect the type safety in all extension points and in the parser.

F3 Run the program represented by the object model, re-using as much code from the existing implementations as possible.

F4 Execute actions found by running the program on any robot or simulation environment.

F5 Map sensors to exogenous actions/events as specified by the program.

F6 Monitor action execution to support failure handling and self-maintenance.

So here, we deliberately do not specify language semantics since that is exactly what we want to keep extensible. What do need is a coherent interfacing framework that takes care of *everything other* than language semantics, namely the functional requirements **F1** to **F6**, in a manner that satisfies the non-functional requirements **Q1** through **Q8**. In general, all of the non-functional requirements should be applied to all of the functional requirements to varying degrees. **Q7 Separation of concerns** and **Q8 Maintainability** apply to the system as a whole and are thus equally important to all of the functional requirements.

The `golog++` interfacing framework

From the functional requirements, we can derive three global system concerns: *Representation* of the Golog program (including parsing), static and runtime *semantics*, and *acting/sensing* e.g. on a physical robot platform. They are represented in Figure 1 as dashed areas. Since we want to be extensible both in the platform interface and in the language semantics, these main concerns are linked together through interfaces that make a compromise between generality and strictness.

Other than in YAGI, for instance, the parser does not instantiate an auto-generated syntax tree that contains unnecessary syntactic detail, but a concise, hand-designed class model (the *metamodel* component in Figure 1). The parser is written in templated C++,

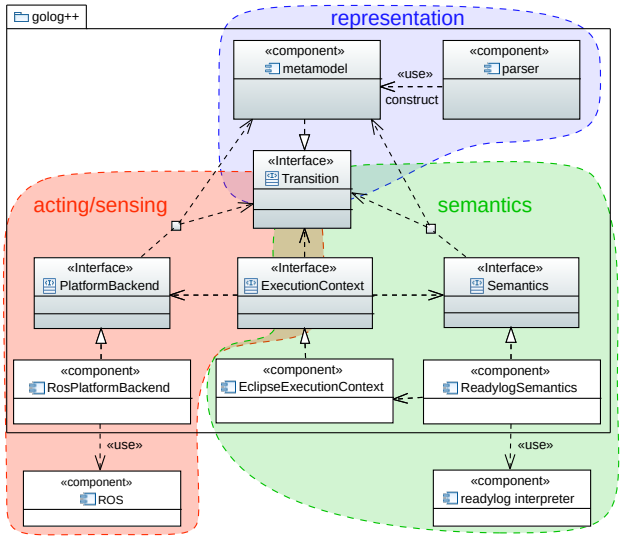

Figure 1: High-level architectural view of the golog++ framework with exemplary ROS/ReadyLog backends. Component and interface boundaries imply separation of concerns and independent extensibility. Core golog++ interfaces and components have a shaded fill. Exchangeable backend components have a white fill. The dashed/colored areas represent the three system concerns.

so the metamodel is usable and enforced in the grammar definition as well. An instance of the metamodel represents the basic action theory and the procedural code that together form a GOLOG domain/program.

Restrictions in the metamodel apply particularly to the extensible type model, syntactic constraints that follow from expression types, referential integrity and extension consistency. It conforms to the following (tentative and incomplete) specification:

Representation concern (language metamodel)

The golog++ metamodel encompasses two sets of classes: The static program representation, and the runtime state. Elements of the static program representation are classes that can form a BAT (like *Fluent, Action, EffectAxiom*), procedural code (like *Function, Pick, Search, Conditional*, etc.), and formula constituents like *Conjunction, Negation, Quantification* and so on. Of the runtime state representation, only the *Transition* interface is shown in Figure 1, since it links together the three main system concerns. Others, such as the *History*, might not even need explicit representation in all implementations.

Declarations and definitions of functions/procedures, actions and fluents are *Globals* (cf. Figure 3, only actions and fluents shown), which means that by themselves they are toplevel entities and cannot be constituents of another expression. All other program

```
$fluent loc($x) {
initially:
  ($b) = table, ($a) = $b
}

action stack($x, $y) {
precondition:
  $x != $y       // Can't stack x on itself
  & $x != table // Can't move the table
  & !exists($z) (    // There is no z...
    loc($z) == $x    // ... that is on x
    | loc($z) == $y // ... or on y
  )
effect:
  loc($x) = $y;
}

        ?function goal()
        { return loc(a) == table & loc(b) == a; }

search while (!goal())
  pick ($x) pick ($y)
    stack($x, $y);
```

Figure 2: A simplified golog++ code example from the blocksworld domain. The $ sign declares fluents, variables and functions as symbol-valued, so $loc(\cdot)$ is a functional fluent, and e.g. $loc(a) = table$ means that the block a is on the *table*, where a, b and *table* are all symbols that have to be defined as members of the fluent's domain. Here, this happens implicitly through the *initially:* section in the fluent's definition.

elements are either void-valued *Statement*s (i.e. instructions in a procedural code block that may fail to execute) or expressions that are syntactically exchangeable with constant values of the same type. The type system is designed to extensible, and the types *BooleanExpression, NumericExpression* and *SymbolicExpression* are always defined.

References can refer to any *Variable* or *Global*, and they inherit from a type alias exposed by their target class that determines what type of expression a reference to that class will be. So while *Variables* and *Globals* themselves are not constituent expressions, *Reference*s to them are.[2]

Certain expressions by definition have a different type than their constituents. For example $poss(\rho)$ and $do(\rho)$ are both boolean expressions for a (void-valued) action or procedure ρ. The ternary expression $\phi_1 \,?\, \phi_2 : \phi_3$ has the type $(T(\phi_3) = T(\phi_2))$, if $T(\phi_1) = Boolean$.

For the sake of rigidity and clarity, there is no implicit type conversion, e.g. comparisons can only be made between identical types and the result is always a boolean value. Every complex formula has the type

[2] cf. Figure 3. Note that the UML standard has no syntax for describing this pattern, so we emulate it by representing *TargetT::ExprT* as a separate interface that can be either *BooleanExpression* or *SymbolicExpression*.

of all its parts. Since all type-dependent code (including the parser) is templated, the restrictions specified above are enforced by the C++ compiler, even within the grammar definition. Implementing a new expression type in the metamodel automatically instantiates all needed templates where possible. Where manual template specialization is required, it is enforced by the C++ compiler.

Scopes are also represented explicitly in the static metamodel. Each *Scope* holds a reference to its parent scope, and it can resolve any symbol that is contained in itself or in one of its parents.

The parser reads a program in `golog++` syntax (like the example in Figure 2) and instantiates these metamodel classes to form a graph that represents the program as an interconnected C++ object structure. The code example in Figure 2 would (among others) produce an instance of the *Action* class called "stack", which accepts two symbol-valued arguments. As precondition, an *Action* can accept any *BooleanExpression*, which in this case would be an instance of the *Conjunction* class, which is again made up of *Comparison*s and a *Quantification*. Via the nesting of constituents, the program's object representation thus forms a syntax tree. *Reference*s like the calls to the `goal()` function and to the `stack($x, $y)` action in the main program can interconnect the objects across the syntactic tree's branches, thus forming a (possibly cyclical) graph that represents the program's referential (call) structure. Each expression in a program is represented by a unique object, and none of these objects are copyable since their exact place within the syntactic tree is part of their identity.

Semantics concern

Every language element in the metamodel owns a specific implementation of the *Semantics* interface that is attached after a program's object representation is fully constructed. The core interfaces (i.e. all shown in Figure 1 and the entire *metamodel*) do not make any assumptions about how the *Semantics* interface is realized. To give a program its semantics (i.e. to make it executable), the *ExecutionContext* recurses twice along the syntax tree of all *Global*s and of the main procedure. On the first recursion, every program element is visited by a factory (leveraging runtime polymorphism) that assigs a unique instance of a specific *Semantics* implementation to it. On the second recursion run, the *ExecutionContext* delegates to a virtual method in its concrete implementation that must use each element's *Semantics* to prepare it for execution (e.g. by compiling, initializing runtime state, etc).

So for the fluent declaration `loc($x)` in Fig. 2, the *ReadylogSemantics* (cf. Fig. 1) would produce the

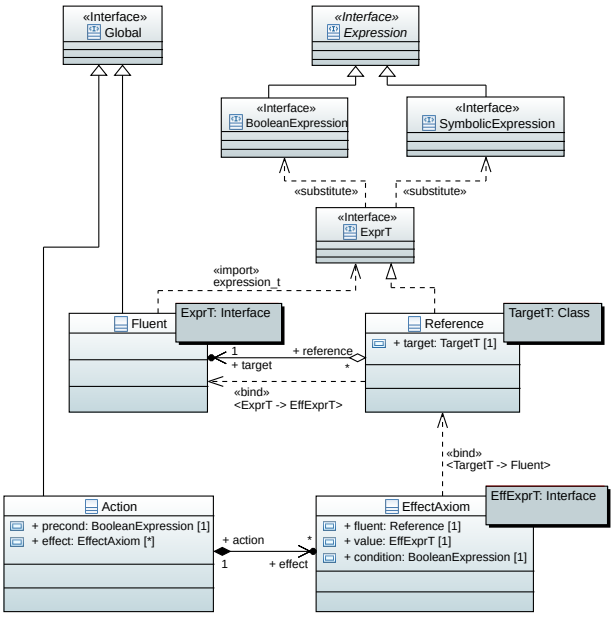

Figure 3: Simplified UML class diagram showing a small fragment of the `golog++` language metamodel. Most class properties, operations and connectors are omitted to focus just on how the *Action*, *EffectAxiom*, *Fluent* and *Reference* classes relate to the expression types.

following Prolog clause:

```
prim_fluent(loc(X)) :- member(X, [a, b]).
```

Here, the argument variable `X` and the domain symbols `a`, `b` and `table` are each constituents of their own, so the fluent's semantics delegates to their semantics for producing the respective Prolog terms.

After compilation, all *Semantics* implementations must be in a state that allows the *ExecutionContext* to test the main program for the next *Transition* and for the *final* state. These methods are also specified by the *ExecutionContext* and realized by its implementation. The *EclipseExecutionContext* shown in Figure 1 for example embeds an `eclipse-clp` interpreter, loads `ReadyLog` [4] and uses its `trans/4` and `final/2` predicates to implement runtime. A *Transition* created by an *ExecutionContext* implementation is a parameterized *Action*. When a legal *Transition* is found, it is passed to the *PlatformBackend* for execution.

The *ExecutionContext* also provides access to *ExogEvent* objects that are created from sensor data as well as from state changes in platform components or in action execution. An *ExecutionContext* implementation must consume these events and apply them to the runtime state implementation of the current `golog++` program (usually by incorporating them into the history).

Acting/sensing concern

The *PlatformBackend* accepts a *Transition* object, and its implementor must use that to trigger an action on some target platform. A *PlatformBackend* implementation must also select and retrieve the sensor data and other runtime state that is needed to instantiate the *ExogEvent* objects as specified by the `golog++` program. A more precise specification of this interface is subject to currently ongoing work.

Discussion and rationale

This specification and its current implementation already satisfy large portions of the functional and non-functional requirements. **Q1 Familiarity** is mostly a given since the procedural semantics of GOLOG are already aligned with traditional imperative programming. Employing a familiar C-like syntax then makes the imperative GOLOG code immediately recognizable to all programmers that know e.g. C/C++, Java or JavaScript. A C-like syntax has been chosen over the Pascal-like variant employed in YAGI since it is more widely used, and we expect better visual ergonomics by using punctuation (i.e. curly braces) as block delimiters. **Q2 Readability** is further helped by using a C++ class-like notation for GOLOG action definitions (again inspired by YAGI), with "`precondition:`" and "`effect:`" as section markers within the action's definition block. However we have dispensed with YAGI's way of notating a fluent as a 0-arity functional symbol that denotes a set of n-tuples. Instead, we returned to the classical SitCalc-like notation with n-ary functional or relational fluents, which should be more recognizable to developers with e.g. a PDDL background.

The goal of **Q3 Learnability** is of course closely related to **Q1 Familiarity**, **Q2 Readability** and **Q4 Visibility**. A typical GOLOG developer may likely be proficient in other languages, but given its currently limited pervasion, we have to assume that he/she is unfamiliar with the Situation Calculus or any other GOLOG dialect. The most important tools to support **Q3** are type safety and interactive debugging (i.e. single-stepping a live program with viewable runtime state).

Support for interactive debugging is limited in the current *ExecutionContext* implementation since the interpretation of all procedural code is encapsulated within the toplevel `trans/4` call to the `ReadyLog` interpreter. To achieve full support for interactively single-stepping procedural code, an *ExecutionContext* implementation would have to handle each procedural statement individually. An alternative would be to change the implementations of all language elements to not `assert/1` or `compile/1` plain Prolog predicates representing the GOLOG program, but to register as C++ external predicates that implement appropriate debugging hooks.

Type safety is important to support **Q3 Learnability** since knowing the type of an expression allows for a much more descriptive code model. In particular, references can be resolved statically, which allows for implementation of proper code browsing (**Q4 Visibility**). Using the type of an expression to restrict the grammar allows us to catch many more cases of accidental misuse of symbol references and other expressions.

Q5 Extensibility, **Q6 Minimalism**, **Q7 Separation of concerns** and **Q8 Maintainability** are again a set of non-functional requirements that tend to go hand in hand. Together, they form the technical foundation for continued proliferation of any piece of software. In `golog++`, we achieve **Minimalism** by dealing purely with the three main concerns of *representation*, *semantics* and *acting/sensing*. It is important to note that only the *representation* concern (parsing into an instance of the metamodel) has a stand-alone implementation, i.e. it is the only one that does not depend on some external implementation of abstract interfaces to run. The metamodel's static and runtime state representation forms the (singular!) interface through which the semantics and acting/sensing concerns interact. Consequentially, **Q7 Separation of concerns** in `golog++` means that anything concerned with semantics or acting/sensing can ever only depend on the metamodel, while the metamodel can ever only depend on the abstract acting/sensing and semantics interfaces.

Conclusion and outlook

The `golog++` language grammar will be a more dedicated subject in future work when the syntactic extension points are developed. The mid-term vision is to support namespacing and language profiles similar to OWL [13].

One use case of the `golog++` framework is the *ConTrAkt* project, where the GOLOG language will be extended to support platform self-monitoring and autonomous, platform-aware failure recovery. To that end, a language specification for platform constraint modeling is in development. Its goal is to allow domain-independent encoding of platform details so that domain modellers don't have to worry about platform quirks [8]. It will make use of extension points in the acting/sensing concern (i.e. the abstract platform interface) which are yet to be specified. It will also require syntactic extensions and possibly an external constraint solver to be embedded via an *ExecutionContext* implementation.

The design goals of the framework presented here

are flexibility, extensibility, usability and ease of deployment. Its most important feature is to decouple GOLOG reasoning, program representation and platform interfacing. Consequentially, to serve N platforms with M different language semantics, we simply need N platform interfaces and M semantics, instead of $N \times M$ leaky abstractions like before. Decoupling also allows us to use the right tool for the job: Modern, templated C++ for metamodeling, interfacing and outer control flow, and Prolog (or other AI-affine languages) for the implementation of language semantics and reasoning.

The complete source code of the `golog++` framework is freely available at `https://github.com/MASKOR/gologpp.git`.

Acknowledgments

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

References

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

[2] A. Dix, J. Finlay, G. D. Abowd, and R. Beale. Design rules. In *Human-Computer Interaction*, chapter 7, pages 258–288. Pearson Education Limited, 3rd edition, 2004.

[3] A. Ferrein, C. Fritz, and G. Lakemeyer. Using golog for deliberation and team coordination in robotic soccer. *KI*, 19(1):24, 2005.

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

[5] A. Ferrein, G. Steinbauer, and S. Vassos. Action-based imperative programming with YAGI. In *Proceedings of the 8th International Conference on Cognitive Robotics. AAAI Press*, 2012.

[6] D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX – bridging the gap between logic (GOLOG) and a real robot. In O. Herzog and A. Günter, editors, *KI-98: Advances in Artificial Intelligence*, pages 165–176, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[7] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. OWL 2 web ontology language primer. *W3C recommendation*, 27(1):123, 2009.

[8] T. Hofmann, V. Mataré, S. Schiffer, A. Ferrein, and G. Lakemeyer. Constraint-based online transformation of abstract plans into executable robot actions. In *AAAI Spring Symposium 2018 on Integrating Representation, Reasoning, Learning, and Execution for Goal Directed Autonomy*, Stanford, CA, USA, 2018.

[9] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 43–49, 1996.

[10] L. Kunze, T. Roehm, and M. Beetz. Towards semantic robot description languages. In *IEEE International Conference on Robotics and Automation*, pages 5589–5595, 2011.

[11] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1–3):59–84, April-June 1997.

[12] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, Mar 2001.

[13] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, et al. OWL 2 web ontology language profiles. *W3C recommendation*, 27:61, 2009.

[14] T. Niemueller, A. Ferrein, D. Beck, and G. Lakemeyer. Design principles of the component-based robot software framework fawkes. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 300–311. Springer, 2010.

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

[16] M. Reichardt, T. Föhst, and K. Berns. An overview on framework design for autonomous robots. *it-Information Technology*, 57(2):75–84, 2015.

[17] S. Schiffer, A. Ferrein, and G. Lakemeyer. CAESAR – An Intelligent Domestic Service Robot. *Journal of Intelligent Service Robotics*, 23(Special Issue on Artificial Intelligence in Robotics: Sensing, Representation and Action):259–273, 2012.

[18] R. M. Wygant. CLIPS – a powerful development and delivery expert system tool. *Computers & industrial engineering*, 17(1-4):546–549, 1989.