# XRobots: A Flexible Language for Programming Mobile Robots Based on Hierarchical State Machines

Steve Tousignant, Eric Van Wyk, and Maria Gini

***Abstract*— This paper introduces a domain-specific language for programming mobile robots that is based on hierarchical state machines. Following Brooks, we refer to states as behaviors. A novelty of this language is that behaviors are treated as first class objects in the language and thus they can be passed as arguments to other parameterized behaviors. The language has template behaviors which allow generalized behaviors to be customized and instantiated. This makes the language quite flexible in terms of programming styles. An example of its flexibility are presented, followed by a description of the challenges in the language design.**

## I. Introduction

As advances in robotics have allowed mobile robots to gain greater complexity and therefore be used to address more challenging tasks, programming them in a general purpose programming language has become a more arduous job. Even the simplest robotic program needs to take input from the sensors, run it through some type of a control algorithm, and write the output to the actuators of the robot. More complex algorithms may add other steps, such as preprocessing the sensory inputs, building a Brooksian subsumption architecture [1] into the control algorithm, or doing more complex computations.

Thus, writing robotic programs in a general purpose programming language poses a number of challenges. For example, sensors and actuators tend to be used as global variables which makes *modularity* difficult. The conceptual pattern that a given stimuli causes a given reaction becomes difficult to trace in the code. Any problem involving states, transitions, sensors, and actuators leads to complex representation in an imperative language.

These challenges are not unlike those found in other areas of software development in which domain-specific programming languages have been proposed (Van Deursen [2] presents a nice survey). Thus, to address these challenges we propose a domain-specific language (DSL) called XRobots. The concept of a DSL is neither new nor uncommon in language design. An advantage of DSLs is that programmers can write code at a higher level of abstraction and in the notation of the problem domain.

XRobots is based on hierarchical state machines (HSMs). HSMs have their origins in the STATECHARTs introduced by Harrel [3] and their evolution is documented by Yannakakis [4]. They have been used in several areas of computer science and are widely used in the engineering fields. HSMs use states and transitions as do regular finite state machines (FSMs), and the states typically contain a set of actions which occur on the *entry* of a state and another set of actions that occur on the *exit* of a state. HSMs extend FSMs by introducing the notion that states can be nested in a hierarchical manner. Therefore, a HSM can be in multiple states simultaneously as long as those states have a parent-child relationship.

XRobots facilitates a high degree of code reuse. Modules can be designed such that there is a clear separation between hardware-control behaviors and hardware-independent behaviors which specify the algorithm. Information can be passed between behaviors that interface with the hardware and those that do not in several ways.

The ability to parameterize behaviors and treat behaviors as first class objects allows programmers to pass information from one behavior to the next. We allow parameters to be passed in two ways, *by-value* and *by-reference*, as is done in C++. Since behaviors are first class objects, they can also be passed into other behaviors either by-value or by-reference. This allows a programmer to write a general behavior for robotic algorithms and plug in hardware specific behaviors in order to customize that algorithm for a specific robot.

We present XRobots using the example of the `followWall` algorithm instantiated with hardware-specific behaviors and discuss its execution in Section II. We describe behaviors in Section III and discuss the nuances of behavior passing in IV. Template behaviors, described in Section V, are another feature of the language that allows one to instantiate behaviors to create a customized behavior that can be invoked at some future step in the algorithm. While behaviors passing and template behaviors are advantages of this programming style, they do present certain challenges, which we discuss in Section VI. Section VII summarizes related work. The foundational work presented in this paper provides continuing research opportunities, which are discussed in Section VIII.

## II. Example of XRobots Program Execution

We begin describing an algorithm to follow a wall for the iiRobot Create robot. Our design for the algorithm combines hardware-specific behaviors, with an algorithm specific behavior. The state machine defined by the algorithm specific behavior is shown in Fig. 1, and most of the code is shown later in Fig. 5

The algorithm first enters the initial behavior `FindWall` (denoted by the small arrow). After a short delay, it transitions to the hardware-specific behavior `senseWall`, which

S. Tousignant, E. Van Wyk, and M. Gini are with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 stousig@cs.umn.edu, evw@cs.umn.edu, gini@cs.umn.edu

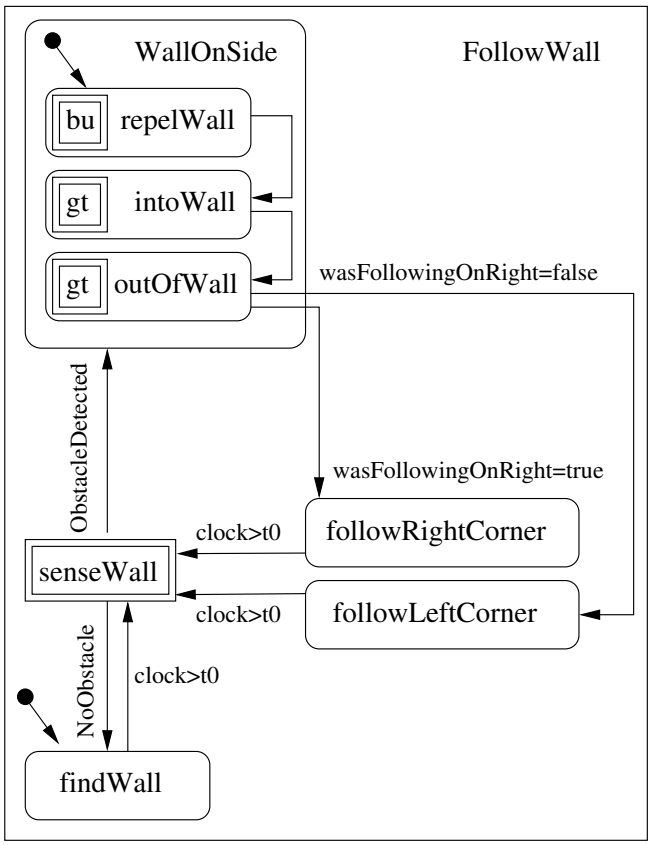


Fig. 1. High-level sub-behaviors for the Follow-Wall program

is passed into the `followWall` behavior as a parameter. Because `senseWall` is a hardware-specific behavior argument to `followWall`, we represent it with double rectangles in the diagram. As a note, in this example we use other hardware-specific behaviors. `senseWall` transitions to the `WallOnSide` behavior when an obstacle is detected, as indicated by the condition labelling that directed edge. Otherwise it transitions back to `findWall`. Once it has transitioned to `wallOnSide`, due to the detection of an obstacle, it transitions through the three sub-behaviors; as it transitions from one of these sibling behaviors to another, it must exit the former and enter the latter. It will continue iterating over those sub-behaviors until it fails to detect the wall it is following. While following the wall, the program stores `true` in a boolean if the wall was on its right, and `false` otherwise. We call this boolean `wasFollowingOnRight`. Immediately after losing the wall, it will follow the left or right corner, depending on the value of `wasFollowingOnRight`. If another wall is found, presumably after the corner is navigated, it will resume iterating the `wallOnSide` behavior otherwise, if no wall is found after the suspected corner, the algorithm will return to the `findWall` behavior.

With the example in mind, we define some general nomenclature. At any specific point in the execution of the program, we say that any behavior that has been entered, but not exited is an *active behavior*. Furthermore, the behavior that was most recently entered (but not exited) is the *current behavior*. So in this example, when `intoWall` is the current behavior both `wallOnSide` and `followWall` are active behaviors. The order in which behaviors are entered and exited follows a first-in-last-out ordering and thus we can view behaviors as being pushed onto a stack when they are entered and popped off when they exit. The top element of this stack of active behaviors is the current behavior.

Note that every program has a `root` behavior which is, by definition, an ancestor of all the behaviors in the program. The parameters to `root` are the values of the robot's sensors and references to its actuators (see Fig. 6).

## III. Example of The XRobots Behavior Construct

*Behaviors* are the fundamental building-block of XRobots. They correlate to states in HSMs and thus can be nested in XRobots. The language provides the option of labeling one of the behaviors specified in another as an *initial* behavior to be entered when the parent is also entered. We show the code from `repelWall` in Fig. 2 as an example; this behavior corresponds to the `repelWall` behavior in Fig. 1.

```
/* #1 name and parameter list */
Behavior repelWall () {
  /* #2 Empty declaration Block */

  /* #3 Entry Block */
  Entry {
    clock := 0.0;
  }
  /* #4 Transition block */
  Under Condition  clock > 0.2
    Apply Behavior
      outOfWall(wasFollowingOnRight);
  Under Condition True
    Apply Behavior bu(200.0, 1);

  /* behaviors outOfWall and bu not shown */

  /* #5 Empty Exit block */
  Exit { }
}
```

Fig. 2. A simple behavior to move away from a wall

This example, in Fig. 2, illustrates the following components of a behavior which are referenced by numbered comments.

1) Name and parameter list: Every behavior has a name and a list (possibly empty) of parameters, which is a list of typed formal arguments which need to be passed into the behavior when it is called. `repelWall` has no parameters, but if it did there would be a list of names and types in the parenthesis similar to C++ syntax. Also like C++, we can pass parameters by-value or by-reference. The qualifiers `ByRef` and `ByVal` indicate how the parameter will be passed. If no qualifier is present, the default of pass-by-value is used. Passing references and values will be discussed later.
2) Declaration block: A list of declarations, including sub-behavior definitions, used in the behavior. Variables

defined within a behavior are local to that behavior and visible to any sub-behavior. In the example the declaration block is empty.

3) Entry Block: A block of statements that are executed when the behavior is entered. In this case we set the variable `clock`. This variable is declared in the ancestor behavior `followWall`, shown in Fig. 5.
4) Transition block: A list of transitions, each consisting of a boolean expression used as a condition and a behavior invocation. When the condition evaluates to `true` the transition is *enabled*. Transitions are written in order of priority, So the first enabled transition will be invoked. In this case, we pass `wasFollowingOnRight` into `outOfWall` and `200.0` and `1` into `bu`. `bu` is the hardware-specific behavior argument for backing up.
5) Exit Block: A block of statements that are executed when the behavior is exited. In this case the exit block is empty.

## IV. Behavior Passing

As mentioned earlier, in XRobots behaviors can be declared within other behaviors, can be parameterized, and, since they are first class objects, they can be passed into other behaviors as arguments. These arguments, regardless of type, can be declared as either pass by-reference or pass by-value. The same is true for the primitive data types supported by XRobots. Passing behaviors, however, is more interesting.

When a behavior is passed *by-value* a local copy of the behavior is instantiated. Thus it is similar to a sub-behavior of the behavior into which it was passed. In other words, it is *local* to the behavior into which it was passed in terms of how transitions operate; however, the passed-in sub-behaviors cannot access the local variables of the behavior into which it was passed. Alternatively, if we pass a behavior by-reference, we are simply passing a pointer to the behavior in the program hierarchy. When we transition to the referenced behavior, we simply follow that pointer and transition to the target behavior.

Using the examples we have seen so far, we can show the two ways behaviors can be passed as arguments to other behaviors, by-value and by-reference. The hardware specific behaviors passed into the behavior `followWall` are passed by-value, (see Fig. 5). Behaviors passed by-value act as if they are sub-behaviors of the behavior into which they are passed. This feature allows us to pass into `followWall` a set of hardware specific behaviors, such as those in Fig. 3 and Fig. 7, and they are treated as sub-behaviors. The behavior `followWall` (Fig. 5) has the sub-behaviors shown in Fig. 1. This, and all the sub-behaviors of the behavior `followWall`, do not use the sensors or actuators of the robot, but hand off interfacing with the sensors to hardware specific behaviors which are passed in as parameters. We could swap one set of hardware-specific behaviors for another quite easily.

The arguments of the behavior created by the template behavior `bumpObject`, shown in Fig. 7, are good examples of passing behaviors by-reference. In this case, pointers to behaviors `wallOnSide` and `findWall` are passed in so that the behavior can later transition to these behaviors. It is noteworthy that an instance of `bumpObject` created in `root` is passed into the `senseWall` parameter of the instantiation of `followWall`, see Fig. 5. So when `intoWall` calls `senseWall`, see Fig. 5, `senseWall` is actually a locally instantiate version of `bumpObject`.

The parameters for `senseWall` and `bumpObect` are references to behaviors; as a note the parameter types for `bumpObject` have to match those for `senseWall` because we are passing the former as the actual parameter for the latter, which is the formal parameter. Now in `intoWall`, when we apply behavior `senseWall` we pass references to `wallOnSide` and `findWall`, both sub-behaviors of followWall. Since `bumpObject` will either transition to one of the two passed- in behaviors, we want it to transition to the proper behavior in its location position, hence we pass in a reference to it.

## V. Template Behaviors

*Template behaviors* outline a general, parametric structure for a behavior, but do not create a behavior until the template is instantiated in a behavior's declaration block. They have the same five components of a behavior. However, template behaviors have an additional component, a *template parameter lists*, which is a list of parameters for the template itself.

A template behavior is shown in Fig. 3. This behavior takes two template parameters, `lVel` and `rVel`, which are both references to the actuators of the differential-drive robot.

```
// From module Create
Template Behavior gentleTurn
  < ByRef float lVel, ByRef float rVel >
  ( float rspeed, float lspeed )
{
  Entry {
    rVel := rspeed;
    lVel := lspeed;
  }
}
```

Fig. 3. The gentle-turn template behavior for a differential drive robot

```
// From module v_omega
Template Behavior gentleTurn
  < ByRef float v, ByRef float omega >
  ( float rspeed, float lspeed )
{
  float radius;// a constant for the robot
  Entry {
    radius := 16.0
    v      := sqrt(rspeed^2 + lspeed^2);
    omega  := (rspeed - lspeed)/radius;
  }
}
```

Fig. 4. The gentle-turn template behavior for a v-omega robot

Using template behaviors we can generate reusable behaviors that control the sensing or actuation of a robot. In

this example, the template behavior expects to receive the references to the differential drive actuators. Then after it is instantiated, two parameters will be passed into the behavior when it is invoked which are the speed of the two wheels. In general, we pass in two unequal floats getting the robot to make a slight turn. We have written a similar template behavior to move a robot with a "v-omega" drive system; this behavior is shown in Fig. 4. The convenience here is that both have the same interface, in this case the interface of a differential-drive robot. However, if we wanted to given them both the interface of a "v-omega" robot, we easily could. In either case, the invocation of the behavior resulting from either of these template behaviors will be identical.

In Fig. 5 we show `followWall` and some of the nested behaviors that correspond to the diagram shown in Fig. 1. The outer most behavior in both is `followWall` which has the sub-behaviors `followRightCorner`, `followLeftCorner`, and `wallOnSide` in both the figure and the code. The double rectangles such as `bu` and `gt` are hardware specific behavior arguments in the code.

Template behaviors can be instantiated in a behavior's declaration block to create a sub-behavior. The program will act as if the sub-behavior had been defined at the point of instantiation in terms of stack behavior; however, the variables that are visible to the parent behavior will not be visible in the instantiated behavior since XRobots uses static scoping. Instantiation happens at runtime, unlike C++ templates which are instantiated at compile-time.

In general, we write a set of template behaviors to represent the hardware specific code and use them in specific algorithms. We have implemented a rudimentary module system that allows the user to organize and combine a generic algorithm with specific modules for hardware.

Therefore, we could *instantiate* different template behaviors inside a program to implement a follow-wall algorithm on a differential drive robot as shown in Fig. 6. The parameters to `root`, as previously mentioned, are the set of the robot's sensors and actuators used in the program. This program is intended to specialize the `followWall` algorithm for a Create robot with a differential drive. Since the `followWall` module contains a behavior, importing it makes the `followWall` behavior local to the `root`. Conversely, the `Create` module only has template behaviors, so its import gives `root` access to instantiate the hardware-specific template behaviors defined in that module. We instantiate these template behaviors in the declaration block of `root`, giving them any valid local name; in this case we chose `bu`, `gt`, `r`, `bo`. Lastly, we invoke `followWall` unconditionally passing to it the hardware specific behavior we have just instantiated.

To invoke the `followWall` behavior, (Fig. 5), we must provide the following hardware specific behaviors:

1) A backup behavior for the specific robot
2) A gentle turn behavior for the specific robot
3) A behavior which gives the specific robot some random motion
4) A behavior which senses a wall given the sensors of

```
// From module followWall
Behavior followWall
  (ByVal Behavior bu(float v, int c),
   ByVal Behavior gt(float rv, float lv,
                     bool tb),
   ByVal Behavior random(),
   ByVal Behavior senseWall (
         ByRef Behavior wallAlg(bool rb),
         ByRef Behavior clearAlg()  )
  )
{
  float clock;

  Behavior followRightCorner () {...}
  Behavior followLeftCorner () {...}
  Initial Behavior findWall () {...}

  Behavior wallOnSide (
    bool wasFollowingOnRight)
  {
    Behavior repelWall() {
      Entry {
        clock := 0.0;
      }

     Under Condition  clock > 0.2
       Apply Behavior
         outOfWall(wasFollowingOnRight);
      Under Condition True
        Apply Behavior bu(200.0, 1);
    }

    Behavior outOfWall() {
      Under Condition clock > 0.7
        Apply Behavior
          intoWall(wasFollowingOnRight);
      Under Condition True
        Apply Behavior
          gt(25.0,250.0, wasFollowingOnRight);
    }

    Behavior intoWall() {
      Under Condition
        clock > 1.7 && wasFollowingOnRight
        Apply Behavior
          senseWall(wallOnSide, findWall);
      Under Condition clock > 1.7
        Apply Behavior
          senseWall(wallOnSide, findWall);
      Under Condition
        clock > 1.2 && wasFollowingOnRight
        Apply Behavior followRightCorner();
      Under Condition clock > 1.2
        Apply Behavior followLeftCorner();
      Under Condition True
        Apply Behavior
          gt(200.0,25.0, wasFollowingOnRight);
    }

    Under Condition True
      Apply Behavior bu(100.0, 10);
    }
  }  // end of wallOnSide

} // end of followWall
```

Fig. 5. The followWall behavior

```
Import "FollowWall";
Import "Create";

Behavior root
  ( ByRef float rvel, ByRef float lvel,
    ByRef bool rbump, ByRef bool lbump )
{
  Behavior bu := backup<lbump, rbump,
                        lvel,  rvel> ;
  Behavior gt := gentleTurn<rvel, lvel> ;
  Behavior r  := random<rvel, lvel> ;
  Behavior bo := bumpObject<rbump, lbump> ;
  Under Condition True
    Apply Behavior followWall(bu, gt, r, bo);
}
```

Fig. 6. An instantiation of the follow wall behavior for the Create.

that robot.

For the Create the `senseWall` behavior is named `bumpObject`, which is shown in Fig. 7. However, the formal argument represents a hardware independent way to sense the wall, and this argument is called `senseWall` (see Fig. 5). The gentle turn template behavior for the Create is provided in Figure 3. `Backup` and `random` are not shown.

```
Template Behavior bumpObject
  < ByRef bool lbump, ByRef bool rbump >
  ( ByRef Behavior wallAlg (bool b),
    ByRef Behavior clearAlg ()  )
{
  Under Condition rbump
    Apply Behavior  wallAlg(True);
  Under Condition lbump
    Apply Behavior wallAlg(False);
  Under Condition True
    Apply Behavior clearAlg();
}
```

Fig. 7. A sense wall template behavior for the Create

The template behaviors for the hardware are instantiated inside the root behavior whose parameters are references to actuators and values of sensors. This was previously described and shown is Fig. 6.

We have described the major capabilities of XRobots. We view the ability to write software for various robotic platforms by combining sets of template behaviors as a strength of the language. It allows roboticists to encapsulate hardware-specific code in one set of modules and robotic algorithms in another. A simple module system is used to enable reuse of the interfaces to various robotic hardware. Documenting the entire syntax and semantics of XRobots is beyond the scope of this paper. However, the denotational semantics is detailed in an unpublished manuscript report [5].

For our implementation, we have chosen to use Silver [6], an attribute grammar system, to translate XRobots programs into Haskell code and to use C for the interface between the Haskell program and the robotic hardware itself. The C program collects sensor data from the robot and passes them through a pipe to the Haskell program. When the Haskell program returns data, the interface captures them and uses the actuator values to control the robot. The key to this interface in the bidirectional conversion between Haskell and C data types. We have prototypes that interface both with Player/Stage and the Create API. By using Haskell, we use a modern, functional language in robotics. Historically, functional languages where considered a key tool in AI and robotics, and we see value in their continued use.

## VI. Challenges of the Language

With the added expressivity of passing behaviors by-value and by-reference come some challenges in language design. We describe here the three most important:

1) ensuring that invoking a by-reference behavior follows standard rules;
2) error detection when passing by-value behaviors;
3) dealing with the pragmatics of the language.

Transitions with a by-reference target behavior should use the same protocol as transitions where the target behavior has been hard coded, such as `outOfWall` in the transition block of Figure 2. The only difference, and thus the challenge, is retrieving the reference to the behavior so that it is in a similar form as a hard coded behavior. We must, for example, not attempt to transition to a behavior that is nested several layers deeper in the hierarchy than the current depth. What we wish to avoid is having one algorithm for each case since they fundamentally do the same operation.

Passing behaviors by-value introduces the possibility for some interesting errors to arise. Say, for instance, we have a behavior that accesses variables from an ancestor behavior, and we pass that behavior by-value into a behavior in another branch of the hierarchy. When that other branch tries to make that behavior the current behavior, the ancestor variables it tries to access may not be accessible if the ancestor behavior that defined them is no longer active. It will be fairly straight forward at runtime to throw an error that the variable is not defined, but it would be preferable to detect this situation at compile time and flag the error at that point.

The last challenge deals with the pragmatics of the language. We are unsure how easy it may be for the programmer to reason about programming in this model. The higher-order nature of the language (i.e. behaviors as first class objects) may be problematic to people who are not used to such conventions. This claim may be especially true for those whose background is solely in imperative programming. Therefore, one of the challenges for us as language developers is how to minimize this barrier.

## VII. Related Work

We examine briefly programming languages whose main purpose is to simplify robotic programming. Such languages can mostly be classified as either reactive languages, or imperative languages, or languages based on a standardized middle-ware, such as CORBA. Domain specific languages are starting gaining popularity in the robotics community, because they promise to simplify the process of developing the large and complex programs that are needed for robots.

In a major change from the approaches that were commonly used in robotics, Brooks [7] introduced the subsumption architecture, an architecture based on layers of components connected to each other, that operate on sensor data and produce control commands to the robot. The components use "inhibition" and "suppression" mechanisms to override input or output from other components, enabling the building of complex programs that are scalable and modular. Brooks later introduced the Behavior Language to make it easy to implement his subsumption architecture [8]. Since the subsumption architecture is based on Augmented Finite State Machines (AFSM), so is the language.

Player/Stage [9] is currently the most widely used public domain software for programming real robots and for simulating them. The Robotic Operating System (ROS) [10] provides a standardized interface between robotic algorithms and hardware. Popular packages, such as Player, can be wrapped and used in ROS. Its developers argue its advantages include being: thin, it is small memory-wise; peer-to-peer, it does not require a central server; multi-lingual; tool based, a large set of small tools is used to handle the workflow; and free and open source.

A survey of development environments for robotics is in [11]. ASEME (Agent Systems Engineering Methodology) [12] has been proposed recently for developing software for agents. The approach uses the model-driven engineering paradigm, which relies on model transformations, and is intended to cover all the phases of the design and development of software for a complex distributed system of agents. ASEME is being used to program robots for RoboCup.

XABSL [13] is a recent example of an extensible behavior specification language designed for robotics. The language has been used for multiple robotics platforms, most notably to program robots for RoboCup.

Reckhauss et al. [14], proposed a "platform independent model" coupled with a "platform specific model." They develop a platform independent model to control a whole array of robots, and a platform specific model to control each specific robot. Clearly this is a way to handle robot heterogeneity. However, each platform specific model can have its own syntax so you may end up with a number of related, but disparate languages with identical semantics. The authors cite this as an example of model driven development.

We presented a preliminary version of XRobot with no discussion of how to make programs hardware independent and of programming styles in [15].

## VIII. Further Work and Conclusions

We currently have a working version of the compiler for XRobots built in Silver [6]. The compiler generates Haskell code, which can interface with C code to control the robot directly or via some other platform such as Player/Stage.

We intend to expand this work on several fronts. We would also like to build a simple simulator so that we can test programs while examining the internal flows of data. Visual inspection of a robotic program may not be sufficient, but tracing though a program step by step will be helpful.

We intend to test the language by comparing it to some known code base. A possibility is using the challenge problems from the CURIE project [16] and comparing both our results and the quality of our code to their findings. Eventually, we will look for user feedback on the ease of development in our language.

In this paper we have discussed the benefits of a language for programming mobile robots based on an augmented HSM model. What augments the HSM model is the use of parameterized behaviors (states), the ability to treat behaviors as first class objects, and template behaviors, which allow the user to instantiate a behavior based on a predefined template. The programming model has potential advantages over the state of the art in that there is some history of using state-base behavioral models in robotics and it has significant higher-order capabilities. We have defined the language and described how the language functions. The opportunities and difficulties of passing around behaviors as by-value and by-reference parameters has been discussed.

## References


[1] R. A. Brooks, "A robust layered control system for a mobile robot," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1985.

[2] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, pp. 26–36, 2000.

[3] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231–274, June 1987.

[4] M. Yannakakis, "Hierarchical state machines," in *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, ser. Lecture Notes in Computer Science, J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, Eds. Springer Berlin / Heidelberg, 2000, vol. 1872, pp. 315–330.

[5] S. Tousignant and E. Van Wyk, "The syntax and semantics of XRobots," an unpublished language specification. [Online]. Available: http://wwww-users.cs.umn.edu/~stousig/synsem.pdf

[6] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: an extensible attribute grammar system," *Science of Computer Programming*, vol. 75, no. 1–2, pp. 39–54, January 2010.

[7] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14 – 23, Mar. 1986.

[8] ——, "The behavior language: User"s guide," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1990.

[9] B. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Int'l Conf. on Advanced Robotics*, Coimbra, Portugal, June 2003.

[10] M. Quigley, K. Conley, B. P. 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*, 2009.

[11] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, pp. 101–132, 2007.

[12] M. P. Spanoudakis N., "Modular JADE agents design and implementation using ASEME," in *IEEE/WIC/ACM Int'l Conf. on Intelligent Agent Technology*, Toronto, Canada, 2010.

[13] M. Lötzsch, M. Risler, and M. Jüngel, "XABSL - A pragmatic approach to behavior engineering," in *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, Beijing, China, 2006, pp. 5124–5129.

[14] M. Reckhaus, N. Hochgeschwender, P. G. Ploeger, and G. K. Kraetzschmar, "A platform-independent programming environment for robot control," in *1st Int'l Workshop on Domain-Specific Languages and models for Robotic systems (DSLRob10)*, Oct. 2010.

[15] S. Tousignant, E. V. Wyk, and M. Gini, "An overview of XRobots: A hierarchical state machine based language," in *Workshop on Software development and Integration in Robotics, IEEE Int'l Conf. on Robotics and Automation*, May 2011.

[16] [Online]. Available: http://web.mae.cornell.edu/hadaskg/outreach/curie2010.html