Towards a Deep, Domain-specific Modeling Framework for Robot Applications

Colin Atkinson, Ralph Gerbig, Katharina Markert, Mariia Zrianina, Alexander Egurnov, and Fabian Kajzar

University of Mannheim, Germany,
{atkinson, gerbig}@informatik.uni-mannheim.de;
{kmarkert, mzrianin, aegurnov, fkajzar}@mail.uni-mannheim.de

Abstract. In the future, robots will play an increasingly important role in many areas of human society from domestic housekeeping and geriatric care to manufacturing and running businesses. To best exploit these new opportunities, and allow third party developers to create new robot applications in as simple and efficient a manner as possible, new user-friendly approaches for describing desired robot behavior need to be supported. This paper introduces a prototype domain-specific modeling framework designed to support the quick, simple and reliable creation of control software for standard robot platforms. To provide the best mix of general purpose and domain-specific language features the framework leverages the deep modeling paradigm and accommodates the execution phases as well as design phases of a robot application's lifecycle.

Keywords: Deep modeling, ontological classification, linguistic classification, domain-specific languages

1 Introduction

As robots become more ubiquitous and embedded in our environment there is a need to simplify the creation of software systems to control them. Today this is a highly specialized and time-consuming task, involving the laborious handcrafting of new applications using low-level programming techniques. However, as more quasi-standard robot platforms emerge (such as the NAO [2], Turtlebot [19] and Lego Mindstorm [21] platforms on the hardware side and the Robot Operating System (ROS) [17] on the software side), the development of robot applications should become easier and more accessible. This, in turn, should encourage the emergence of communities of "robot app" developers offering robot-controlling software on open marketplaces similar to those for smartphone apps today.

Several important developments in software engineering environments need to take place before this vision can become a reality, however. First, a small number of truly ubiquitous "standard" robot platforms need to emerge, supported by rich software frameworks. Such frameworks need to include a lean, efficient execution platform, a rich library of predefined routines and a clean, general-purpose programming/modeling language for applying them. Second, these general-purpose language features need to be augmented with domain-specific modeling capabilities that allow developers to describe their programs using concepts and notations that fit their application domain. Ideally, these languages should be synergistic. Finally, the information represented in these

languages should seamlessly accommodate all phases of an application's life cycle, from design and implementation to installation and operation. This in turn, requires, information modeling techniques that can seamlessly represent multiple levels of classification.

The modeling approach that offers the most intuitive, flexible and yet stable way of supporting such a software engineering environment is the deep (or multi-level) modeling approach [7]. This has been designed from the ground up to support the uniform and level-agnostic representation of domain concepts at multiple abstraction levels, and makes it possible for them to be visualized in both domain-specific and general purpose notations interchangeably. For the purpose of developing robot applications, therefore, what is needed is a predefined framework of robot-control model elements (i.e types and instances) carefully arranged among the multiple classification levels within a deep modeling environment, each represented by appropriate domain-specific symbols. Each level in such a multi-level framework can be regarded as a language in its own right, and where appropriate we will use this term. However, we prefer to use the term "framework" to refer to the whole multi-level ensemble of models. In this paper, we present an early version of a deep modeling framework for robot applications. Developers wishing to create their own robot applications can take this framework and extend/customize the types and objects within it to their own needs. The term "framework" is therefore used in the sense of previous reusable environments such as the San Francisco Framework [11] etc. However, our framework supports more powerful and flexible extension mechanisms.

The remainder of this paper is structured as follows. In the next section, Section 2, we provide a brief overview of deep modeling and the main concepts that are needed to support it. In Section 3, we then provide an overview of the proposed deep robot modeling framework, and the different levels of classification that it embodies. In particular, we elaborate on the role and nature of each of the four individual ontological classification levels within the framework and discuss the kinds of model elements that they contain. In Section 4, we briefly discuss the main related work and in Section 5 we conclude with some final remarks.

2 Deep Modeling

Deep modeling involves the creation of models spanning multiple classification levels. One of the most well known modeling architectures supporting this approach is the orthogonal classification architecture (OCA) [7] which distinguishes two fundamental types of classification — linguistic classification, defining which construct in the underlying modeling language a model element is an instance of and ontological classification defining which domain concept in the problem domain a model element is an instance of. By arranging these different kinds of classification into two separate, orthogonal dimensions, the OCA manages to provide the flexibility of multiple (i.e. more than two) classification levels whilst retaining the benefits of strict modeling. In contrast, state-of-the-art meta-modeling approaches allow only one pair of class/instance levels to be modeled at a time (e.g. an M_2 meta-model which is then instantiated by an M_1 model).

These are therefore commonly characterized as two-level modeling technologies and generally mix linguistic and ontological classification in one dimension.

One important consequence of multi-level modeling is that elements in the middle levels are usually classes and objects at the same time - that is, they usually have both a type facet and an instance facet. To accommodate this, deep models are usually constructed from so called "clabjects" that have an inherent type/instance duality. To support deep instantiation — the instantiation of model elements across multiple classification levels — each clabject has a non-negative Integer attribute called potency that captures its "typeness". The potency specifies over how many consecutive levels a clabject can be instantiated. Attributes and their values also have a potency. The potency of an attribute (also known as its durability) specifies over how many instantiation steps an attribute can endure (i.e. be passed to instances). On the other hand, the potency of an attribute's value (also known as its mutability) defines over how many levels that value can be changed. The values for all three kinds of potency can be either a non negative integer or "*" representing infinity. When instantiating a clabject, the potency of the clabject and the durability and mutability of its attributes are reduced by one. When instantiating a clabject with "*" potency, the potency of the instance can be "*" again or a non-negative integer. Clabjects with a potency of zero cannot be further instantiated, attributes with a durability of zero are not passed on to instances of the containing clabjects and a mutability of zero rules out any further changes to the value of an attribute.

Figure 1 gives a schematic illustration of how models are represented in the OCA [7]. There are always three linguistic levels, L_2 - L_0, where the top most level, L_2, represents the Pan-level Model (PLM) which is the single, overarching linguistic (meta) model describing the abstract syntax of the deep modeling methodology. The middle level, L_1, contains the domain model content created by users, and L_0, represents the real world representation of the modeled content in the sense of the "Four-Layer Metamodel Hierarchy" in the UML [18]. The levels O_0 - O_2, rendered in the Level-agnostic Modeling Language (LML) [8], are the ontological classification levels which exist within the L_1 linguistic level and are therefore orthogonal to it. This model shows only three ontological levels for space reasons, the maximum number of ontological levels depends on the modeled problem domain and is unlimited. In this figure, linguistic classification is represented by vertically dashed arrows while ontological classification is represented by horizontally dotted lines. However in a real-world model these two representations of classification are usually not used for two reasons. Firstly, the linguistic model is not displayed since the linguistic classification information is already captured by the symbol used to represent model elements. Secondly, representing classification by means of edges clutters diagrams and introduces unnecessary visual complexity. Hence, ontological classification is usually shown using the colon notation as in Figure 1. Deep instantiation is captured by means of the potency value attached to clabjects which in the LML is represented as a superscript to the right of a clabject's name.

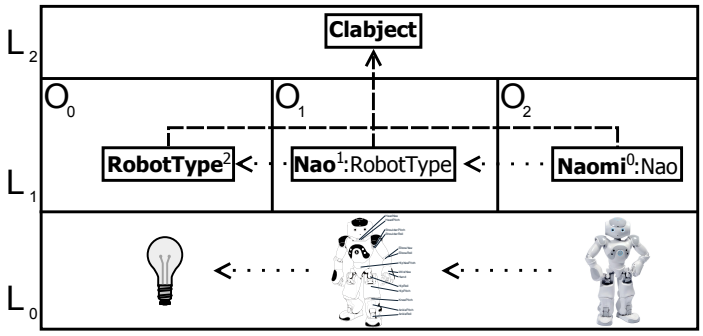

Fig. 1. An example of a deep model.

The example in Figure 1 shows how the clabjects in a robot application would be arranged in the deep robot modeling framework presented in the following sections. On the highest (i.e. most abstract) ontological level 0_0, the concept of a *RobotType* is introduced. Specific robot types, such as the *NAO* robot type from Aldebaran Robotics [2] are modeled as instances of *RobotType* at the next level of abstraction O_1. The clabject *NAO* is thus an ontological instance of *RobotType* and at the same time a type for robot instances in the following levels. The most concrete ontological level in Figure 1, O_2, contains specific robot individuals, such as a robot called *Naomi*, which is an ontological instance of *NAO*. Notice that each clabject's potency, represented as superscript after the clabject's name, is always one less than that of its ontological type resulting in a specific robot at O_2 which cannot be further instantiated since it has a potency of *0*. All model elements are also indicated as being an instance of Clabject which defines their linguistic type. Other linguistic types such as generalization or attribute are also available but are not shown in this small schematic illustration. The bottom linguistic level, L_0, contains the real world entities that are actually represented by the clabjects in L_0. Note that *Naomi* is a physical object, while *NAO* and *RobotType* are conceptual entities (i.e. types) in the domain.

3 Deep Robot Modeling Framework

The overall structure of the proposed Deep Robot Modeling Framework (DRMF) is presented in Figure 2 which essentially shows the L_1 linguistic level of the framework, but rotated anti-clockwise relative to Figure 1 and, thus, represented vertically rather than horizontally. The framework is composed of four ontological levels with the most abstract level O_0, depicted at the top and the most concrete, O_3, depicted at the bottom. The different levels of the model define languages which are used for different purposes. Their purposes are explained in their own dedicated subsections in the following.

The prototype realization of the framework has been implemented using the Melanee [4] deep modeling framework under development at the University of Mannheim. It is therefore based on the linguistic L_2 model of Melanee which is an EMF implementation of the PLM. The PLM is the vertical linguistic level on the left hand, spanning all ontological levels in the center. Similarly, the "real

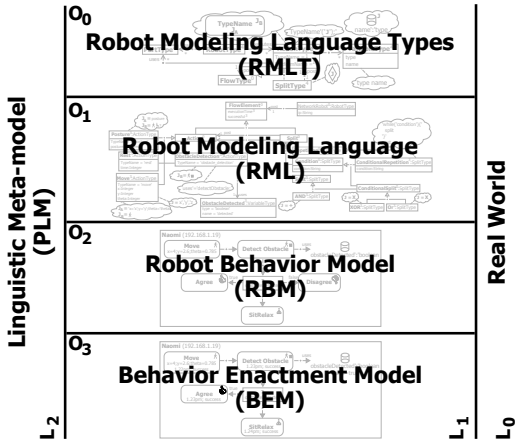

Fig. 2. An overview of the deep robot modeling framework.

world", L_0, containing the objects and concepts in the real world (in this case the robot application) is a vertical linguistic level on the right hand side.

Since the whole framework is based on Melanee, the framework is able to offer some advanced modeling concepts which are only partially supported, if at all, by other comparable modeling infrastructures and environments. The first is the support for symbiotic general-purpose and domain-specific languages. This feature is made possible because Melanee allows domain-specific symbols to be associated with clabjects directly within the ontological levels. The option of rendering clabjects in one or more domain-specific ways is therefore always additional to the option of rendering clabjects in the general purpose LML notation which is Melanee's built in concrete syntax for clabjects. When choosing how a clabject should be rendered, therefore, users are able to switch between all the defined domain-specific symbols or the built-in LML symbol at the click of a button. The Melanee rendering mechanism is fully reflexive, which means that when looking for a symbol to render a clabject, Melanee searches up the hierarchy of supertypes and (ontological) types of the clabject to be rendered, looking for the closest associated symbol. As a last resort, if no domain-specific symbol has been found, the built in LML notation is used. The rendering algorithm also supports concepts of aspect-orient modeling. Join points can be defined in visualizers for which aspects can then be provided in other visualizers. The visualizer search algorithm then merges aspects into join points when working out which symbol to use for a clabjet. The domain-specific modeling language features are used to provide a standard graphical and textual representation at the *Robot Modeling Language Types* level (O_0) which can then be further refined by aspects provided at lower levels of abstraction e.g. the *Robot Modeling Language* (O_1), the *Robot Behavior Model* (O_2) or the *Behavior Enactment Model* (O_3).

The second advanced modeling feature is the uniform and balanced support for textual as well as graphical visualization of clabjects. This is made possible by Melanee's support for full projective editing [5], which means that all visual-

izations, whether textual or graphical are derived by projecting the underlying model content into a particular representational form by selecting a particular set of visualizers. This is a very powerful feature because it means that the same underlying model can be viewed and edited in a graphical way (using graphical visualizers) and in a textual way (using textual visualizers) depending on the skills and goals of the stakeholder concerned. Moreover, each visualization is generated on the fly, when needed, so that changes to the model input through one view are automatically updated in all other open views. The textual visualization of the model content is particularly important since it allows the DRMF to interact with existing text-driven technologies. In general, any textual output can be generated, be it code in a high-level programming language like Java or C++ (as in our implementation), XML, JSON or any general-purpose language (e.g. python, perl, LUA, bash) or specialized scripting language (e.g. Urbi script [9]). By having textual representations of the model, users can apply any kind of algorithm to a model, run it on the robot or load it into other tools.

The third advanced modeling feature supported by Melanee is the ability to model equally and uniformly at all ontological classification levels, with changes at one level immediately impacting all other dependent levels. This makes it possible for modelers to dynamically customize (on-the-fly) the different languages provided by the DRMF to their specific needs. Hence, new types and default renderings can be introduced at the *RMLT* level or new features to model new behaviors can be introduced into the *RML*. An emendation service [6] is provided to help users handle the impact of model changes at any level. This service scans the whole model for model elements which are impacted by a change and suggests automatic amendments to ensure that all the classification relationships valid before the change remain valid.

3.1 O_0 — Robot Modeling Language Types (RMLT)

The *Robot Modeling Language Types* (RMLT) model defines the general concepts needed to create a robot programing language based on a state transition system. More specifically, it defines the types that can make up a robot and general algorithm description concepts such as *ActionType* or *VariableType*.

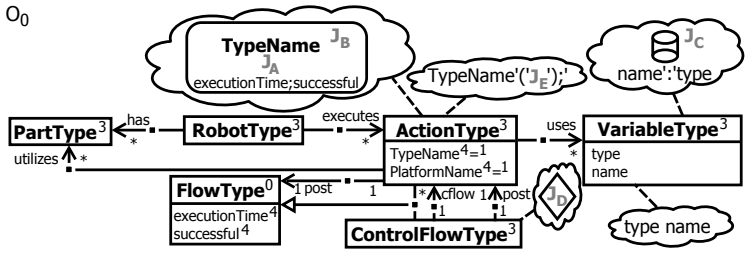

Fig. 3. Level O_0 of the DRMF, the Robot Modeling Language Types.

An excerpt of the RMLT is shown in Figure 3. It provides four basic types: *PartType*, *RobotType*, *VariableType* and *FlowType* which is specialized by the sub-

classes *ActionType* and *ControlFlowType*. The central model element is *RobotType* which is a type for representing a specific kind of robot and its behavior. The left hand side of the *RobotType* provides types for describing its structure (i.e. *PartType*). This is needed as some Robots do not have a static structure but can be changed depending on the task they are required to perform. The right hand side of *RobotType* defines types for the set of actions that a robot can execute (i.e the behavior). The underlying idea is that a program consists of a set of actions and control flow statements. Similar to the parts of a robot, the blocks of the program are attached to a robot. Each *ActionType* can be connected to another *ActionType* facilitating the creation of sequences of action types. Furthermore, an *ActionType* allows instances to use a variable for reading or storing information. *ActionTypes* represent all types of actions that a robot can perform ranging from sensing, waiting for events to actions like moving an arm. In addition to the *ActionType*, a *ControlFlowType* is provided representing the the execution order of actions (e.g. parallel execution and repetition). Default textual and graphical renderings are provided by the RMLT which are represented schematically by the clouds in the example. Points for extending these are offered by join points (represented by the grey *J*s in Figure 3). The RMLT can also be extended with new types by leveraging the full power of the deep modeling approach.

3.2 O_1 — Robot Modeling Language (RML)

The *Robot Modeling Language* (RML), at level O_1, defines the set of actions which can be used to define applications for a robot. This level allows language engineers to define types needed to build robotic applications and to solve particular tasks. The RML can then be used by an end-user to build robotic applications at level O_2. To define a language that can be used by an end user the types of the RMLT need to be instantiated. These instantiations include a robot with such information as connection parameters (e.g. IP attributes) and if necessary the parts that are available for modifying the robot. Additionally the action and flow control elements available in the RML are instantiated from the *FlowType* subclasses provided by the *RMLT*. An executable textual definition and graphical renderings can be defined by a language engineer for the robot specific actions and control elements. For this task the renderings provided by the *RMLT* can be modified by providing aspects or by defining completely new renderings. Using the *RMLT*, different languages can be defined to create applications for different kinds of robots such as humanoid robots, industrial robots and vacuum cleaners etc. The *RML* can be either created for a specific robot or for a family of robots. When defining a language for a family of robots specific implementation types are provided by subclassing more general model elements.

Figure 4 shows a RML defined specially for the NAO robot type. In general, the RML contains a family of types for each kind of robot. However, in Figure 4 we have shown a NAO example model for space reasons. Because the NAO robot's body structure is fixed and cannot be modified the details of the robot and its parts are left out. A *NetworkRobot*, representing the concept of a NAO robot running over a network is instantiated from *RobotType* with an additional String attribute for storing its IP. Actions for the NAO which are instantiated

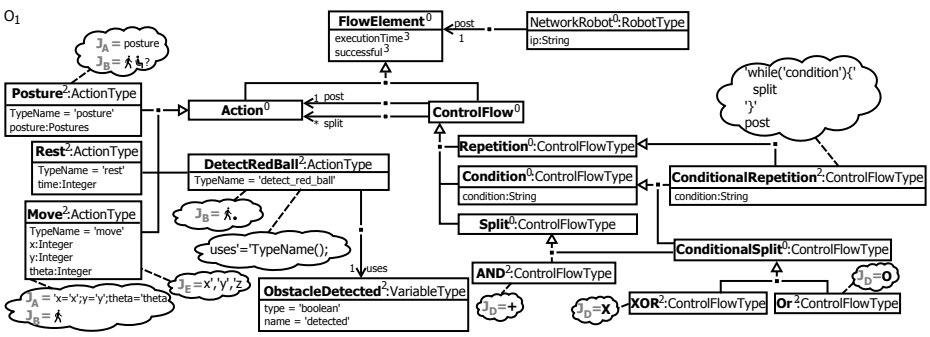

Fig. 4. Example for a O_1 level of the DRMF, the Robot Modeling Language for NAO.

from *ActionType* include default operations provided by the API (e.g. *Move*), custom implementations (e.g. *Posture*) and actions for sensing and reacting on events (e.g. *DetectRedBall*). The commonly known concepts for control flow (e.g. *XOR*, *Repetition*) are instantiated from *ControlFlowType*. The graphical and textual renderings are adapted by providing aspects for join points which is indicated through clouds containing the name of the join point followed by the information provided by the aspect. The defined types can now be used to define applications on the next level.

3.3 O_2 — Robot Behavior Model (RBM)

To model behavior for a robot the *RML* located at O_1 is instantiated at O_2 as shown in the example in Figure 5. The example shows a simple program for a robot called *Naomi*, a NAO robot which is available under the IP address *192.168.1.19*. The program instructs Naomi to first move forward and detect a red ball. If a ball it detected the robot will execute the *Agree* behavior and if not the *Disagree* behavior. The application then instructs the robot to sit down and terminates.

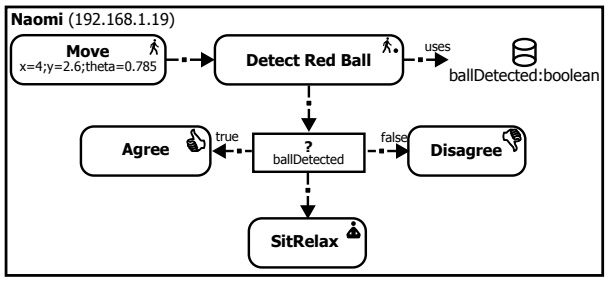

Fig. 5. Level O_2 of the DRMF, the Robot Behavior Model.

To execute the application it is translated into an executable textual format by interpreting the visualizers provided by the *RMLT* and *RML*. If needed these can even be adapted at the *RBM* level. In the prototype realization the application is translated into an internal C++ domain-specific language, compiled and then executed. Other tool chains could also be invoked. The domain-specific language code created for the application in Figure 5 is shown in Listing 1.

```
#include "naoAPI.h"
void NAOProgram::script(){
  move_navigation(4.0, 2.6, 0.785);
  boolean ballDetected = detect_red_ball();
  if (ballDetected)
    agree();
  else
    disagree();
  posture("SitRelax");
}
```

Listing 1. The source code generated from the model displayed in Figure 5.

3.4 O_3 — Behavior Enactment Model (BEM)

Robotic behaviors themselves serve as types for the execution of a robotic behavior. In other words, each behavior can be executed (i.e. instantiated) multiple times, with each instance represented as a separate object. Such an instance of a *RBM* is called an Behavior Enactment Model (*BEM*). The models are usually retrieved from logging information that was created during the execution of a *RBM*. A possible enactment model of the application presented in Figure 5 is shown in Figure 6. The example shows the application that was executed by *Naomi* available under *192.168.1.19* starting with a move at *1.22pm* which was finished with *success*. After the move, the *Detect Red Ball* was switched on at *1.23pm* and finished with *success* resulting in the red ball detection. The robot then made an *Agree* gesture at *1.23pm* before sitting down at *1.24pm*.

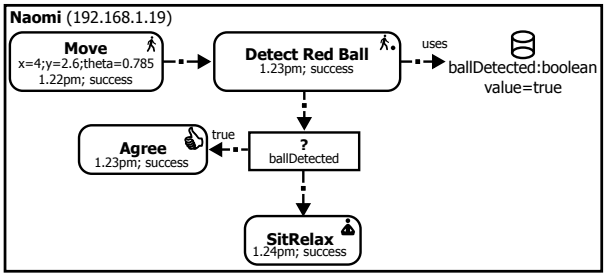

Fig. 6. Level O_3 of the DRMF, the Behavior Enactment Model.

It can be observed that the rendering in Figure 6 uses the whole palette of visualization possibilities defined at the levels above. The RBM in contrast did not use the visualization possibilities for the *executionTime* and the *successful* flag as there were no values for these attributes at the time of application definition.

4 Related Work

In recent years several software frameworks have been developed to provide simple and intuitive ways of writing software applications for quasi-standard robot platforms. This includes academic research (e.g. [10] [12] [20]) as well as industrial products. One of the most well known is Lego Mindstorms Evolution 3, developed especially for the Lego robots which can be built out of the Lego model kits. This is an extremely flexible and powerful system which allows anyone to build a robot using a few standard parts like motors, color sensors, touch sensors, infrared sensors and other Lego elements. These parts only have to be plugged to

the so called brink — "a small computer that controls the motors and sensors" [21]. Afterwards, the user can graphically implement a program by choosing the desired activities from the pallet of available blocks. The software is advertised as having an "easy, intuitive and icon-based programming interface" [14] which gives first-time programmers hands-on access to information technology. Because of this target group, the software only has a limited set of functions and cannot be extended in any way. Evolution 3 only supports the creation of software for Lego robots, and thus cannot be regarded as a general robot modeling framework.

Choregraphe is an environment developed by Aldebaran Robotics, the manufacturer of the NAO humanoid robot, to allow robots to be programmed by graphical applications [3]. It also supports code reuse and debugging capabilities and makes it possible to monitor and control NAO robots manually. The program uses an intuitive drag-and-drop interface in which a program is created using boxes that can be combined into a kind of flow diagram. Aldebaran Robotics provides several tutorials as well as online documentation which simplifies the use of the tool [1]. In summary, although it is easy to use, Choregraphe allows the creation of complex programs. Like Lego Mindstorms Evolution 3, Choregraphe can only be used in combination with the NAO robot and thus cannot be regarded as a general robot modeling framework.

Robotino View 2 is a visual development environment provided by Festo Didactic exclusively for Robotino robots. It supports a slightly different way of visualizing programs than other tools, allowing it to provide some unique features. In particular, Robotino View 2 programs resemble electrical circuit diagrams rather than classical data flow chart. This makes them easier to understand for engineers, but creates a larger learning curve for programmers familiar with traditional langauges. Another unique feature allows users to draw complex lines from several segments. This comes in handy when models grow large and helps minimize intersections. Like Choregraphe, Robotino View 2 allows users to create custom blocks by including C++ code. It also uses two levels of programming, though they are very different to one another. The Block library includes all the blocks needed to create both simple and sophisticated programs, and the simulation environment is freely available from developer's website. Robotino View 2 shares the same limitation as the two previously mentioned frameworks — it is proprietary and can only be used with one kind of robot.

Microsoft Robotics Developer Studio 4 (MRDS4) [16] is another programming environment for building robotics applications. It provides a Visual Programming Language with an intuitive drag-and-drop interface for hobbyists and support for Microsoft Visual Studio for professional developers. MRDS4 has several significant advantages. First, numerous robots such as Lego Mindstorms NXT, Roomba [13] and Reference Platform [15] are supported. Second, a high-fidelity simulation environment is provided by Visual Simulation Environment (VSE), powered by NVIDIA PhysX engine, and the functionality of MRDS4 can be extended by providing additional libraries and services. Third, extensive documentation, samples and tutorials are available "out of the box". The main disadvantages of MRDS4 is the computational overhead resulting from the use of

the simulation environment to control real robots. Another problem is that simulations tend to be overly simplified and do not take into account environment parameters such as surface type and weather.

Although these different languages and platforms are superficially very different, at a high enough level of abstraction they all contain the same basic constructs – predefined types representing the components and actions from which the structure and behavior of individual robots are constructed. The same is true of the Robot Operating System [17] which represents an attempt to define a standard set of component and action types by the Open Source Robotics Foundation. In principle, therefore, they could all be brought together under the umbrella of a single, unified robot modeling framework, where common types and specific types are arranged in inheritance hierarchies in the usual way. The great advantage of using deep modeling technology for such a unified robot modeling framework is that new types can be added, and existing types modified, at any time, on the fly, by simply instantiating the predefined meta types. All the information in the framework is therefore directly manipulable data, but nevertheless can be created and verified using the advantages of a strong typing system.

5 Conclusion

In order to open up the creation of robot applications to a wider range of developers, and encourage the emergence of a community of third party "robot app" developers, it is necessary to offer a robot modeling framework that is efficient, extensible, easy-to-use and able to support the description of applications in a variety of languages. The environment should also support the modeling and visualization of all information relevant to a robot, including dynamic information that is used to control and monitor its operation at run-time. These goals can best be achieved using a deep modeling environment, augmented with support for symbiotic languages, concurrent textual and graphical concrete syntaxes and on-the-fly visualization customization via aspect-orientation. In this paper we have presented a prototype framework, known as the Deep Robot Modeling Framework (DRMF), which supports these capabilities using the Melanee deep modeling environment under development at the University of Mannheim. The current version of the prototype supports a rudimentary implementation of all of these features in the context of the NAO robot platform developed by Alderbaran Robots, although the basic framework is platform independent. Applications developed using the NAO-specific languages are automatically mapped into C++ code that can be loaded onto, and used to drive, individual NAO robots. In the future, we plan to extend the environment to exploit other advanced features of Melenee such as the integrated support for exploratory and constructive modeling.

References

1. Aldebaran: Choregraphe user guide - nao software 1.14.5 documentation. `https://community.aldebaran-robotics.com/doc/1-14/software/choregraphe/index.html` (2014)

2. Aldebaran Robotics: Aldebaran robotics — humanoid robotics & programmable robots. http://www.aldebaran.com (2014)
3. Aldebaran Robotics: Choreographe overview. https://community.aldebaran-robotics.com/doc/1-14/software/choregraphe/choregraphe_overview.html\#choregraphe-overview (2014)
4. Atkinson, C., Gerbig, R.: Melanie: Multi-level modeling and ontology engineering environment. In: Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards. pp. 7:1–7:2. MW '12, ACM, New York, NY, USA (2012)
5. Atkinson, C., Gerbig, R.: Harmonizing textual and graphical visualizations of domain specific models. In: Proceedings of the Second Workshop on Graphical Modeling Language Development. pp. 32–41. GMLD '13, ACM, New York, NY, USA (2013)
6. Atkinson, C., Gerbig, R., Kennel, B.: On-the-fly emendation of multi-level models. In: Vallecillo, A., Tolvanen, J.P., Kindler, E., Strrle, H., Kolovos, D. (eds.) Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 7349, pp. 194–209. Springer Berlin Heidelberg (2012)
7. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. IEEE Trans. Softw. Eng. 35(6) (2009)
8. Atkinson, C., Kennel, B., Goß, B.: The level-agnostic modeling language. In: Malloy, B., Staab, S., Brand, M. (eds.) Software Language Engineering, Lecture Notes in Computer Science, vol. 6563. Springer Berlin Heidelberg (2011)
9. Baillie, J.C.: Urbi: Towards a universal robotic low-level programming language. In: Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on. pp. 820–825 (2005)
10. Banyasad, O., Cox, P.T.: Visual programming of subsumption-based reactive behaviour. In: Technical Report CS-2008-03. pp. 365–380. Dalhousie University (2008)
11. Bohrer, K., Johnson, V., Nilsson, A., Rubin, B.: The san francisco project: An object-oriented framework approach to building business applications. In: Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International. pp. 416–424 (Aug 1997)
12. Cox, P., Smedley, T.: Visual programming for robot control. In: Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on. pp. 217–224 (1998)
13. Kurt, T.E.: Hacking Roomba. Wiley (2006)
14. LEGO: Website, available online at http://shop.lego.com/en-US/LEGO-MINDSTORMS-EV3-31313; visited on April 13th 2014.
15. Microsoft Robotics Group: Robotics Developer Studio: Reference Platform Design V1.0. Microsoft Robotics Group (2012)
16. Morgen, S.: Programming Microsoft Robotics Studio. Microsoft Press, 1st edn. (2008)
17. O'Kane, J.M.: A Gentle Introduction to ROS. Independently published (2013)
18. OMG: Uml infrastructure 2.4.1. http://www.omg.org/spec/UML/2.4.1 (2011)
19. Open Source Robotics Foundation: Turtlebot. http://www.turtlebot.com/ (2014)
20. Simpson, J., Jacobsen, C.L.: Visual process-oriented programming for robotics. In: Communicating Process Architectures 2008, volume 66 of Concurrent Systems Engineering. pp. 365–380. IOS Press (2008)
21. Valk, L.: The Lego Mindstroms NXT 2.0 Discovery Book A Beginners Guide to Building and Programming Robots. William Pollock (2010)