An Overview of XRobots: A Hierarchical State Machine Based Language

Steve Tousignant, Eric Van Wyk, Maria Gini

***Abstract*— This paper introduces a prototype domain-specific language for programming mobile robots that is based on hierarchical state machines. A novelty of this language is that states are treated as first class entities in the language and thus they can be passed as arguments to other parameterized states. The structure and behavior of the language is presented, along with an example program. Further work and language design challenges are also discussed.**

I. Introduction

As advances in robotics have allowed small mobile robots to gain greater complexity and therefore be used to address more challenging tasks, programming them in a general purpose language becomes a more arduous job. The difficulty of this task is due to the fact that 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. However, more complex algorithms may add other steps such as preprocessing the sensory inputs or building a Brooksian subsumption architecture [1] into the control algorithm.

Thus, writing robotic algorithms in a general purpose programming language poses a number of challenges. For example, sensors and actuators, as realized in imperative programs, tend to be global variables which makes *modularity* difficult. The conceptual pattern that a given stimuli causes a given reaction becomes difficult to trace in the code. To state the problem more generally, the problem space has little correlation to the solution space.

These challenges are not unlike those found in other areas of software development in which domain-specific programming and modeling languages have been proposed (see [2] for a survey). Thus, to address problems in developing software for mobile robots we propose a domain-specific language called XRobots that is based on hierarchical state machines (HSMs). HSMs have their origins in the STATECHARTs introduced by Harrel [3] and their evolution is documented in [4]. They have been used in several areas of computer science and are widely used in the engineering fields. HSMs have states and transitions them as in regular finite state machines (FSMs) and in models of these 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 build on the concepts of FSMs by introducing the notion that states can be nested in a hierarchical manner. Therefore, a HSM can be in multiple states simultaneously so long as those states have a parent-child relationship.

Using an HSM-based language mirrors the way many researchers think about problems from the mobile robots domain. In XRobots the states in a HSM specify the *behaviors* that the robot is expected to exhibit. This seems natural since roboticists often think in terms of a robot's behaviors. Behaviors, as a language construct in XRobots, can read from and write to a number of variables on entering or exiting the behavior; these variables can include the actuators and sensors of the robot. Since our behaviors are based on the states of HSMs, the robot will remain in a behavior until one of its transitions takes it out of that behavior/state. Transitions specify a boolean condition and the target behavior, which may include parameters that are passed to that behavior. When its condition becomes true, the transition is *enabled*. Since we have based the language on HSMs, to transition from one behavior to another we will need to exit one set of behaviors, i.e. those that constitute the current state, and enter another behavior.

The aspect that we add to HSMs is the ability to parameterize behaviors and treat behaviors as first class structures. Therefore, we can pass information from one state to the next. We allow parameters to be passed in two ways, by *value* and by *reference*. These terms have the common meaning associated with them in programming languages: when something is passed by value a copy is created and passed to the called behavior; when something is passed by reference we use a reference to refer back to its run-time value and location. Since behaviors are first class structures, they can also be passed into other behaviors either by value or by reference. This increases the potential for code reuse within the language.

The remainder of this paper is organized as follows. Section II gives an overview of the language itself, and section III walks the reader through an example program. We describe some of the challenges of designing such a language in Section IV. Section V is a brief overview of related work. Finally, Section VI outlines future work.

II. XRobots Language Overview

Behaviors are the fundamental building-block of XRobots. They correlate to states in HSMs and thus in XRobots can be nested. 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. Behaviors have a number of components that are illustrated in Fig. 1 and referenced by labeled comments.

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

```
/** #1 name and parameter list **/
Behavior driveStraightFor(float duration){
     /** #2 Declaration Block **/
     float newDuration;
     /** #3 Entry Block */
     Entry{
        rVel := 200.0;
        lVel := 200.0;
        newDuration = duration - 5.0;
     }

     /** #4 Transition block **/
     Under Condition duration > 0
      Apply Behavior driveStraightFor(newDuration)
     Under Condition duration <= 0
      Apply Behavior Stop() /** The behavior stop is not shown **/

     /** #5 Exit block **/
     Exit{
        rVel := 0.0;
        lVel := 0.0;
     }
}
```

Fig. 1. A simple Behavior

1) Name: every behavior has a name and standard rules of scoping of names apply. In Fig. 1 the single behavior is named `driveStraightFor`. This can be seen at the comment labeled #1.
2) Parameter List: A list of types and formal arguments which need to be passed into the behavior when it is called. See again label #1.
3) Declarations block: a list of declarations, including sub-behavior definitions, used in the behavior. See label #2.
4) Entry Block: a block of statements that execute when the behavior is entered. See label #3.
5) Transition block : A list of transitions consisting of boolean expressions and target behaviors. When the condition evaluates to true the condition is enabled and the system transitions to the next behavior. See label #4.
6) Exit Block: a block of statements that execute when the behavior is exited. See label #5.

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*. 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. As expected, such a stack is in-fact used in the implementation of XRobots. A property of the language is that if a behavior is active and thus on the stack, all behaviors above it in the stack are sub-behaviors and enclosed by it in the hierarchy. Note that all programs have a root behavior which is, by definition, an ancestor of all the behaviors in a program.

What HSMs add to FMS is the ability to nest states within states. In XRobots this is realized by behaviors as first class structures which can be declared within other behaviors. Here, behaviors can be parameterized, and since behaviors are first class structures they can by passed into other behaviors as arguments. These arguments, regardless of type, can be declared as either pass by-reference or pass by-value. For primitive data types supported by XRobots, such as integer, floating-point and boolean types, this is handled as expected and as in other languages such as C++. For behaviors, however, this is more interesting.

We start by explaining the simple case where the targets of transitions are constant named behaviors. At the beginning of the program the *root* behavior is made active and any entry code block in it is executed. When any behavior is entered, any child behavior labeled “initial” becomes active and its entry code is executed. The transition conditions of each active behavior are tested. These tests are ordered from parent to child in the HSM and in order of appearance in a single behavior. The first condition that is found to be true enables that transition. In an enabled transition, all active behaviors that are descendants of the common ancestor of the current and target behaviors are exited. As each behavior exits, its exit code is executed. Lastly, the target behavior is made active and its entry code is executed. However, if no transitions are enabled, the program does not change

behaviors. The program pauses, new sensor values are read, and the process repeats, with a potentially new set of enabled transitions. It is worth noting that while many behaviors can be exited at once, only one can be entered; so the target behavior must be a child or sibling of an active behavior.

In the above description, we assumed the target behavior is specified by name. However, since behaviors are first class structures, they can also be used as arguments to another behavior. So variables with a behavior-type may appear in the list of types that are the formal parameters of behaviors. The arguments are passed into these parameters by value unless the `ByRef` indicator is present which indicates that they are to be passed in by reference.

When a behavior is passed by reference, a reference (which can be implemented as a pointer as in languages such as C++) is stored for the name of the behavior argument of the called behavior's state machine. If the transition that uses the argument is enabled, we look up the reference of that argument and use it as the target of the transition. All the protocols of the HSM, which were describe above, are followed. This means that we locate the target in the HSM, find the common ancestor behavior of the current and the target state in the HSM, and exit and enter states as described above. The only difference here is that we have a stored variable holding the target behavior.

Alternatively, when a behavior is passed by value, a representation of the "value" of the behavior is passed. This passed-in behavior is dynamically instantiated as a new sub-behavior of the called behavior. If a transition is taken to this behavior, again, the standard XRobots rules and semantics for applying behaviors are again enforced.

Alternatively, if a behavior is passed by reference into some other behavior, the transition is handled using the static location of the behavior passed. Therefore, a transition to a by-reference behavior argument is no different than a transition where the target is written directly except for the fact that the target is not a constant but name (stored in the reference) that must be looked up.

This augmented HSM model allows the code to more easily correspond to the problem space. It is rather natural to think of mobile robotics problems in terms of behaviors. A behavior may be something like following a wall, but the behavior to follow a wall may be made up of smaller behaviors, such as finding the wall and arcing into and out of the wall. So it also seems quite natural to think of a behavior being made up of smaller behaviors and being composable into a larger behavior. Such behaviors (to use the term informally) can be specified quite naturally in XRobots.

III. Example Program

This section contins a more complex example of an XRobots program that shows how behaviors can be passed along with other features. It is shown in Fig. 2 and Fig. 3. This algorithm is designed for a simple differential-drive robot with a left and right front bumper.

The robot navigates the outline of a square four times and then the outline of a triangle four times. It continues this pattern indefinitely unless an obstacle is hit. The example has behaviors organized in the following hierarchy.

root
- *straightLine*
- *start*
- *square*
  - *rightCorner*
- *triangle*
  - *triangleCorner*
- *stop*

The behavior *root* is a container for the program. It is the outermost state in the HSM and contains the child behaviors *straightLine*, *square*, and *triangle*. These three behaviors are the nested states inside the behavior *root*. This pattern of nesting states machines within a state machine is what gives us a hierarchy. At present, programs must have a *root* behavior, though it need not be named *root*. The parameters to *root* are the robot's actuators and sensors; these are indicated by the appropriate keywords. It should be noted we are assuming a differential-drive robot and that we declare the sensors *rBump* and *lBump* and the actuators *rVel* and *lVel* as parameters to the *root* behavior, see label #1 in Fig 2. The sensors are used for obstacle avoidance. The actuators are set in the entry block of each non-root behavior, see, for example, comment #4 and #6. Additionally, *root* only has a declaration block.

The behaviors *square* and *triangle* both take an integer parameter *count* as well as a behavior *avoid*. The integer parameter keeps track of how many times each shape has been traced. Therefore it determines when to transition between behaviors. However, the behavior *straightLine* takes both a by-reference and by-value behavior, as well as a by-value integer. The by-value behavior parameter determines what the robot will do if its bump sensors detect an obstacle. In this case, the robot will call the *stop* behavior defined after comment #8 . The point of passing the obstacle avoidance behavior as a parameter is that we could easily replace *stop* with some other obstacle avoidance algorithm. Note the only place we have hard-coded the name of *stop* is in the *start* behavior at comment #6, so changing this name would be easy. The sole point of the *start* behavior is to configure the parameters of *triangle*.

The behavior parameter associated with comment #2 *nextBehavior*, is the behavior *straightLine* will call when it is finished. We determine *straightLine* is finished by the clock, which automatically increments at every time step unless otherwise rest set such as in the entry block of *straightLine*, see comment #5 in Fig 2. The value of the clock is used as part if not all of our transition conditions, see comment #4 for example. Similarly, all other transitions use some combination of the clock and the counts as conditions for transitions. We are making the simple assumption that travelling so long at a constant speed will move the robot a given distance and ignoring things like wheel slippage.

straightLine's second parameter is a by-value behavior called *avoid*, which we pass on to *triangle* and *square*. It

```
/** #1 Root behavior with sensor and actuator parameters **/
Behavior root ( sensor bool rBumb, sensor bool lBumb,
                actuator float rVel, actuator float lvel ) {

 Behavior  straightLine (
       /** #2 parameter list  using the type behavior**/
       ByRef Behavior nextBehavior(int, ByVal Behavior),
       ByVal Behavior avoid(),
       int count ) {
   Entry { /** #3 setting the clock and differential-drive**/
     clock := 0.0;
     rvel  := 100.0;
     lvel  := 100.0;
   }
   /** #4 calling a behavior specified by a by value parameter **/
   Under Condition rBump || lBump  Apply behavior avoid()
   Under Condition clock > 25.00
     /** #5 calling the behavior specified by the
              argument nextBehavior with the parameter count **/
     Apply Behavior nextBehavior(count, avoid)

  }
 /** #6 the initial behavior start initializes the system **/
 Initial Behavior start(){
  Under Condition True Apply Behavior square(0, stop)
 }
 /** #7 the behavior square takes the count of homw many squares have been
     drawn and an abstacle avoidance behavior **/
 Behavior  square(int  sq_count, ByVal Behavior aviod){
  Behavior rightCorner(int corner_count)
   Entry{
     clock := 0.0;
     rVel  := 250.0;
     lVel  := 25.0;
   }
    Under Condition clock > 7.0 && cornerCount <  4
       Apply Behavior
            straightLine(square, corner_count)
    UnderCondition clock > 7.0 &&  cornerCount == 4
      Apply Behavior square(sq_count, stop)
   Exit{
         corner_count:= corner_count + 1;
   }
  }
```

Fig. 2. Triangle Square Program (Part 1)

```
  Under Condition rBump || lBump
    Apply Behavior avoid()
  Under Condition sq_count < 4
    Apply Behavior straightLine(square,  corner_count)

  Under Condition tri_count => 4
    Apply Behavior triangle(0)

  Exit{
      sq_count := sq_count + 1;
  }
 }
 Behavior triangle(int tri_count, ByVal Behavior avoid){
  Behavior triangleCorner(int corner_count)
   Entry{
    clock := 0.0;
    rVel  := 250.0;
    lVel  := 25.0;
  }
  Under Condition clock > 15.0  & & corner_count < 3
            Apply  Behavior triangle(corner_counter)
  Under Condition clock > 15.0 && corner_count == 3
    Apply Behavior straightLine(triangle, corner_counter)

  Exit{
   corner_count := corner_count + 1;
  }
 }
 Under  Condition rBump || lBump
     Apply behavior avoid()

 Under Condition tri_count < 4
    Apply  Behavior straightLine(triangle, avoid, corner_count)

 Under Condition tri_count => 4
    Apply Behavior square(0)

 Exit
    tri_count:=tri_count+1;
 }
/** #8 Use stopping as an obstacle avoidance technique and an example of a
    Behavior passed around by value **/
 Behavior stop(){
  Entry{
   rVel := 0;
   lVel := 0;
  }
 }
}
```

Fig. 3. Triangle Square Program (Part 2)

determines the behavior to use if an obstacle is encountered. *straightLine*'s integer parameter, *count*, counts how many times each shape has been traced. It is used as input when the behavior stored in *nextBehavior*, i.e. *triangle* or *square*, is called. The parameterized behavior determines the type of shape the robot will navigate. This program is an example of most, but not all of the constructs in the language.

The behavior *rightCorner* is nested in *square*, and thus its ancestors are *square* and *root*. Because of the hierarchy, the parameters of the ancestor state *root* are visible in the child state *rightCorner*.

The behavior *triangleCorner* is a good example of the entry, transition, and exit block arranged in proximity to each other. This fact is because the declaration block, which must come before the entry, is empty. The declaration block contains the declaration of any variables within the behavior, including other behaviors. For example, *triangleCorner* is declared at the top of *triangle* in its declaration list.

This program provides a couple of examples of passing arguments. Both *triangleCorner* and *rightCorner*, take an integer argument that determines how many times the triangle or square, respectively, have been circumnavigated. When we call these behaviors, we must provide a value for the integer count. The arguments to the behavior *straightLine* are slightly more complicated since it takes both a behavior and an integer, see comment #5. Additionally the behavior argument has a parameter list of an integer, so only behaviors with one integer argument used when calling *straightLine*.

IV. 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 with constant target behavior. The only difference, and thus the challenge, is retrieving the reference to the behavior so that it is in a similar form as a constant target 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 type of transition 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 state, 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 structures) 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.

V. Related Work

This section examines 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 middleware, 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 [5] 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 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 [6]. Since the subsumption architecture is based on Augmented Finite State Machines (AFSM), so is the language. The Behavior Languages syntax is compiled into a set of AFSM, which can in turn be translated into code to run on a number of different sets of hardware.

Player/Stage [7] is currently the most widely used public domain software for programming real robots and for simulating them. Player/Stage evolved over the years from using only IP ports and low-level communication between Player and Stage or the physical robots, to a CORBA based system based on components, where Player acts as an abstraction between Stage or the hardware and a high-level programming language which is used to control the robots.

More recently, Gerkey, one of the developers of Player/Stage, introduced the Robotic Operating System (ROS) [8]. ROS provides a standardized interface between robotic algorithms and hardware. Popular packages, such as Player, can be wrapped and used in ROS. Its developer argue its advantages include: being thin – it is small memory-wise; peer-to-peer – it does not require a central server; multilingual; tool based – a large set of small tools is used to handle the workflow; and Free and open source. ROS could be thought of as the next generation of Player/Stage.

Many programming languages have been proposed for robots. For a survey of the field, see [9]. For instance, a task-level robotic programming language was presented in [10]. While the robot language itself is imperative, it is based on the reactive language ESTREL. The code is

reminiscent of classical AI plans. At the heart of the language are robot tasks which are simple plans instructing the robot to complete some procedures. Like many other languages for robots, these plans can be combined with conditions, run in sequence or in parallel, and inserted into loops. Through the use of the mechanisms, simple plans can be built into more complex plans.

A language of a different flavor is the Multiagent Robot Language (MRL), introduced in [11], which is based upon Guarded Horn Clauses and similar to the Guarded Command Language, GCL [12]. Each declaration in the language has a head, guard, and body. The head is comparable to a function or rule name. The guard and body are atomic formulas such that if the guard is entailed, the body is executed.

CORBA-based approaches have been and are still popular in robotics, because they provide components and useful services [13]. Unfortunately, they are also complex to learn and have often a non trivial overhead. Smart [14] asserts that the robotic research agenda could be significantly accelerated by a standardized robotic middleware. He claims that most research teams waste time building their own platforms when they could get straight to the heart of their research. But he does admit there are large barriers to a universal operating system such as the heterogeneity of robotic hardware, the limited computational power of most robots, and the frequency of numerous types of failures in robots. He asserts that developing such a system will take the support of the whole robotics community and a number of iterations to get it right.

Orca [15] is a framework for developing component-based software for robots. The building blocks of Orca are data objects, communication patterns, and transport mechanisms. A component is built by selecting from a standard set of these building blocks. The framework is open source so that individuals can extend it by adding new building blocks, and doing so will account for new hardware, communication protocols, etc. The authors make a point of the difference between objects and components. They define components as freestanding executables, so you do not have to compile your system if you are using pre-built components. Each component would encapsulate the behaviors of the device it controlled. In one sense such a system would be compiled as components and then assembled into the running system.

URBI [16], has similarities with older versions of Player/Stage. It is based on a client/server model where the server runs on the robot itself. Thus the server contains all the low-level interfaces with the hardware. The client can be written in any language that handles TCP sockets, but C++ or Java are most commonly used. URBI has a large user base and supports multiple hardware platforms, ranging from the Lego Mindstorm to the the humanoid NAO.

ASEME (Agent Systems Engineering Methodology) [17] 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 applied to program robots for RoboCup.

Domain specific languages are gaining popularity in robotics. Wellborn [18] presents a DSL for robotics embedded in Java. His DSL extends java with Resources, Coordinator, Portals, etc, that allow broadcast, client/server, and peer-to-peer communication between robots. His primary goal is to decrease the amount of communication needed in these applications. His contribution is more to the area of distributed computing than to robotics. In fact, the author leaves it as future work to get his system running on real robots.

XABSL [19] 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. [20], present an example of a Platform Independent Model (PIM) coupled with a Platform Specific Model (PSM). They develop a PIM to control a whole array of robots, and a PSM to control each specific robot. Clearly this is a way to handle robot heterogeneity. However, each PSM 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.

VI. Further Work and Conclusion

We intend to expand this work on several fronts. The first, and most obvious steps are formalizing the semantics and completing a stable version of the compiler. We would also like to build a 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 know code base. A possibility is using the challenges design for the CURIE project [21] and comparing both our results and the quality of our code to there findings. Eventually, it would also be nice to get user feedback on the ease of development in our language.

In this paper we have introduced a language for programming mobile robots based on an augmented HSM model. What augments the HSM model is the use of parameterized behaviors (states) and the ability to treat behaviors (states) as first class structures. The programming model has potential advantages over the state of the art in that it more closely resembles the problem space and has significant higher-order capabilities. We have shown the syntax of the language and conceptually how the language will function. 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, June 2000. [Online]. Available: http://doi.acm.org/10.1145/352029.352035

[3] D. Harel, “Statecharts: A visual formalism for complex systems,” *Sci. Comput. Program.*, vol. 8, pp. 231–274, June 1987. [Online]. Available: http://portal.acm.org/citation.cfm?id=34884.34886

[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] 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.

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

[7] 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. [Online]. Available: citeseer.ist.psu.edu/gerkey03playerstage.html

[8] 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.

[9] J. Kramer and M. Scheutz, “Development environments for autonomous mobile robots: A survey,” *Autonomous Robots*, vol. 22, pp. 101–132, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10514-006-9013-8

[10] E. Coste Maniere, B. Espiau, and É. Rutten, “Task-level robot programming combining object-oriented design and synchronous approach : a tentative study,” INRIA, Research Report RR-1441, 1991. [Online]. Available: http://hal.inria.fr/inria-00075119/PDF/RR-1441.pdf

[11] H. Nishiyama, H. Ohwada, and F. Mizoguchi, “A multiagent robot language for communication and concurrency control,” in *Int’l Conf. on Multi Agent Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 1998, p. 206.

[12] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Commun. ACM*, vol. 18, pp. 453–457, August 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[13] D. Brugali, *Software Engineering for Experimental Robotics*. Springer, 2007.

[14] W. D. Smart, “Is a common middleware for robotics possible?” in *Proc. IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*, 2007. [Online]. Available: http://www.cse.wustl.edu/ wds/?q=papers&display=detail&tag=iros-ws2007

[15] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, “Towards component-based robotics,” in *Proc. IEEE/RSJ Int’l Conf. on Intelligent Robots and Systems*, 2005, pp. 1475–1480.

[16] Baillie, “Urbi: Towards a universal robotic low-level programming language,” in *Proc. IEEE/RSJ Int’l Conf. on Intelligent Robots and Systems*, 2005.

[17] 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.

[18] C. R. Welborn, “Specifying a domain specific language for cooperative robotics,” Ph.D. dissertation, Texas Tech University, 2006.

[19] 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.

[20] 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.

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