CABSL – C-Based Agent Behavior Specification Language

Thomas Röfer[1,2(✉)]

[1] Cyber-Physical Systems, Deutsches Forschungszentrum für Künstliche Intelligenz, Enrique-Schmidt-Str. 5, 28359 Bremen, Germany
thomas.roefer@dfki.de

[2] Universität Bremen, Fachbereich 3 – Mathematik und Informatik, Postfach 330 440, 28334 Bremen, Germany

Abstract. This paper describes the *C-based Agent Behavior Specification Language* (CABSL) that is available as open source [8]. It allows specifying the behavior of a robot or a software agent in C++11. Semantically, it follows the ideas of the *Extensible Agent Behavior Specification Language* (XABSL) developed by Lötzsch *et al.* [6], i.e. robot behavior is described as a hierarchy of finite state machines. However, its integration into a C++ program requires significantly less programming overhead than when using XABSL. CABSL has been part of all B-Human code releases since 2013 [9], but it is now also available as a standalone release that works without the B-Human base system.

1 Introduction

Modeling the behavior of a software agent or a robot is an important part of building autonomous systems. In the domain of RoboCup, real-time requirements and limited computational resources often prevent the use of planning-based approaches. Instead, the behavior is explicitly specified. In this context, hierarchical finite state machines have been proven to be a successful concept. They come in different forms, e.g. as Nilsson's *Teleo-Reactive programs* [7] or as *Hierarchical Task Networks* [3], although the latter are typically used to plan ahead. The *C-based Agent Behavior Specification Language* (CABSL) presented in this paper allows following the approach of modeling behavior as hierarchical finite state machines directly in C++, i.e. the language in which many robots are programmed anyway. As a result, CABSL avoids the programming overhead that usually comes from combining different programming languages.

The paper is organized as follows: First, Sect. 2 discusses behavior modeling languages and in particular CABSL's predecessor XABSL. Then, the language CABSL is presented in Sect. 3. In Sect. 4, CABSL is compared to XABSL. Section 5 discusses the impact CABSL had so far. Finally, the paper concludes in Sect. 6.

H. Akiyama et al. (Eds.): RoboCup 2017, LNAI 11175, pp. 135–142, 2018.
https://doi.org/10.1007/978-3-030-00308-1_11

2 XABSL

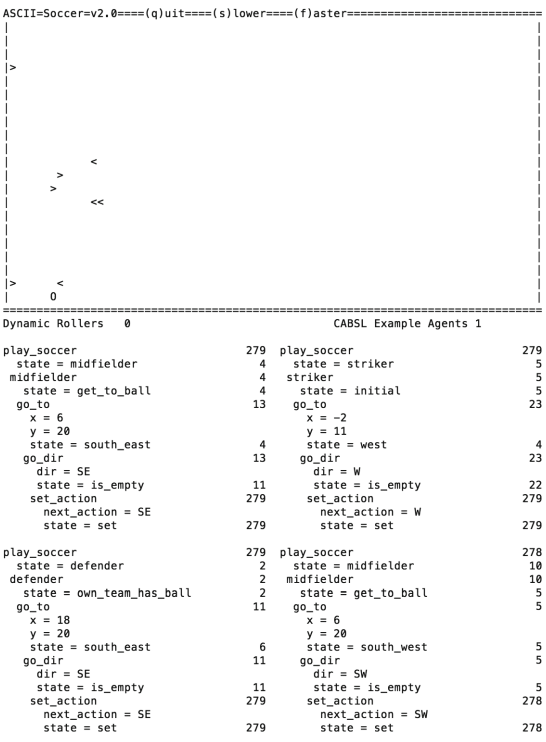
Many ways have been developed to specify the behavior of a software agent or a robot. Risler [12] gives a good overview of the different approaches in his PhD thesis. The approach that is most similar to the system presented in this paper is the *Extensible Agent Behavior Specification Language* (XABSL) developed by Lötzsch *et al.* [6]. It initially allowed specifying behavior using XML and used XML Schema to compile the source code to an intermediate representation that was interpreted by the *XABSL Engine* at runtime. Bastian Schmitz later developed a compiler in Ruby that understands a more C-like language and creates the same intermediate code [10]. It was later integrated into the official XABSL release. Risler later also integrated multi-agent features into the language [12]. The XABSL release [13] comes with an example behavior for the ASCII Soccer simulator by Balch [1], which was also used as an example for the CABSL release described in this paper (cf. Fig. 1).

```
option(midfielder) {
  initial_state(get_to_ball) {
    transition {
      if (ball_distance <= 3)
        if (westmost_teammate_x
            > ball_x + 2)
          goto dribble;
        else
          goto pass;
    }
    action {
      go_to(ball_x, ball_y);
    }
  }

  state(pass) {
    transition {
      if (ball_distance > 3)
        goto get_to_ball;
    }
    action {
      pass();
    }
  }

  state(dribble) {
    transition {
      if (ball_distance > 3)
        goto get_to_ball;
    }
    action {
      dribble();
    }
  }
}
```

(a) Option *midfielder*



(b) ASCII Soccer with CABSL activation graphs

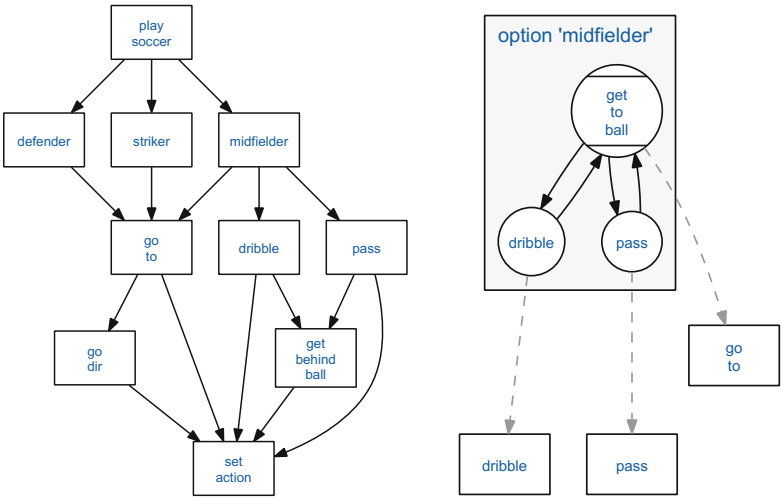
Fig. 1. Behavior originally developed by Lötzsch *et al.* [6] for the ASCII Soccer simulator by Balch [1] ported to CABSL

3 Approach

A robot control program is executed in cycles. In each cycle, the agent acquires new data from the environment, e.g. through sensors, runs its behavior, and then executes the commands the behavior has computed, i.e. the agent acts. This means that a robot control program is a big loop, but the behavior is just a mapping from the state of the world to actions that also considers what decisions it has made before.

3.1 Options

CABSL describes the behavior of an agent as a hierarchy of finite state machines, the so-called *options*. Each *option* consists of a number of *states*. Each state can define transitions to other states within the same option as well as the *actions* that will be executed if the option is in that state. One of the possible actions is to call another option, which lets all options form a directed acyclic graph (cf. Fig. 2a). In each execution cycle, a subset of all options is executed, i.e. the options that are reachable from the root option through the actions of their current states. This current set of options is called the *activation graph* (actually, it is a tree). Starting from the root option, each option switches its current state if necessary and then it executes the actions listed in the state that is then active. The actions can set output values or they can call other options, which again might switch their current state followed by the execution of actions. Per execution cycle, each option switches its current state at most once.



(a) The option graph (b) The option *midfielder* (cf. Fig. 1a)

Fig. 2. The option graph

3.2 States

Options can have a number of states, e.g. *get_to_ball*, *pass*, and *dribble* in the example (cf. Figs. 1a, 2b). One of them is the *initial state*, in which the option starts when it is called for the first time and to which it falls back if it is executed again, but was not executed in the previous cycle, because no other option called it. There are two other kinds of special states, namely *target states* and *aborted states*. They are intended to signify to the calling option that the current option has either succeeded or failed. The calling option can determine, whether the last sub-option it called has reached one of these states. It is up to the programmer to define what succeeding or failing actually mean, i.e. under which conditions these states are reached.

3.3 Transitions

States can have *transitions* to other states, e.g. *get_to_ball* has transitions to *pass* and *dribble* (cf. Fig. 1a). They are basically decision trees, in which each leaf contains a `goto`-statement to another state. If none of these leafs is reached, the option stays in its current state.

An option can have a *common transition*. Its decision tree is executed before the decision tree in the current state. The decision tree of the current state functions as the `else`-branch of the decision tree in the common transition. If the common transition already results in a change of the current state, the transitions defined in the states are not checked anymore. In general, common transitions should be used sparsely, because they conflict with the general idea that each state has its own conditions under which the state is left.

3.4 Actions

States have *actions* that are executed if the option is in that state. They set output symbols and/or call other options, e.g. execute *go_to* in the state *get_to_ball* (cf. Figs. 1a, 2b). Although any C++ statements can be used in an action block, it is best to avoid control statements, i.e. branching and loops. Branching is better handled by state transitions and loops should not be used in the behavior, but rather in functions that are executed before or after the behavior or that are called from within the behavior. It is allowed to call more than one sub-option from a single state, but only the last one called can be checked for having reached a target state or an aborted state.

3.5 Symbols

All options have access to the member variables of the C++ class in the body of which they are included, e.g. *ball_distance*, *westmost_teammate_x*, *ball_x*, and *ball_y* (cf. Fig. 1a). These member variables function as *input symbols* and *output symbols*. Input symbols are usually used in the conditions for state transitions. Output symbols are set in actions.

There are four predefined symbols that can, e.g., be used in the conditions for state transitions:

`option_time` is the time the option was continuously active, i.e. the time since the first execution cycle it became active without being inactive afterwards. The unit of the time depends on the time values passed to the behavior engine. For instance, the time measure used in the ASCII Soccer behavior is just the number of execution cycles. The option times are shown behind the option names in Fig. 1b.

`state_time` is the time the option was continuously in the same state, i.e. the time since the first execution cycle it switched to that state without switching to another one afterwards. The `state_time` is also reset when the option becomes inactive, i.e. the `state_time` can never be bigger than the `option_time`. Again, the state times are shown behind the state names in Fig. 1b.

`action_done` is true if the last sub-option executed in the previous cycle was in a *target state*.

`action_aborted` is true if the last sub-option executed in the previous cycle was in an *aborted state*.

3.6 Parameters

Options can have *parameters*. From the perspective of the option, they are not different from input symbols. As in C++, parameters hide member variables of the surrounding class with the same name. When calling an option, its actual parameters are passed to it as they would be in C++.

However, parameters must be streamable into a standard *iostream* as text, because they are added to a data structure that allows visualizing the activation graph. Therefore, any parameter of a non-primitive datatype `P` must overload `std::ostream& operator(std::ostream&, P)`. The visualization of options with parameters can be seen in Fig. 1b for the options `go_to(int x, int y)`, `go_dir(Action next_action)`, and `set_action(Action next_action)`. `Action` is an enumeration type the values of which (mainly cardinal directions) can only be shown literally in the activation graph, because a specialized streaming operator was defined. As can be seen in the following section, the syntax of parameter definitions is different from the syntax C++ normally uses, because the C++ preprocessor must be able to extract them.

3.7 Grammar

In the following grammar, `<C-ident>` is a normal C++ identifier. `<C-expr>` is a normal C++ expression that can be used as default value for a parameter. `<C-ifelse>` is a decision tree. It should contain `goto` statements to other states (names of states are C++-labels). `<C-statements>` is a sequence of arbitrary C++ statements.

```
<cabsl>       = { <option> }

<option>      = option '(' <C-ident> { ',' <param-decl> } { ',' <param-dflt> } ')'
                '{'
                [ common_transition <transition> ]
                { <other-state> } initial_state <state> { <other-state> }
                '}'

<param-decl>  = '(' <type> ')' '␣' <C-ident>

<param-dflt>  = '(' <type> ')' '(' <C-expr> ')' '␣' <C-ident>

<other-state> = ( state | target_state | aborted_state ) <state>

<state>       = '(' C-ident ')'
                '{'
                [ transition <transition> ]
                [ action <action> ]
                '}'

<transition>  = '{' <C-ifelse> '}'

<action>      = '{' <C-statements> '}'
```

3.8 Code Generation

Technically, the C++ preprocessor translates each option to a member function (two for options with parameters) and a member variable of the surrounding class. The member variable stores the context of the option, e.g. which is its current state. The context is passed as a hidden parameter to each option with the help of a temporary wrapper object. Each state is translated to an `if`-statement that checks whether the state is the current one and that contains the transitions and actions. CABSL uses C++ labels as well as line numbers (`__LINE__`) to identify states. Therefore it is not allowed to write more than one state per line. Each state defines an unreachable `goto` statement to the label `initial_state`, which is defined by the initial state. Thereby, the C++ compiler will ensure that there is exactly one initial state if the option has at least one state. The compiler can also warn if there are unreachable states.

As all options are defined inside the same class body, where C++ supports total visibility, every option can call every other option and the sequence in which the options are defined is not important. In addition, each option has access to all member variables defined in the surrounding class. Each option sets a marker in its context whether its current state is a normal, target, or aborted state. It also preserves that marker of the last sub-option it called for the next execution cycle so that the symbols `action_done` and `action_aborted` can use it. The context also stores when the option was activated (again) and when the current state was entered to support the symbols `option_time` and `state_time`.

4 Comparison to XABSL

There are several advantages of CABSL over XABSL. The first is the very small coding overhead when using CABSL. In XABSL, all input symbols and output symbols need to be registered with the engine both on the C++ side and on the XABSL side. In CABSL, all members of the surrounding class can directly

be used. XABSL only supports the datatypes *boolean*, *double*, and *enumeration*. In contrast, CABSL supports any datatype that can be used in C++. CABSL can also perform any kind of C++ computation, while XABSL requires moving computations that exceed its expressiveness to external C++ functions that again have to be registered with the engine. As CABSL is just C++, IDEs can offer auto-completion of identifiers, check the code while typing, and their integrated debuggers can be used. In addition, no custom build step is required.

However, there are also a number of drawbacks of using CABSL instead of XABSL. While XABSL has a fixed syntax and the XABSL compiler makes sure that the programmer follows it, CABSL is just a set of C++ macros that can also be used in ways that violate the intended grammar, but are still accepted by the C++ compiler. All further drawbacks listed here were never relevant for the author's team, because they concern features that were never used. CABSL is limited to C++, while XABSL also provides a Java implementation of the engine. XABSL can generate an extensive documentation in HTML. However, this feature was more useful when XABSL sources still used the XML format, which made them hard to read. For use in publications, CABSL comes with a script that can generate graphs as shown in Fig. 2. The XABSL engine supports replacing the behavior on the fly, i.e. it is possible to upload a new behavior specification to a robot while the code is running. This requires some external infrastructure for sending the code to the robot, but is in general easy to implement. CABSL does not offer a similar feature, but if desired, it would be possible to compile the CABSL behavior into a shared library, which could then be exchanged at runtime. In fact, de Haas *et al.* [4] follow a similar approach for their behavior specification language that was also compiled to native code.

5 Impact

The team B-Human has used CABSL since 2013 and has become world champion twice with behaviors written in it [9]. Some teams who base their robot control software on B-Human's yearly code releases also use it, e.g. [11,14]. For instance, the Nao Devils Dortmund state in their team report [5]: "Since 2013, we use CABSL which [...] implements XABSL as C++ macros allowing for easy access to all data structures...". The Bembelbots Frankfurt, who do not use B-Human's base system, write in their report [2]: "...later replaced by the commonly used XABSL language. Several drawbacks let us move further to the C-implementation published by B-Human [...]. Old behavior models could easily be reimplemented...". The standalone release of CABSL is exactly meant for such users, who just want to specify a robot behavior, but who do not want to use the whole B-Human system.

6 Conclusion

The paper presented the behavior specification language CABSL. It is an easy way to specify robot behavior in the form of hierarchical finite state machines

in C++. It has already been used by several other RoboCup teams. It is now available as an open source standalone release that allows using its most recent version without using the B-Human framework [8].

References

1. Balch, T.: The ASCII robot soccer homepage (1995). https://www.cs.cmu.edu/~trb/soccer
2. Brast, J.C., et al.: Team report 2016 (2016). http://www.jrl.cs.uni-frankfurt.de/web/wp-content/uploads/2016/12/TeamReport_Bembelbots_2016.pdf
3. Georgievski, I., Aiello, M.: An overview of hierarchical task network planning. CoRR abs/1403.7426 (2014). http://arxiv.org/abs/1403.7426
4. de Haas, T.J., Laue, T., Röfer, T.: A scripting-based approach to robot behavior engineering using hierarchical generators. In: Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA). IEEE (2012)
5. Hofmann, M., et al.: Nao Devils Dortmund team report 2016. Technical report. Technische Universität Dortmund (2016). https://github.com/NaoDevils/CodeRelease2016/blob/master/TeamReportNaoDevils2016.pdf
6. Lötzsch, M., Risler, M., Jüngel, M.: XABSL - a pragmatic approach to behavior engineering. In: Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS), pp. 5124–5129, Beijing, China (2006)
7. Nilsson, N.: Teleo-reactive programs for agent control. J. Artif. Intell. Res. **1**, 139–158 (1994)
8. Röfer, T.: CABSL - C-based agent behavior specification language (2017). https://github.com/bhuman/CABSL
9. Röfer, T., et al.: B-Human team report and code release 2013 (2013). https://www.b-human.de/downloads/publications/2013/CodeRelease2013.pdf
10. Röfer, T., et al.: Germanteam robocup 2005 (2005). http://www.informatik.uni-bremen.de/kogrob/papers/GT2005.pdf
11. Riccio, F.: Programming NAO-robots (2014). http://www.dis.uniroma1.it/~nardi/Didattica/CAI/matdid/RobotProgrammingNao.pdf. Slides from a lecture at Sapienza Universitá di Roma
12. Risler, M.: Behavior control for single and multiple autonomous agents based on hierarchical finite state machines. Fortschritt-berichte vdi, Technische Universität Darmstadt, 15 May 2009. http://tuprints.ulb.tu-darmstadt.de/2046
13. Risler, M., Lötzsch, M., Jüngel, M., Krause, T.: XABSL - the extensible agent behavior specification language (2009). http://www.xabsl.de
14. Sterner, N.A.: Behavior programming of the goal keeper using CABSL, semester thesis, nomadZ project reports, ETH Zürich (2014). https://www1.ethz.ch/robocup/ProjectReports/2014_Nico_Behavior_Programming_of_the_Goalkeeper_Using_CABSL