Model-Driven Robot Software Engineering (MORSE) 2014 Preface

January 21, 2015

Software engineering is the discipline of creating software with high quality and good reusability. Since several years, more and more standard platforms for service robots have appeared, and these platforms demand for high-quality, versatile, and reusable software. Operating systems, such as Embedded Linux, and distribution technologies, such as Web Services, have successfully been ported to these standard robotic platforms enabling the transfer of a large amount of the standard software engineering body of knowledge to robots. In particular, there is a need for Model-Driven Software Development (MDSD) for robots, because models can capture certain quality aspects of robotic software better than code, enabling simpler testing, easier verification, and finally, certification of safety-critical applications. Though code written in a classical programming language can often not be verified for relevant features, models can, because they abstract from unnecessary detail. And this highlights their potential for robotic software engineering: If robot software is ever going to be certified on the large scale, it must consist of models.

Therefore, the objective of the first international workshop on "Model-Driven Robot Software Engineering (MORSE) 2014" has been to assemble researchers from both fields, Model-Driven Software Development and Robotics, to discuss the interaction of their areas, to investigate fruitful research directions, and to identify challenges for further research. The call for papers mentioned, among others, the following research topics arising in the overlap of Software Engineering and Robotics:

- Model-Driven Software Development for Robotic Systems
- Robotic Software Platforms: Models, Processes and Tools
- Software and App Reuse for Robotics, Robot Ecosystems
- Model-Driven Quality Assurance of Robotic Systems

The workshop ran at the STAF multi-conference in York (GB), on July 21, 2014. STAF already hosts two conferences for Model-Driven Software Development, International Conference on Model Transformation (ICMT) and International Conference on Graph Transformation (ICGT). In consequence, the workshop welcomed 20 participants, indicating a broad interest in the topic.

On its call for papers, MORSE received 9 submissions. Each paper was assigned to three reviewers who read and corrected them in two reviewing rounds, one before and one after the workshop. The idea was to give hints to the authors to achieve a high-quality publication for a post-proceedings, and not to filter out papers, because the community is young and people need to learn of each other. We thank the reviewers for their effort to investigate the papers several times and hope that this volume is interesting enough to justify a repetition of the workshop at STAF 2015 in L'Aquila/Italy.

Uwe Aßmann, Technische Universität Dresden, Germany
Gerd Wagner, Brandenburgische Technische Universität
Cottbus-Senftenberg, Germany
PC chairs of MORSE 2014

Reviewer List

Table of Contents for PDF-Volume

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.

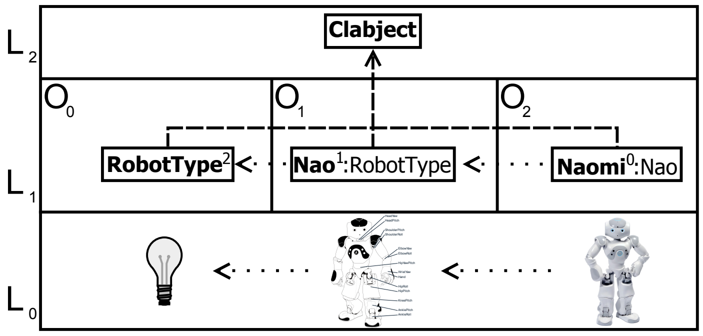


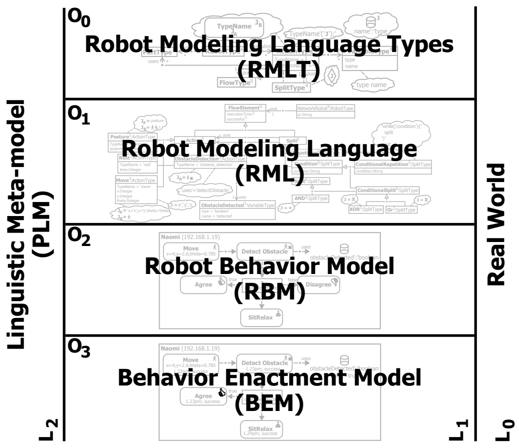
Fig. 1. An example of a deep model.

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



O_0
Robot Modeling Language Types (RMLT)
O_1
Robot Modeling Language (RML)
O_2
Robot Behavior Model (RBM)
O_3
Behavior Enactment Model (BEM)
Linguistic Meta-model (PLM)
Real World
L_2
L_1
L_0

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 visualizations, 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*.

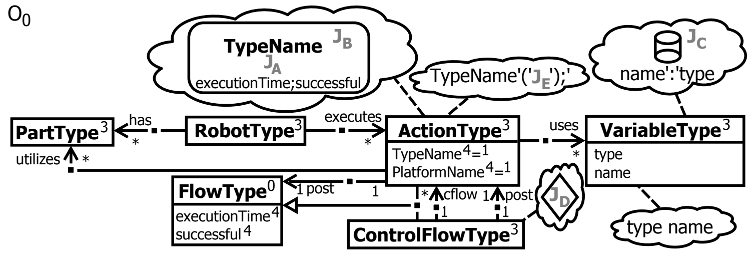


Fig. 3. Level O_0 of the DRMF, the Robot Modeling Language Types.

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 subclasses *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

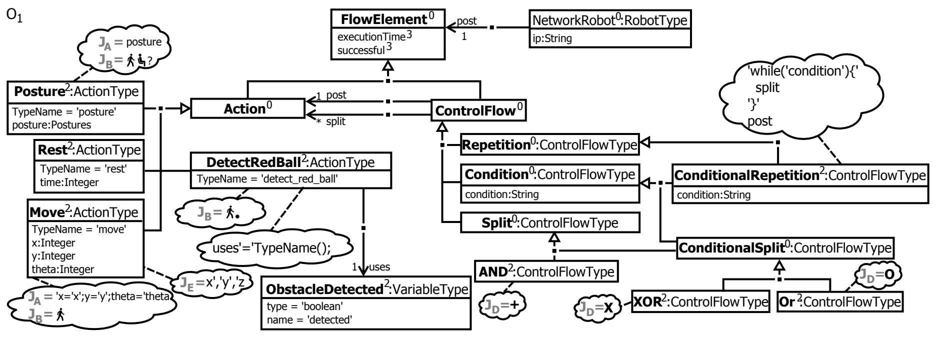


Fig. 4. Example for a O_1 level of the DRMF, the Robot Modeling Language for NAO.

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 $\mathbf{O_2}$ — 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.

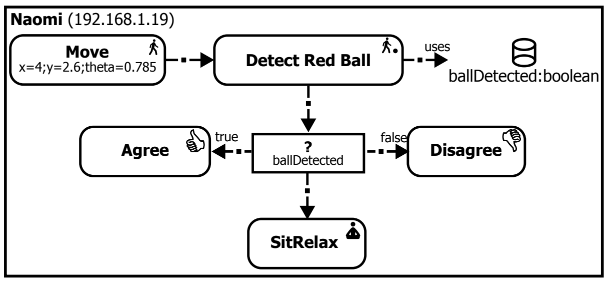


Fig. 5. Level O_2 of the DRMF, the Robot Behavior Model.

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*.

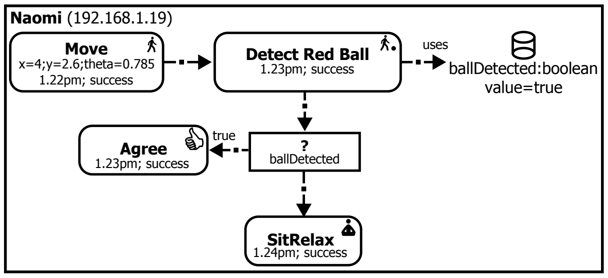


Fig. 6. Level O_3 of the DRMF, the Behavior Enactment Model.

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)

A Family of Domain-Specific Languages for Specifying Civilian Missions of Multi-Robot Systems

Davide Di Ruscio[1], Ivano Malavolta[2], and Patrizio Pelliccione[3]

[1]Department of Information Engineering Computer Science and Mathematics
University of L'Aquila (Italy)
[2]Gran Sasso Science Institute, L'Aquila (Italy)
[3]Department of Computer Science and Engineering
Chalmers University of Technology |University of Gothenburg (Sweden)
davide.diruscio@univaq.it,ivano.malavolta@gssi.infn.it,patrizio.pelliccione@gu.se

Abstract. The next future will be pervaded by robots performing a variety of tasks (e.g., environmental monitoring, patrolling large public areas for security assurance). So far, researchers and practitioners are mainly focusing on hardware/software solutions for specialized and complex tasks; however, despite the accuracy and the advanced capabilities of current solutions, this trend leads to task-specific solutions, difficult to be reused and combined.
In this paper we propose a family of domain-specific languages for specifying missions of multi-robot systems by means of *models* that are (i) independent from the technologies, (ii) ready to be analysed, simulated, and executed, (iii) extensible to new application areas, and (iv) closer to the problem domain, thus democratizing the use of robots to non-technical operators. We show the applicability of the proposed family of languages in a real project in the domain of autonomous unmanned aerial vehicles.

1 Introduction

The next future will be pervaded by robots that, moving underwater, on terrain, or flying, will simplify everyday tasks or will open a myriad of new opportunities. Multi-robot systems will behave as a team, in which each single robot accomplishes a well defined task towards the accomplishment of a global mission. On one side a multi-robot team can accomplish a mission more quickly than a single robot, and on the other side a multi-robot team can accomplish missions requiring particular capabilities that might be impossible or impractical to find on a single robot: a team can make effective use of specialists designed for a single purpose, e.g., scouting an area or picking up objects.

The specification of a mission is difficult when considering a single robot since many details should be taken into account, and it might require technical expertise about the dynamics and the characteristics of the used robot. It becomes even more complex when dealing with missions involving multi-robots. Then, it emerges the need of software engineering approaches and methodologies especially tailored to develop and maintain multi-robot systems.

Model-driven Engineering (MDE) [1] is a promising research field for simplifying the design, implementation and execution of software systems for the robotics platforms of the future. In MDE, we can notice a shift from third generation programming language code to models expressed in domain-specific modeling languages (DSMLs). In this context, MDE enables the development of multi-robot systems by means of models defined with concepts that are much less bound to the underlying technology and are closer to the problem domain. This makes the models easier to specify, understand, and maintain, helping the understanding of complex problems and their potential solutions through abstractions [2].

In this paper we present a family of domain-specific languages for specifying civilian missions of multi-robot systems. The proposed languages are organized in different layers going from languages conceived for the end user, namely those to describe missions and the environmental context, intermediate language describing the detailed behaviour of each robot (hidden to the user), and the robot language containing the hardware and low-level specification of each type of robot within the team. In order to show the applicability of the proposed languages in practice, we instantiate it to the domain of civilian missions of autonomous quadrotors. The resulting platform is called FLYAQ [3] and it provides to on.site operators a graphical interface enabling the specification of missions at a high-level of abstraction.

Roadmap of the paper. The remainder of the paper is structured as follows: a description of civilian missions is provided in Section 2. The architecture of the proposed family of languages is presented in Section 3, while their instantiation and implementation to manage swarms of autonomous quadrotors is provided in Section 4. Section 5 discusses the related work, whereas Section 6 concludes the paper and outlines some perspective work.

2 Civilian missions

Several civilian missions have been discussed in the literature. Skrzypietz [4] subdivides civilian missions for Unmanned Aircraft Systems (UAS) in six categories:

▷ *Scientific Research*, such as atmospheric, geological research, forestry.

▷ *Disaster Prevention and Management*, like damage assessment after earthquakes, searching for survivors after airplane accidents and disasters.

▷ *Homeland Security*, such as coastal surveillance, securing large public events.

▷ *Protection of Critical Infrastructure*, such as monitoring oil and gas pipelines, protecting maritime transportation from piracy, observing traffic flows.

▷ *Communications*, like broadband communication, telecommunication relays.

▷ *Environmental Protection*, such as pollution emission, protection of water resources.

According to Washington Post[1], venture investors in the United States poured $40.9 million into drone-related start-ups in the first nine months of 2013, more than double the amount for all of 2012. Moreover, according to the "Unmanned Aerial Vehicle (UAV) Market (2013 - 2018)" market research, the total global UAV Market (2013-2018) is expected to reach $8,351.1 million by 2018. [2]. This is justified by the number of advantages that the use of these devices brings: (i) *costs*: civilian missions typically requires high costs for personnel which have to be carried on site, and to the communication overhead required for synchronization purposes of the teams; (ii) *safety*: on-site personnel may be exposed to significant risks (e.g., in case of fire, earthquake, and

[1] http://wapo.st/1bJueLH

[2] http://www.marketsandmarkets.com/Market-Reports/unmanned-aerial-vehicles-uav-market-662.html

flood); (iii) *timing and endurance*: monitoring activities are very time consuming. Also, the activities are stopped during the night, slowing the execution of the mission.

3 The Family of Languages

The family of domain-specific modeling languages that we propose supports the specification of missions and their actual execution by means of swarms of robots. The developed languages stack is shown in Fig. 1.

MML, namely the *Monitoring Mission Language*, is the language especially conceived for domain experts. The MML language is composed of two distinct layers, that are: the mission layer and the context layer. The former enables the specification of civilian missions without referring to specific aspects like the technical characteristics of the robots that will be used to execute the missions; missions are specified as sequences of tasks, suitably linked together via task dependencies, forks and joins. The specification of a mission is complemented by the definition of the context in which the mission will be realized. Elements of the context can be modeled by means of the *Mission Context Language*.

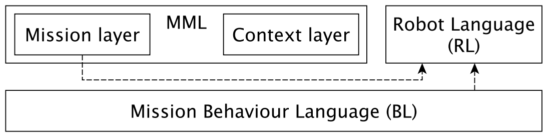


Fig. 1. The family of domain-specific modeling languages

The type and configuration of the robots that will be in charge of realizing the specified mission are described through the *Robot Language* (RL). The *Behaviour Language* (BL) is a language that is hidden to domain experts. It contains a specification of the atomic movements and actions of the robots being considered in the mission. As shown in Fig. 1, both the monitoring mission language and the mission behaviour language have a reference to the robot language (currently those references are implicitly established by the name of the referenced robot). This is needed because: (i) the mission layer of the monitoring mission language contains a list of all the robots of the swarm, and the type of each of them must be specified, and (ii) the mission behaviour language contains all the low-level movements and actions of the robots, whose type must be known in the model in order to correctly instruct the robots at run-time.

It is important to note two important aspects of the family of languages. Firstly, the two layers of the MML language are kept separated in order to allow mission operator to reuse already existing context models across missions and different organizations. Also, the context layer puts restrictions on the mission layer since they share the same location, thus enabling for a straightforward (automatic) composition of the two. Secondly, the family of languages has been designed to support the automatic generation of BL models from MML models; it enables to (i) obtain BL models which are inherently consistent to their corresponding MML models, and (ii) to mask the complexity of low-level details about the used robots (and their actions) to on-site operators. In light of this, the control code for the robots depends only on the constructs of the BL language, and thus can be either automatically generated or directly executed by interpreting BL models at run-time.

Three are the stakeholders of the proposed family of languages, namely:

1. Operator: the in-the-field stakeholder specifying the mission as an MML model; examples of operators include fire fighters, policemen, etc.;
2. Robot Engineer: models a specific kind of robot via an RL model, together with a corresponding controller for instructing the robot according to the basic operations supported by the BL language;
3. Platform Extender: domain and MDE expert who extends the MML mission layer with new kinds of tasks supporting a specific application domain (e.g., agricultural missions, security-oriented missions, smart grid monitoring missions, etc.); those extensions are performed once for each application domain being considered, and can be reused across missions and organizations.

Issues related to mission correctness, e.g., safety and security, are fundamental for civilian missions for multi-robot systems. In this context, the proposed languages have been designed to be generic enough for describing this kind of missions from an high-level point of view, and thus mission correctness is not part of the languages themselves. In any case, the proposed languages provide the right level of abstraction for allowing analysis tools to be executed on the models for proving, for example, safety and security properties. We believe that the Behaviour Language is the best candidate for this kind of analyses, since it can be easily transformed into a corresponding state machine, a process algebra, Petri net, etc., thus allowing engineers to reuse already existing analysis tools.

The remainder of this section will describe each of these languages. Specifically, for each of them we present the corresponding metamodel i.e., the abstract syntax of the language. For what concerns their concrete syntax, the MML language can have a graphical syntax like, e.g., an overlay on a geographical map representing the various tasks, dependencies and contextual elements (see Section 4.2 for a concrete example). Differently, the RL and BL languages are represented by using an XML-based representation since they will be created and managed by domain experts or even by other software components.

3.1 Monitoring Mission Language (MML)

Mission Layer - The mission layer of the MML language has been conceived by analyzing the concepts that are involved when specifying monitoring missions. First of all, each robot has its own *home* position represented by means of the corresponding $\mathrm{Coordinate}$ element (see Fig. 2). Then, a monitoring mission consists of a number of dependent Tasks to be executed by a Swarm of Robots.

The MML mission layer contains three abstract task metaclasses, each of them focusing on a specific kind of geometric entity being considered. More specifically, $\mathrm{PointTask}$ represents tasks that refer to a specific point of the environment. To this end the *point* reference represents the coordinate of the considered point. $\mathrm{LineTask}$ represents tasks that refer to a set of points forming a polyline in the environment. Consequently, a line task consists of the *initialPosition* that the considered robot will have at the beginning ot the task, and a set of *points* consisting of the points of the polyline. Also, $\mathrm{PolygonTask}$ represents tasks that refer to specific areas. A polygon task consists of the *initialPosition* reference, and the *shell* referecing the points representing the border of the area that will be involved during the execution of the task.

The ControlTask is an abstract metaclass that represents the synchronization tasks of the mission. In particular, each task of the considered mission can be executed by one or more robots and can be performed in sequence (see the metaclass Join) or in parallel (see the metaclass Fork) to other tasks of the modeled mission.

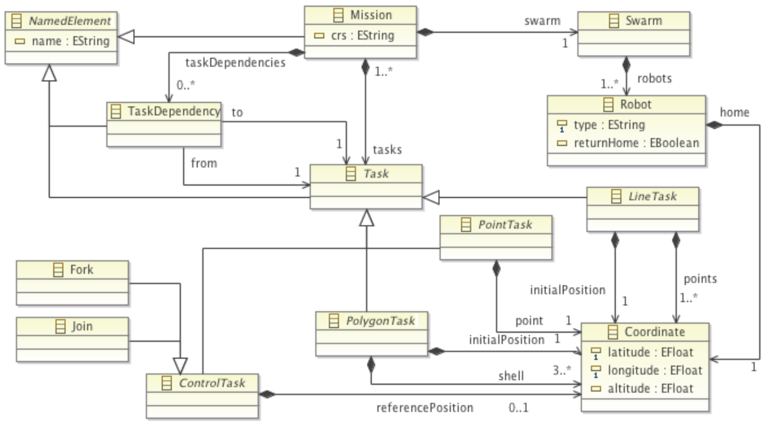


Fig. 2. The Monitoring Mission Language

The mission layer (and so its corresponding metamodel) is defined to be extensible. This means that it specifies only general tasks that need to be specialized according to specific needs, to the actual civilian mission (see Sect. 2 for a description of these missions), and to the robots that will be used. The extensibility of MML allows operators to achieve versatility and strong adherence to the environmental monitoring missions domain. More specifically, MML can be extended with additional constructs that are specifically tailored to the considered domain. For example, if operators are interested to monitoring solar panel installations in a rural environment, MML might be extended with the concept of solar panel groups, thermal image acquisition tasks, and solar panel damage discovery and notification tasks.

Context Layer - As said in the previous section, the specification of monitoring missions includes also the description of the context where they will be executed. By referring to [5], this modeling layer concerns spatial context and situational context. Indeed it represents those portions of geographical areas that have some relevant property, and those elements which can influence the execution of the mission, but that are not part of the mission itself.

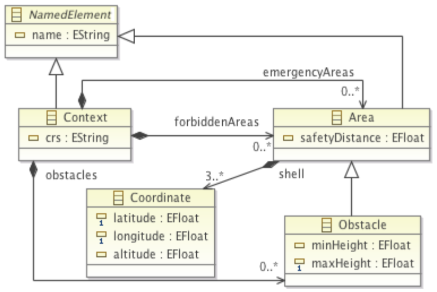


Fig. 3. The Mission Context Language

Fig. 3 shows the metamodel of the context layer of MML. In particular a given monitoring mission will be executed in a Context consisting of a number of *obstacles* and *forbiddedAreas*. Such information will play a key role in order to properly deduce the movements that the robots have to perform in order to execute the missions specified in MML and to satisfy the environmental constraints specified by means of context models.

3.2 Robot Language (RL)

RL has been conceived to enable the specification of the technical characteristics of each type of robot involved in the missions (see Fig. 4). Clearly, Robot is the central concept of the language. The characteristics that the language permits to specify are the following:

- *onBoardObstacleAvoidance*: it permits to specify if the considered robot is endowed with mechanisms able to autonomously avoid obstacles;
- *minVoltage*/*maxVoltage*: they are used to specify the minimal/maximum voltage required/supported by the robot to properly work;
- *maxPowerConsumption:* it is used to specify the maximum power consumption expressed in Watt of the robot being modeled;
- *gps*, *accelerometer*, *magnetometer* and *barometer*: they are boolean attributes used to specify the available on-board sensors of the robot being modeled;
- *communicationRange*: it is used to specify the maximum range expressed in meters of the supported radio control;
- *dataRate*: it permits to specify the data transmission rate expressed in Kbps between the robot and the control station;

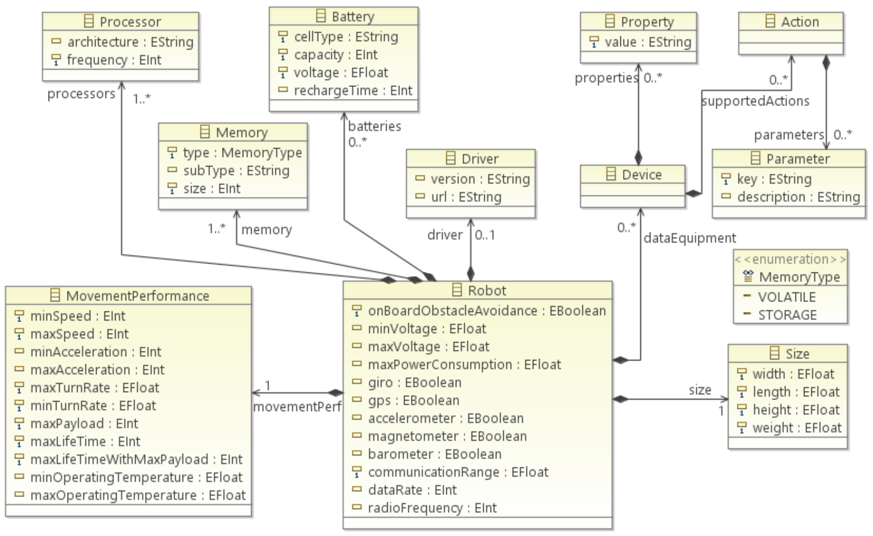


Fig. 4. The Robot Language

- *radioFrequency*: it permits to specify the radio frequency expressed in MHz used by the robot to communicate with the control station.

Furthermore, the Size metaclass is used to specify the size of the robot by means of the attributes such as its *width*, *length*, *height*, and *weight*. The Processor metaclass permits to specify the hardware *architecture* and *frequency* of the processors owned by the robot being modeled. The concept of Memory describes the memory of the considered robot in terms of its type (i.e., if the memory is volatile or permanent), sub-type (e.g., DDR2, DDR3, SSD), and size in kilobytes. Additional devices owned by the robot to gather data (e.g., camera, thermal sensors) and to perform actions (e.g, lights, leds, mechanical actuators, sound emitters) are specified via the Device metaclass. MovementPerformance permits to specify the movement characteristics of the robot (i.e., minimum and maximum speed, acceleration, turn angle). Additionally, it permits to specify the maximum weight of the payload that the robot can bring and consequently also the maximum life time of the robot while bringing a pay load. The minimum and maximum working temperature of the robot can be also specified. Finally, the *Driver* metaclass refers to the software driver, which is required to interact with the robot.

It is important to note that not every robot is specified in RL, but every *type* of robot. This makes RL models reusable and shared across missions, projects, and organizations.

3.3 Behaviour Language (BL)

BL permits to specify atomic movements of each robot in order to perform the missions specified by means of MML specifications. As shown in Fig. 5 a BL model specifies the behaviour of all the robots which will perform the mission and for each of them all the *movements* to be performed are singularly defined. According to Fig. 5 the atomic movements that a robot can perform are the following:

- Start: it represents the first movement used to begin any sequence of movements;
- Stop: it represents the final movement used to end any sequence of movements;
- HeadTo: it represents a rotation in order to head towards the specified *direction*;
- Pause: it permits to specify pauses of robots during their movements;
- Circle: it permits to specify circular movements of robots around a specific point;
- GoTo: it represents the movement towards a given *targetPosition*.

Before executing one move after the end of another one it is possible to specify a transition (see MoveTransition). In particular, if the *fluid* attribute is specified as *true* than the robot will execute the movements seamlessly without any interruption. Alternatively, it is possible to specify a pause (see the metaclass Slot) between two subsequent moves. Additionally, before and after each movement, the robot can perform a number of Actions that can be distinguished as follows:

- DeviceAction: it permits to specify control actions of the robot. To this end the name of the action and the parameters to be used are specified by means of the features *actionName* and *parameters*, respectively;
- CommunicationAction is used to specify the communication among the robots involved in the execution of the considered mission. In this respect, the possible actions that can be performed are the following: (i) CheckNotification: it is used

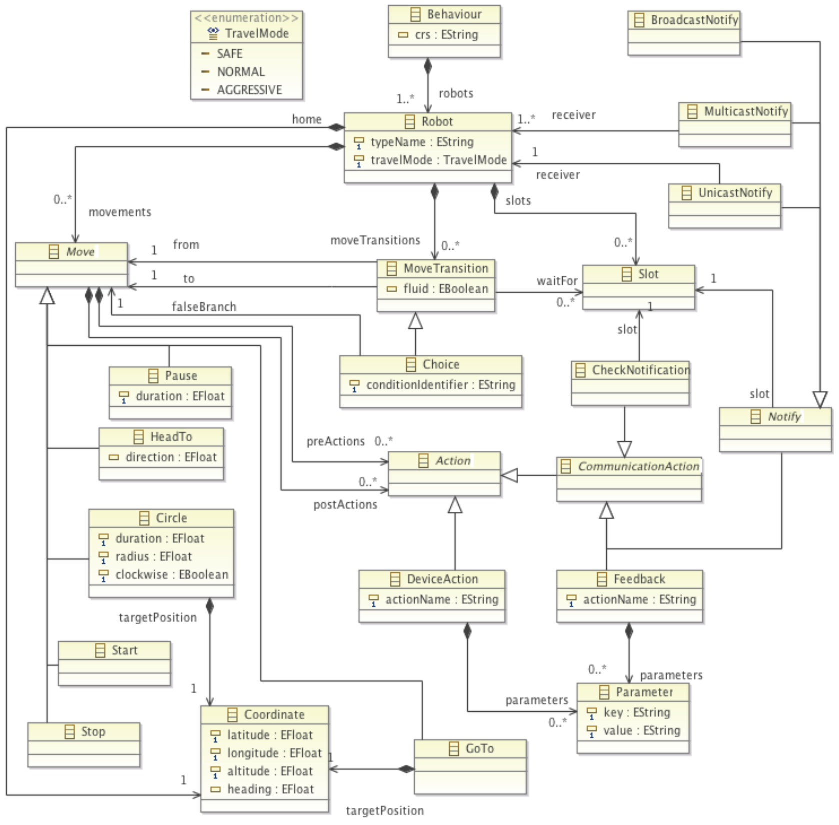
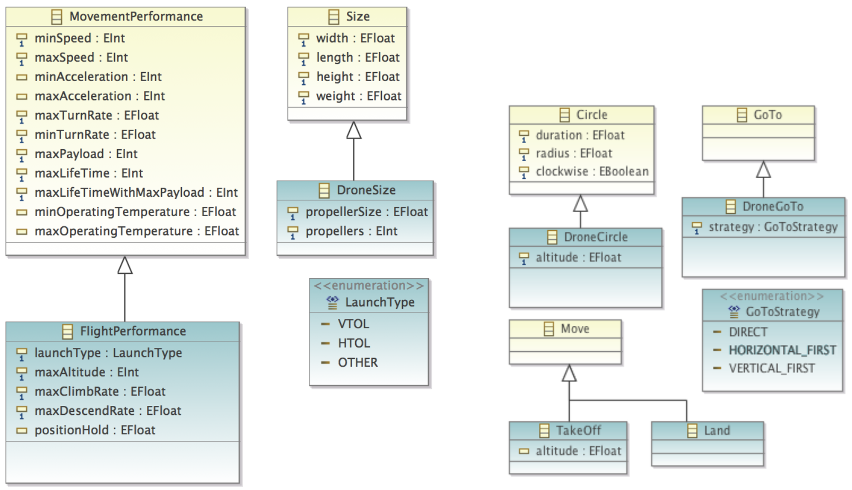


Fig. 5. The Behaviour Language

to manage the reception of notifications from other robots; (ii) Feedback represents a feedback message that the robots can send back to the control station; (iii) Notify is a superclass representing all the possible notification actions that can be distinguished as UnicastNotify, MulticastNotify, and BroadcastNotify.

4 Leveraging the DSL family for autonomous quadrotors

Currently, in collaboration with Telecom Italia we are working on an open-source platform for the specification and the execution of environmental monitoring mission. The platform is called FLYAQ [3] and it allows non-technical operators to straightforwardly define monitoring missions of swarms of flying drones at a high level of abstraction, thus masking the complexity of the low-level and flight dynamics-related information of the drones. More specifically, we employ quadrotors, that are a special kind of unmanned aerial vehicle that takes the form of a multirotor helicopter that is lifted and propelled by four rotors [6]. In this section we present the instantiation of the languages proposed in Section 3 to support the specification of swarms of autonomous quadrotors.



(a) Extension of the robot language (b) Extension of the behaviour language

Fig. 6. Extension of the languages of the DSL family for representing missions of quadrotors

4.1 Extension of the DSL family

In this section we describe how the languages of our DSL family have been extended to support the domain of environmental monitoring missions for autonomous quadrotors. In light of this we evaluated each language of the DSL family, we analysed it in terms of its expressivity with respect to the specific domain in order to check if language concepts fit well with the domain. Interestingly, we did not need to adapt the **Mission** language of the DSL family since it is still a good fit for the new application domain without any extension. Indeed, from an abstract point of view a swarm of autonomous quadrotors can be considered as a swarm of robots performing some task for fulfilling the goal of a global mission. Task may still refer to polygons, polylines, and points within a given mission environment, and tasks may have dependencies and controlled by fork and joins. Even the **Context** language has not been extended since its concepts are still satisfactory in the current state of the FLYAQ project. Indeed, when reasoning about the context of a mission performed by quadrotors the main issues are about: the presence of obstacles (represented by the Obstacle metaclass in the context language metamodel), emergency landing areas (represented by the *emergencyAreas* reference), and no-fly zones where no quadrotor can fly over (represented by the *forbiddenAreas* reference). For what concerns the **Robot** language, we needed to extend it with additional concepts, specific to the nature of the managed robots (i.e., flying quadrotors). Fig. 6(a) shows a fragment of the robot language metamodel focussing on the metaclasses that have been added, they are:

- DroneSize extends the *Size* metaclass of the robot modelling language with two attributes for representing (i) the number of propellers of the drone, and (ii) the size of the propellers of the drone in millimeters. This two additional attributes are necessary since there is a great variability of flying drones with respect to the number and size of their propellers [6], and flying drones behave very differently depending on those two properties. For example, the popular AscTec Firefly[3] drone has six propellers and, by citing its official data sheet, *the redundant propulsion system enables a controlled flight even with only 5 functioning motors and actively compensates for failure*; it is difficult to imagine a drone with only four propellers providing this peculiar safety function.

– FlightPerformance extends the MovementPerformance metaclass of the mission behaviour language with a set of additional attributes:
  - *launchType* represents the information about how the flying drone can be launched. The value of this attribute can be one among Vertical Take-Off and Landing (*VTOL*), Horizontal Take-Off and Landing (*HTOL*), or any other (*OTHER*);
  - *maxAltitude* represents the maximum altitude that the drone can reach when performing a mission;
  - *maxClimbRate* represents the maximum rate of change in altitude of the drone (in metre per second);
  - *maxDescendRate* represents the maximum rate of change that the drone supports when it is descending (in metre per second);
  - *positionHold* represents the maximum wind speed (in metre per second) supported by the drone to maintain a given geographical location.

The attributes added to the MovementPerformance metaclass are specific to the aerial vehicles domain, and have been necessary for representing specific concepts related to this domain (e.g., *maxAltitude* and *launchType*).

When considering the **Behaviour** language of our DSL family we needed to slightly extend it with some additional concepts, they are shown in Fig. 6(b). Basically, we needed to extend the Circle metaclass for specifying the altitude at which the quadrotor must perform the circle movement. Moreover, we extended the GoTo metaclass for specifying the kind of trajectory that the quadrotor must following when it needs to reach a certain point. The available trajectory kinds represent how the quadrotor can reach a given point in the 3D space, they are three: (i) *DIRECT*, the quadrotor follows a straight line between its current position and the target point; (ii) *HORIZONTAL_FIRST*, firstly the quadrotor moves horizontally until it goes below the target point, and then it adjusts its altitude so that it reaches the target point; (iii) *VERTICAL_FIRST*, firstly the quadrotor moves vertically until it reaches the same altitude of the target point, and then it moves horizontally until it reaches the target. We added the above mentioned concepts to the Behaviour language in order to provide better flexibility when reasoning about the movements of the quadrotors in the environment. For example, the GoTo movement could not be extended at all (e.g., all the quadrotors always move directly to the target point), however we argue that this solution could have been too restrictive for performing real missions in practice.

As a matter of fact, we have been actually quite surprised when noticing that we did not have to extend the Mission and Context modeling languages. This result is positive since those two modeling languages have proven to be expressive enough for the needs

[3] http://www.asctec.de/uav-applications/research/products/asctec-firefly/

of a concrete project. However, when reasoning at a lower level of abstraction some kind of adaptation is needed (e.g., for specifying the propeller size of a quadrotor); still, the extensions performed in the Robot and Behaviour languages are not massive and, most interestingly, they did not disrupted the semantics of the languages being extended.

4.2 Implementation of the extension of the DSL family

In this section we provide a description on how we developed the FLYAQ platform by taking advantage of the languages presented in the previous section. Figure 7 gives an overview of the FLYAQ platform. On-site operators design the mission, store, and monitor the status of ongoing missions via a standard web browser connected with the platform through a secure HTTP connection. This design decision enables operators to (re-)use any kind of device, such as tablets, laptops, etc., which are capable to run standard web browsers. Quadrotors are instructed and controlled by the platform via MAVLink communication so that radio modems can retain control up to eight miles.

More in details, the FLYAQ platform offers a web-based **graphical interface** to specify missions in the ground station at a high-level of abstraction and integrated with Open Street Map. As a matter of fact the web interface of FLYAQ is a domain-specific editor for both the mission (see Section 3.1 and context (see Section 3.1) layers of the the MML language. The graphical interface of FLYAQ is implemented using HTML5, JavaScript, CSS3, and web sockets for real-time communication with the quadrotors (i.e., for getting the telemetry feedback from each quadrotor in the field).

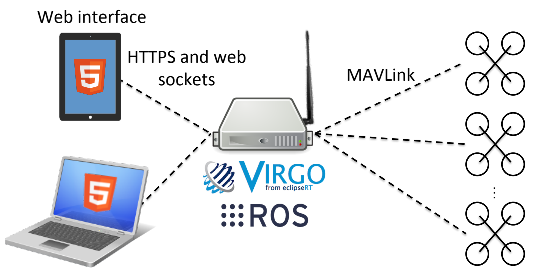


Fig. 7. Overview of the FLYAQ platform

The FLYAQ platform has an internal **engine** that leverages *model transformations* and *formal reasoning* to automatically transform a mission specified using the Mission modeling language into low-level steps conforming to the Behaviour modeling language (see Section 3.3); the transformation step takes also into consideration a model of the context and the models of the drones that will concretely execute the mission in the field.

The FLYAQ internal engine is implemented using Eclipse Virgo[4], the Eclipse Modeling Framework (EMF[5]), Java, and exposes a Rest API to the FLYAQ web interface. So far, the extension of the DSL family languages for adding new concepts related to the UAV domain has been realized manually, i.e., by manually extending the base metamodels of the DSL family in order to obtain the extended metamodels presented in Section 4.1. As a future work we are planning to leverage a more systematic language extension process, with properties such as language independence, possibility to compose and decompose the involved

[4] http://www.eclipse.org/virgo
[5] http://www.eclipse.org/modeling/emf

languages, some level of automation, and so on. The authors have already worked on the topic [7], and the integration of a systematic mechanism for languages extension is currently being evaluated.

Finally, a **layer of controllers** abstracts the types of the specific quadrotors to all the other components of the platform; this is fundamental because it makes the platform totally agnostic of the quadrotors used for executing the mission (abstraction is one of the strongest points of using MDE techniques). The layer managing the controllers is implemented using Java and ROS [8] and its extension rosbridge[6], a middleware communication framework specifically tailored for real-time communication with robots. Each controller can be implemented by using any kind of programming language (thanks to the ROS middleware communication middleware).

5 Related work

Over the last years many research groups have been working on the adoption of model-driven engineering for developing complex robotic systems [9–11]. The advantages of using models for developing this kind of systems are manifold, e.g., higher level abstractions for behaviour descriptions, possibility to apply tools for verifying properties, such as safety, and for generating implementation code.

Typically, the adoption of model-based approaches for developing robot software systems has focused on control or mechanical design aspects. In [9] the authors go further by proposing the adoption of models to manage the complete development of robotic software systems. Similarly, in [10] the authors propose an approach that uses models both at design- and run-time to support robots during their decision making process. In [11] the author proposes an Eclipse based environment for the development of robot control systems, including generation of application code skeleton. In [12] the authors propose a rule-based language for specifying collaborative robot applications. The proposed techniques permit to manage the complexity of specifying collaborative behavior and of managing the communication among robot teams. Concerning the existing work related to the control of quadrotors a very detailed and complete survey on the advances in guidance, navigation, and control of unmanned rotorcrafts systems in general is provided in [13]. Many *algorithms* have been proposed for automatic trajectory generation and control, with a strong focus on either trajectory optimization [14], feasibility [15], or safe obstacle and trajectories intersection avoidance [16].

Differently from the approaches outlined above, our focus is on *i)* the definition of the various tasks of a civilian mission at a high-level of abstraction and *ii)* on the automatic deduction of the behaviour of the robots that will execute the modeled missions. Therefore, the aim of the work that underpins the family of languages presented in this paper is to develop the support for dealing with the specific problem of supporting the specification and execution of civilian missions. In other words, we want to provide non-technical users with the instruments to easily define missions and execute them by means of multitudes of robots. Currently, available technologies somehow permit to develop missions and control the involved robots, even though only software or control engineers and domain experts have the required knowledge and are able to use the

[6] http://robotwebtools.org

complex tools to do so. The proposed languages allow operators to straightforwardly define monitoring missions of swarms of robots by masking all the complexity of the low-level and movement dynamics-related information of the robots.

6 Conclusions and Future work

In this paper we presented a family of domain-specific languages for specifying civilian missions of multi-robot systems. The family of languages is organized in a two-layer architecture, in which the uppermost layer contains languages for the end user, while the other layer, hidden to the user, contains a working language describing the detailed behaviour of each robot of the mission. The family of domain-specific languages has been instantiated to the domain of autonomous unmanned aerial vehicles.

So far, the various tasks in the mission layer are based on their location of interest (i.e., points, lines, and polygons), as future work we are reasoning on how to extend the MML mission layer with timing constraints and other environmental factors, such as path crowdness, convenience (e.g., in terms of used resources), safety, etc. Also, we are planning to experiment the family of domain-specific languages to other kind of robots. This will give us the possibility to further refine the family, and to start experimenting with strategies and mechanisms for combining together different kinds of robots (e.g., ground vehicles with aerial vehicles) that collaboratively perform the same mission. Also, we are analysing the current architecture of the FLYAQ platform in order to better understand how it can be refactored into a fully generic software architecture. In this context, the resulting architecture will enable future third-party software developers and researchers to reuse the generic components of the architecture supporting the core of the DSL family.

References

1. Schmidt, D.: Guest editor's introduction: Model-driven engineering. Computer **39**(2) (Feb 2006) 25–31
2. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5) (September 2003) 19–25
3. Di Ruscio, D., Malavolta, I., Pelliccione, P.: Engineering a platform for mission planning of autonomous and resilient quadrotors. In: Fifth International Workshop on Software Engineering for Resilient Systems, Springer Berlin Heidelberg (2013) 33–47
4. Skrzypietz, T.: Unmanned Aircraft Systems for Civilian Missions. BIGS policy paper: Brandenburgisches Institut für Gesellschaft und Sicherheit. BIGS (2012)
5. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive and Mobile Computing **6**(2) (2010) 161 – 180
6. Lim, H., Park, J., Lee, D., Kim, H.J.: Build your own quadrotor: Open-source projects on unmanned aerial vehicles. Robotics Automation Magazine, IEEE **19**(3) (Sept 2012) 33–45
7. Di Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Developing Next Generation ADLs Through MDE Techniques. In: Procs. ICSE'10, ACM (2010) 85–94
8. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. ICRA workshop on open source software **3**(3.2) (2009) 5

9. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic software systems: From code-driven to model-driven designs. In: Advanced Robotics, 2009. ICAR 2009. International Conference on. (June 2009) 1–8
10. Steck, A., Lotz, A., Schlegel, C.: Model-driven engineering and run-time model-usage in service robotics. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering. GPCE '11 (2011) 73–82
11. Trojanek, P.: Model-driven engineering approach to design and implementation of robot control system. CoRR **abs/1302.5085** (2013)
12. Gtz, S., Leuthuser, M., Reimann, J., Schroeter, J., Wende, C., Wilke, C., Amann, U.: A role-based language for collaborative robot applications. In Hhnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B., eds.: Leveraging Applications of Formal Methods, Verification, and Validation. Communications in Computer and Information Science. Springer Berlin Heidelberg (2012) 1–15
13. Kendoul, F.: Survey of advances in guidance, navigation, and control of unmanned rotorcraft systems. J. Field Robot. **29**(2) (March 2012) 315–378
14. Hehn, M., D'Andrea, R.: Quadrocopter trajectory generation and control. In: IFAC world congress. (2011) 1485–1491
15. Augugliaro, F., Schoellig, A., D'Andrea, R.: Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach. In: Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on. (Oct 2012) 1917–1922
16. Leonard, J., Savvaris, A., Tsourdos, A.: Towards a fully autonomous swarm of unmanned aerial vehicles. In: Control (CONTROL), 2012 UKACC International Conference on. (Sept 2012) 286–291

Towards a General Framework for Modeling, Simulating and Building Sensor/Actuator Systems and Robots for the Web of Things

Ion Mircea Diaconescu[1] and Gerd Wagner[1]

[1]Chair of Internet Technology
Institute of Informatics
Brandenburg University of Technology, Germany
{M.Diaconescu, G.Wagner}@b-tu.de

Abstract. The Web of Things (WoT) refers to those parts of the web consisting of special web application systems connected to the real world via sensors or actuators. These WoT systems include robots connected to the web as a special case. We propose a general framework for modeling, simulating, designing and building WoT systems. We propose a core ontology for WoT systems, which is the basis for our modeling and simulation approach. The modeling and simulation part of our framework is independent of the WoT and could also be employed in the engineering of other forms of embedded systems and robots. As a test case and a proof of concept we present an example of a green house WoT system.

Keywords: Web of Things · sensors · actuators · robots · ontology · modeling · simulation.

1 Introduction

The *Web of Things (WoT)* is a subset of the *Internet of Things (IoT)*. While in the IoT, all kinds of Internet technologies can be used for building sensor-based information systems and device control applications, the WoT is based on web technologies only: foremost DNS, HTTP and HTTP-compatible protocols like Web Sockets and the Constrained Application Protocol (CoAP), and the user interface and frontend computing technologies HTML, CSS and JavaScript. The WoT consists of special web application systems connected to the real world via sensors or actuators, including robots connected to the web as a special case.

There are three recent trends promoting the WoT. First, the web's infrastructure has progressed dramatically 1) by extending the internet's address space with IPV6, 2) by continuously increasing the speed and bandwidth of internet connections, and 3) by improving the speed of HTTP with HTTP 2.0 as well as introducing near-real-time web protocols like Web Sockets. Second, the widespread use of smartphones and tablets, containing various sensors, has created a large pool of sensing and computing resources for the WoT. Third, the

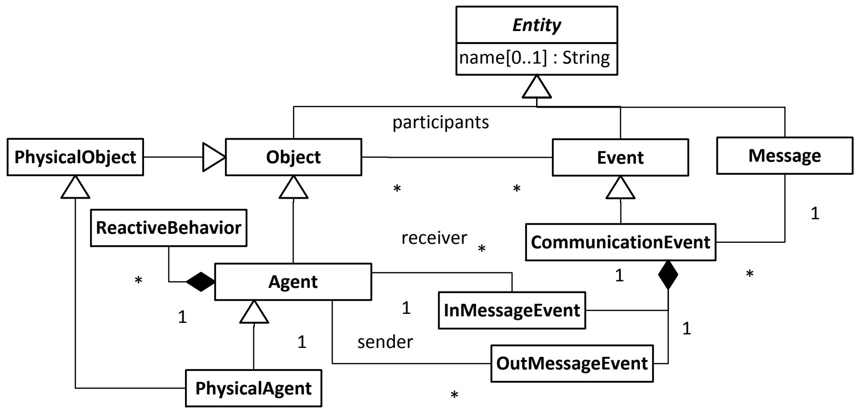


Fig. 1. Top-level concepts of WoTCO

increasing availability of many kinds of cheap sensors, actuators and other electronics components has led to the development of a large *Do It Yourself (DIY)* robotics and WoT community, creating lots of open source software and hardware, and publishing a great variety of DIY projects[1]. The availability of all these resources, and, in particular, of low-cost hardware, creates new opportunities for WoT and robotics-related research and education.

For instance, a simple WoT project can be the temperature monitoring of a room by using a cheap temperature sensor, like the Texas Instruments LM35[2] (available for about 1 Euro), attached to a Raspberry Pi microcomputer (available for about 30 Euro) running a NodeJS-based web application on top of Linux and connected to the Internet via WiFi. More complex WoT systems, like a home security and monitoring system or a home robot that is able to move around and talk to people, can be built with hardware costs of a few hundreds Euro, only, possibly using a no-longer-needed smartphone as the control computer and exploiting its (GSM/3G and WiFi) communication and its (GPS, microphone, camera) sensing capabilities.

A critical issue for any kind of web application, and even more for device control applications in the WoT, is security. However, in this paper we do not treat security issues.

In the robotics and WoT research literature, as well as in the DIY robotics and WoT literature, there is still a lack of methodologies, including general approaches to modeling and simulation. Our aim is to develop a general framework for modeling, simulating, designing and building WoT systems (WoTS). The basis of this framework is a WoTS core ontology defining such concepts as event, object, agent, sensor, actuator, etc.

[1] See, e.g., `http://www.instructables.com/tag/type-id/category-technology/`.
[2] Centigrade Temperature Sensor: `http://www.ti.com/lit/ds/symlink/lm35.pdf`

The last section presents a proof of concept implementation of a WoT system controlling a green house using our open source Java/Android-based WoTS implementation framework.

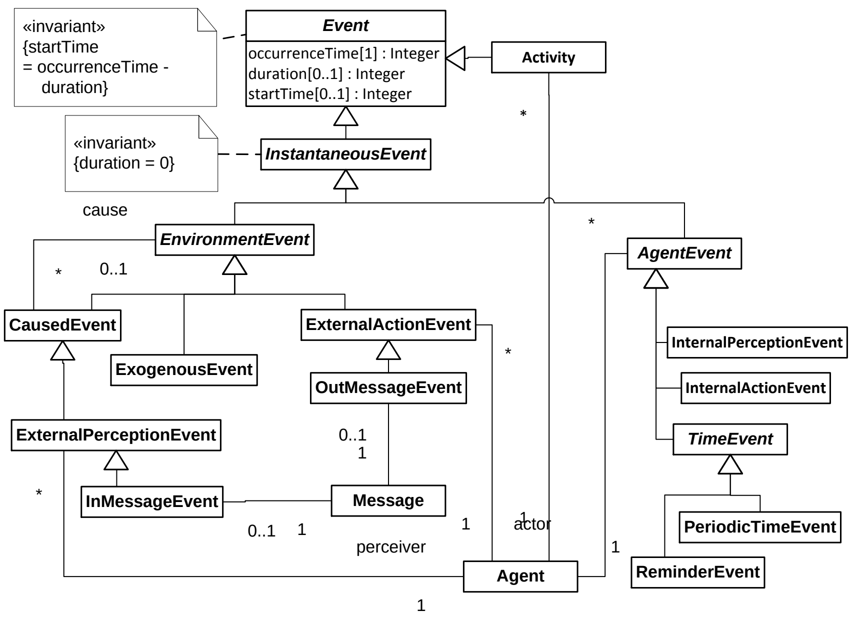


Fig. 2. Top-level event categories in WoTCO

2 A Core Ontology for WoT Systems and Robotics

An ontology is a system of inter-related categories for classifying the things that inhabit (some part of) our real world. A *foundational* (or *upper-level*) ontology identifies the most fundamental categories such as objects and events, while a *core ontology* defines the core concepts of a domain, based on a foundational ontology. We use the *Unified Foundational Ontology (UFO)* proposed by Guizzardi and Wagner in [6,7,8], and we call our core ontology for WoT Systems and Robotics ***WoTCO***.

As can be seen in Figure 1, The most important top-level category in WoTCO is the category of PhysicalAgent, which is derived from both PhysicalObject and Agent. As a physical object, a physical agent is in time and space (and has physical attributes such as mass, spatial coordinates, velocity, etc.), and participates in events. As an agent, a physical agent has reactive behavior and may participate in out-message events as sender, and in in-message events as receiver.

As defined in Figure 3, a WoT system is a physical agent. A WoTS component may be a sensor, an actuator or a human-interface device (HID).

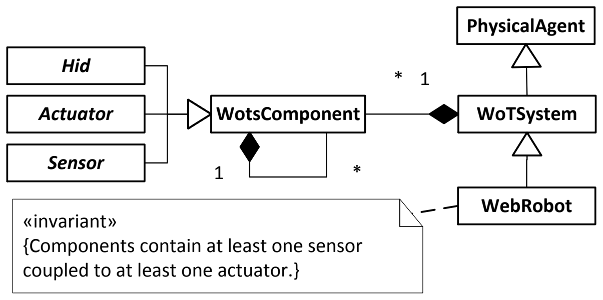


Fig. 3. WoTS components as physical agents

As defined in Figure 2 and 4, we distinguish between environment events, which occur in (and are used for simulating) the environment, and agent events, which occur internally in agents. Our high-level view of the perception-action cycle of a WoTS can be described in WoTCO terms as follows. An external perception event, as an environment event, corresponds to a potential perception event enabled by physical causality. A sensor of a WoTS maps such an external perception event to an internal perception event (or sensor event). Then a reactive behavior rule maps this sensor event to an internal action event (or actuator command), which is mapped to an external action event via the used actuator. The newly created external action event can then cause another external perception event, which starts the cycle over again.

3 Related Work

The IoT-A project has collected a report on the existing frameworks and architectures [1] providing an overview of the current state of the art, and has defined an architectural reference model[2], which is very generic.

The issues of searchability, shareability and composability of WoT systems are discussed in [3].

An attempt to define a core ontology for robotics based on the foundational ontology SUMO is made by the IEEE working group *Ontologies for Robotics and Automation (ORA)* in [9]. Remarkably, this ontology does not include any specific concepts for sensors and actuators, which are subsumed under ”Robot Part”. Many top-level concepts of the ORA ontology are similar to our WoTCO categories, but WoTCO is much more complete.

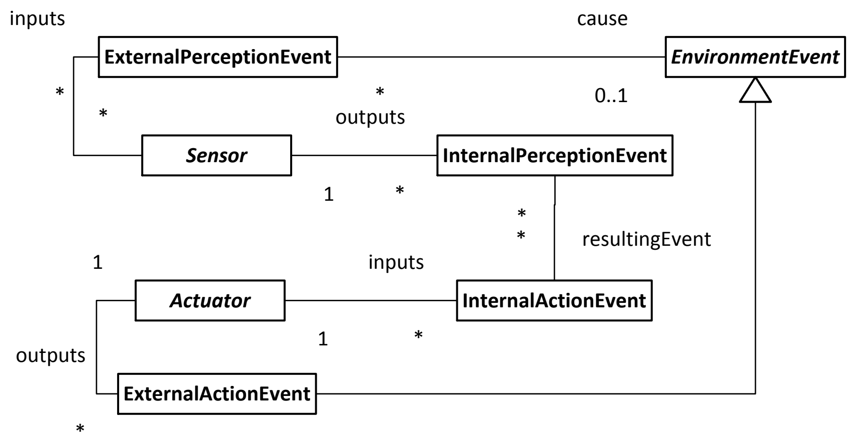


Fig. 4. Sensor/actuator events

4 An Architecture for WoTS and WoTS Simulations

Our goal is to develop a general architecture for modeling, simulating, designing and implementing WoT systems. This means that a WoTS model specifes both both the WoT system to be realized and its simulations, which may be partial in the sense that any number of its components may be present in its configuration while all others are simulated.

4.1 Simulating WoTS Components

Our prototype WoT system presented in the next section is based on the architecture metamodel shown in Figure 5, which is derived from the Agent-Object-Relationship Simulation Metamodel[4,5]. The central concept of this metamodel is `WoTSComponent`, which represents entities that have physical properties (e.g., position, size, speed, etc) and whose reactive behaviors can be described by reaction rules triggered by events. For instance, a motor controller starts its activity when an internal action event "GO" occurs. As shown in Figure 5, a `WoTSComponent` can be simultaneously an event source and an event listener.

A component can be atomic (a `Sensor`, an `Actuator` or a `Hid`) or composite, like, for example, a robot arm composed of a set of interconnected sensors and actuators. Even sensors can be composite devices, as for example a humidity and temperature sensor in a single unit with a single communication interface.

A `WoTComponent` (sensor, actuator, HID) or `WoTSystem` can contain a set of custom defined rules. The rule definition specifies the type of the event which activate it. The project author is free to build simple or complex rules by using the capabilities of the used system implementation programming language.

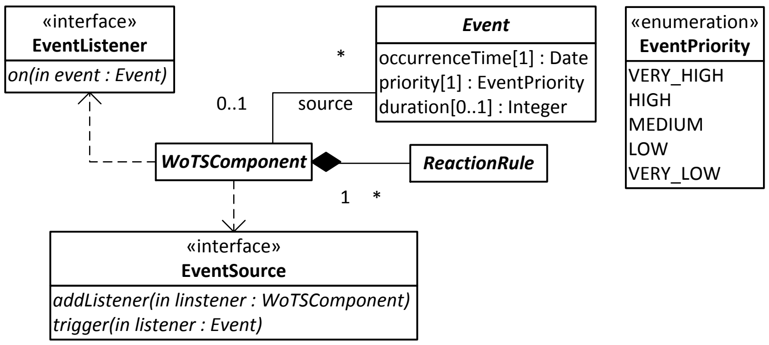


Fig. 5. General architecture model

Also the behavior of a WoT system is defined by reaction rules (e.g., trigger the alarm when an intruder is detected or start watering the flowers when the soil moisture is under a threshold value).

4.2 Sensors

Sensors are mostly used to collect data. In general a sensor can be an atomic component, having just one specific function (e.g., a LM35 centigrade temperature sensor) or a composite one, where multiple sensors are packed in one unit and all of them use the same communication interface (e.g., 1-wire DHT22 temperature and humidity sensor). Sensors can be divided in categories based on their types. Our category divisions (see Figure 6) were obtained by selecting the most relevant types of sensors, according to [11]. In some cases, one category may be further divided, as for example in the case of `WeatherSensor` category, we have: `TemperatureSensor`, `HumiditySensor`, `MoistureSensor`, `BarometerSensor` and so on. For each sensor category (or sub-category) a related builtin event type, or set of event types are defined. The events are forwarded to the registered listeners, responsible to evaluate and use the sensor data by using their rules.

4.3 Actuators

Actuators are in general simple electro-mechanical devices that require a signal (voltage, current or a specific protocols) to activate or deactivate them. The most relevant categories, according to [10], are captured in our model, as shown in Figure 7.

4.4 Human Interface Devices

According to [12], Human Interface Devices (HIDs), represent a special type of WoT Components, and their main purpose is to provide an interface between the

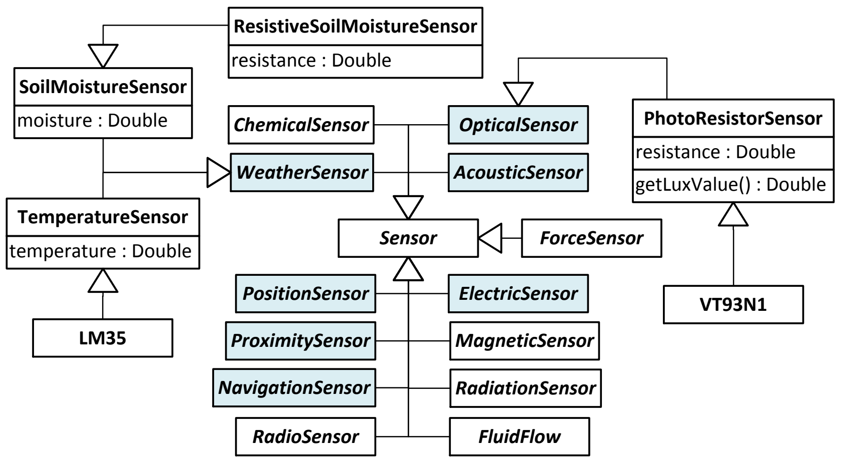


Fig. 6. Sensors architecture model

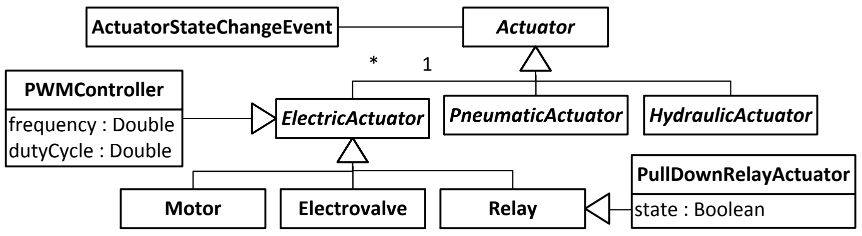


Fig. 7. Actuators architecture model

system and a human user who needs to interact with the system. Such devices can either be input or output devices, but can also be composite devices (providing multiple inputs, multiple outputs or multiple inputs and outputs). Examples of HIDs are displays (with or without touch screen), LEDs, keyboards, etc.

5 Test Case: The Green House Project

This section presents a project as a proof of concept for the proposed architecture. The project is about the implementation of a *Green House*, which is monitored and controlled by a WoT System. Specific needs in terms of temperature, water and light have to be considered for the Green House. The system provides the following functionality:

- soil moisture sensors measure from dry up to flooded soil.
- the temperature is monitored and an automatic cooling system is activated to control the temperature (air flux may come from outdoors).
- a specific light intensity is required for an optimal production.
- the water system can be started or stopped by using an electrovalve.

Figure 8 shows the model instance of the *Green House* project, according with the architecture model discussed in this paper.

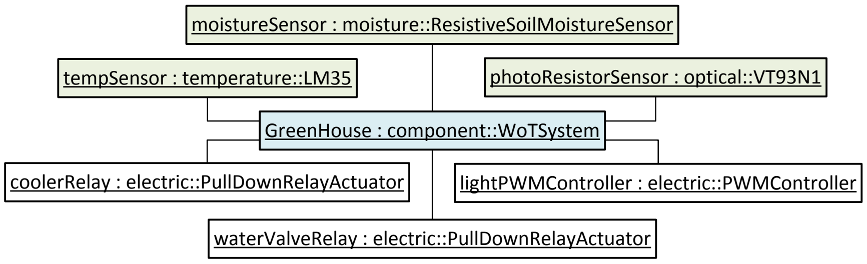

moistureSensor : moisture::ResistiveSoilMoistureSensor
tempSensor : temperature::LM35
photoResistorSensor : optical::VT93N1
GreenHouse : component::WoTSystem
coolerRelay : electric::PullDownRelayActuator
lightPWMController : electric::PWMController
waterValveRelay : electric::PullDownRelayActuator

Fig. 8. Green House test case model instance

5.1 Interfacing with Sensors and Actuators

A WoT project consists of a set of sensors that perceive the environment, a set of actuators with the purpose of performing physical actions, and a computer device connected to them and to the web via a web application. In general, a normal computer or smart device cannot be directly connected to sensors or actuators, but rather an interface board is needed for this purpose. Such a board is a device which allows to communicate via specific protocols (e.g., USB, Serial, Parallel, etc) with the computer and in the same time provides I/O channels (via

GPIO pins) to interface with sensors and actuators. Examples of such boards are: IOIO-OTG[3] and Arduino[4]. One can also use development boards which provide a combination of mini-computers and interface boards in just one device, such as Beaglebone[5] and Raspberry PI [6]. There are some disadvantages in this case since usually the number of available GPIO pins is limited (e.g., Raspberry PI provides only 8, neither having analog capabilities) and others requires advanced programming skills to control the GPIO pins (e.g., Beaglebone requires advanced C/C++ and Assembler knowledges for more than blinking a led projects).

For our project we use the IOIO-OTG interface board, which allows to connect a smart device running Android v2.3 or higher with external components (e.g., sensors and actuators) by using the 46 GPIO pins. It provides multiple interfacing capabilities, such as communication via I2C, SPI and UART protocols, analog data reading (reads voltage within 0-3.3V range) and PWM (pulse width modulation) control with possibility of changing the frequency and duty cycle. The board is connected with the Android device via USB cable or bluetooth. For an optimal usage of the IOIO-OTG interface board, the Android device must have 512MB or more RAM memory and a CPU with a frequency over 1GHz.

The board itself does not require custom software to interface with external components. Instead it interfaces with the Android device via a Java API, which is rather generic and does not provide specific implementation to interface with sensors or actuators, this being part of the custom project software implementation. Our Java prototype of the proposed WoT architecture includes the IOIO-OTG API and extends it with specific sensors and actuators implementation, as the ones (but not only) used for the Green House project.

5.2 Hardware Configuration

The system consists of the following hardware parts:

- A Samsung Galaxy S (GT-I9000) smartphone running Android version 4.3.1.
- A IOIO-OTG board is used as an interface between the smartphone and the system. It connects with the smartphone via bluetooth or USB cable.
- The LM35 centigrade analog sensor is used to monitor the temperature. It represents a specific `TemperatureSensor` implementation, part of the `WeatherSensor` category (see Figure 6 and Figure 8).
- The VT93N1 photo-resistor sensor, is used to detect the light intensity. It represents a specific `PhotoResistorSensor` implementation, part of the `OpticalSensor` category (see Figure 6 and Figure 8).
- DIY custom soil moisture sensors are created with the help of nickeled nails, wires and resistors. Those sensors represents a specific implementation of `ResistiveSoilMoistureSensor`, a specific subclass of `SoilMoistureSensor`, part of the `WeatherSensor` category (see Figure 6 and Figure 8).

[3] IOIO/IOIO-OTG - `https://github.com/ytai/ioio/wiki`
[4] Arduino - `http://www.arduino.cc/`
[5] Beaglebone - `http://beagleboard.org/Products/BeagleBone`
[6] Raspberry PI - `http://www.raspberrypi.org/`

– The electrovalve used to start/stop watter supply is activated and deactivated by using a `PullDownRelayActuator` (see Figure 7 and Figure 8). Such a relay type is closed by default, and is activated by connecting it to the ground of the power supply, from here its ”pull down” name.
– A set of mains powered coolers are used to keep the temperature in a specified range. `PullDownRelayActuator` relays are used to control their on/off states. The airflow used to adjust the temperature level comes from outdoors.
– A recycled ATX PC power supply (provides 12V and 5V at high current levels) is used to power all components except the ones being connected to mains power supply (cooling system).

The total cost of the system is about 200€, from which the sensors and the IOIO interface board cost about 50€. The rest of the price is for the electrovalve, relays, coolers and dimmable lights. The smartphone price is not included, but is currently evaluated to about 50-60€ on the market.

5.3 Software Configuration

Our WoT Java/Android implementation is used to implement the system software. It contains the code required to read the sensors and control the actuators. As already discussed in this paper, our architecture uses an event based communication between components and the system behavior is defined by using reactive rules. For this project a set of rules are used to control the actuator components based on various sensor readings. For readability reasons, a pseudocode version of the rule is shown in this paper, but its Java version (as used by our architecture implementation) is also simple to write.

Temperature and Soil moisture Control: The LM35 sensor is used in `AUTO` mode, thus creating `TemperatureSensorEvents` (builtin event type which carries the temperature value) only when temperature value changed compared with latest known value. Controlling the cooling system is performed by using a rule shown below:

```
WHEN TemperatureSensorEvent event
if ( event.getTemperature() < lowRange)
then CREATE DisableRelayEvent( CoolerRelay)
elseif (event.getTemperature() > highRange)
then CREATE EnableRelayEvent( CoolerRelay)
```

The coolers are started if a high temperature is detected. When the temperature goes back in the normal range, the coolers are stopped. The temperature is maintained in the specified range with the condition that outdoors temperature (from where the airflux come) is below the highest temperature value specified by our system. The same considerations are used to control the soil moisture, the differences being the type of event which triggers the rule (`SoilMoistureSensorEvent`) and the moisture threshold values.

Lights Control: Using PWM (pulse width modulation) one can control a light system to have not only *light on* and *light off* light states, but also various intermediate light intensity levels. Using the IOIO board we generate the PWM signals to control a set of PWM controlled dimmable lights. The following rule allows to control the light by changing the PWM duty cycle:

```
WHEN LightSensorEvent event
if ( event.getLuxValue() > highRange)
then CREATE DisablePWMEvent( PWMLight)
else DEFINE VAR dutyCycle = (targetLuxValue - event.getLuXValue()) / 100
     CREATE ChangePWMDutyCycleEvent( PWMLight, dutyCycle)
```

The sensor returns values between 0 Ohm (direct sun light) and 300K Ohm (complete dark) which are internally converted to LUX values. Increasing the PWM duty cycle results in higher light levels. The `targetLuxValue` represents the target light intensity value (in LUX) for our Green House.

Safety Considerations: A WoT system presents safety risks in some cases. For example the malfunction of the soil moisture sensor in the case of the Green House project may result in flooding the plants. We are working on a solution to categorize the WoT components and events so that posible safety risks are limited as much as possible. Additionally, implementing some WoT systems may require to work with possible dangerous voltage levels for the human body, e.g., using mains power. Such safety risks must be considered by the hardware project author.

Project Enhancements: The project was prototyped in a room by replacing the mains powered coolers with PC coolers and the electrovalves with LEDs. The project will be improved by allowing a human user to interfere with the automated actions if required (e.g manually start or stop the water or coolers). Additionally, a data collector component will be added to have statistics about the expenses by monitoring the consumed water and electricity. This is possible by using sensors to read and monitor consumed electrical power and water volume.

6 Conclusions

We have presented an ontology and metamodels for modeling, designing and simulating WoT systems. A simple, but illustrative, test case implementation was shown as a proof of concept. We still have to make our framework more complete, e.g., by developing a general approach how to create simulation models for specific sensors and actuators based on their technical specification provided by the vendor.

References

1. Consorzio Ferrara Ricerche. Project Deliverable D1.1 - SOTA report on existing integration frameworks/architectures for WSN, RFID and other emerging IoT related Technologies, Alessandro Bassi (Eds.), 2011, `http://www.iot-a.eu/public/public-documents/documents-1/1/1/d1.1/at_download/file`
2. FhG IML. Deliverable D1.3 Updated reference model for IoT v1.5, Andreas Nettstrter (Eds.), 2012, `http://www.iot-a.eu/public/public-documents/documents-1/1/1/D1.3/at_download/file`
3. Dominique Guinard. A Web of Things Application Architecture - Integrating the Real-World into the Web, 2011, `https://www.webofthings.org/dom/thesis.pdf`
4. Gerd Wagner. AOR Modelling and Simulation Towards a General Architecture for Agent-Based Discrete Event Simulation. In P. Giorgini et al. (Eds.): Agent-Oriented Information Systems, Springer-Verlag LNAI 3030, pp. 174188, 2004.
5. Gerd Wagner. A Short Introduction to the ER/AOR Simulation Framework. `http://hydrogen.informatik.tu-cottbus.de/talks/AORS-Tutorial/`
6. Giancarlo Guizzardi and Gerd Wagner. A Unified Foundational Ontology and some Applications of it in Business Modeling. In Proceedings of the CAiSE'04 Workshops, edited by J. Grundspenkis and M. Kirikova, 3:129-143. Faculty of Computer Science and Information Technology, Riga Technical University, Riga, Latvia. June 7-11, 2004.
7. Giancarlo Guizzardi. Ontological Foundations for Structural Conceptual Models. PhD Thesis, University of Twente, The Netherlands. 2005.
8. Giancarlo Guizzardi and Gerd Wagner. Using the Unified Foundational Ontology (UFO) as a Foundation for General Conceptual Modeling Languages. In Roberto Poli (Ed.), Theory and Application of Ontologies, 175-196. Springer-Verlag Berlin/Heidelberg, 2010.
9. Edson Prestes and Joel Luis Carbonera and Sandro Rama Fiorini and Vitor A. M. Jorge and Mara Abel and Raj Madhavanb, Angela Locoro and Paulo Goncalves and Marcos E. Barreto and Maki Habibg and Abdelghani Chibani and Sbastien Grard and Yacine Amirat and Craig Schlenoff. Towards a core ontology for robotics and automation. Robotics and Autonomous Systems 61 (2013), 1193-1204.
10. Wikipedia: Actuator, `http://en.wikipedia.org/wiki/Actuator`.
11. Wikipedia: List of sensors, `http://en.wikipedia.org/wiki/List_of_sensors`.
12. Wikipedia: Human interface device, `http://en.wikipedia.org/wiki/Human_interface_device`.

Towards Context Modeling in Space and Time

Christian Piechnick, Georg Püschel, Sebastian Götz,
Thomas Kühn, Ronny Kaiser, and Uwe Aßmann

Software Technology Group, Technische Universität Dresden,
Nöthnitzer Str. 46, 01187 Dresden, Germany
{christian.piechnick,georg.pueschel,sebastian.goetz1,thomas.kuehn3,ronny.kaiser,uwe.assmann}@tu-dresden.de
http://st.inf.tu-dresden.de

Abstract. One of the main problems in software development for service robots is to create systems that reliably behave as intended, even though the real field of application and the concrete user requirements are unknown during design time. Consequently, the software controlling service robots has to be aware of its environment and has to adapt its behavior accordingly. A model representing environmental data is called a context model. Appropriate context models currently lack means for modeling temporal and spatial information simultaneously. While it is important to reason about historical context data for most of the Self-Adaptive Systems, there is an increasing need for treating the temporal dimension of context models as first-class-citizen. In this paper, we propose a graph- and role-based context model (GRoCoMo), which includes expressive means for describing time and location. A query language enables for reasoning on current and historical data, as well as future trends. A manipulation language enables the specification of rewrite rules for updating context models based on situations detected within the context.

Keywords: Context Modeling; Context Management; Context-Awareness; Temporal Context; Context History;

1 Introduction

A service robot is a reprogrammable, sensor-based, mechatronic device which performs useful services to support human activities [8]. In contrast to production robots, where the operating environment as well as all other influencing factors are known before deployment, the application sites of service robots are unknown. This information can only be gathered during runtime. Therefore, the software system controlling the service robot has to adapt its behavior dynamically. Such a system is called a Self-Adaptive System (SAS). One example of adaptive behavior within robotic software is the path planning of mobile robot platforms [4]. When a robot has to move to a target position, the robot's motion model (e.g., differential steering), as well as the environment (e.g., crowded areas), influence the planning strategy. Figure 1 shows an example path planning problem with two different strategies. The first strategy, *Shortest Path* (solid

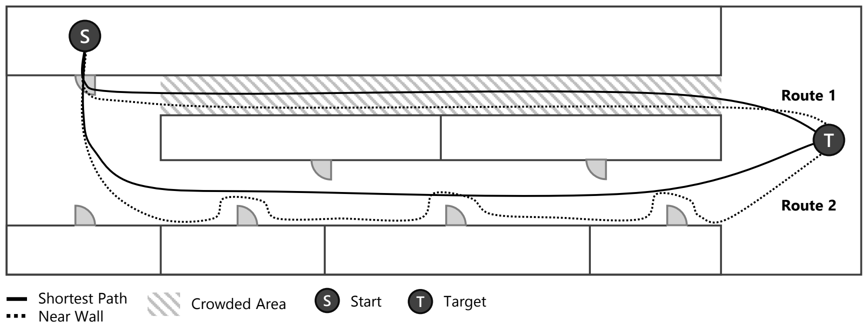


Fig. 1. Example for alternatives in global path planning

lines), calculates the shortest route from the starting location (S) to the target location (T) and provides two different alternatives. The other strategy, *Near Wall* (dashed lines), calculates routes that lead along walls. The grey dashed area is usually used by many persons and therefore, marked as "crowded". When the navigation algorithm decides to use the upper floor, the *Near Wall* strategy is the better option, since the probability to get in the way of humans is decreased. On the other hand, it might be disadvantageous on the lower floor, where the robot potentially has to drive around many open doors. Hence, the decision should be made at runtime. An inherent property of all SAS is that they are implementing a variant of the MAPE-K loop, first introduced by IBM [7]. The MAPE-K loop consists of four phases: in the *(M) Monitor* phase environmental data is gathered and processed and, then, interpreted in the *(A) Analyze* phase. The *(P) Plan* phase investigates the need for reconfiguration and creates reconfiguration plans accordingly, which are applied in the *(E) Execute* phase. All phases share a *(K) Knowledge Base*, which manages relevant information guiding the adaptation process. An essential part of this knowledge base is information about the execution environment (e.g., crowded areas in the upper example), i.e., the **Context Model**. Because different domains have varying requirements w.r.t. context modeling and management, various different modeling approaches for context information were developed during the last decades. Nevertheless, recent context modeling approaches are not sufficient to handle crucial aspects for the domain of service robots. Namely, reasoning on dynamic collaborations like in robotic applications, requires more expressive means to cover time and location in a processable manner. To address this problem, we present an extended graph-based context model with time and location as first class citizens and, thus, extended means for modeling and managing temporal and spatial context data for robotic applications.

This paper is structured as follows. In Section 2, we give an overview on context modeling and outline important properties of context models. In Section 3, we discuss relevant context modeling approaches and their suitability w.r.t. the identified properties. We present our approach in Section 4 and discuss our prototypical implementation in Section 5. Finally, Section 6 presents our conclusion and future work.

2 Context Modeling and Management

To enable an application to adapt itself to changing environmental conditions, information about the environment must be gathered and analyzed. For this purpose, a variety of approaches have been developed, to tackle the specific requirements in different application domains of SAS [2]. Zimmermann et al. identified six different modeling elements of context models [17]:

Z1 - Entities: An entity can be any real or virtual thing that is of interest for the adaptation process (e.g., user, device, application, location, etc.).

Z2 - Individuality: The individuality encompasses any information that can be observed about an entity (e.g., dynamic and static properties, etc.).

Z3 - Relationships: A relationship expresses a semantic dependency between two entities. Zimmermann et al. distinguish between social-, functional-, and compositional relationships.

Z4 - Activities: The activities dimension encompasses any information about an entity's past, present and future needs, goals, tasks and plans.

Z5 - Time: Statements in a context model often have a temporal dimension. Time can be expressed using time zones (e.g., Central European Time) or virtual times (e.g., milliseconds after system start). Furthermore, overlay models can be used for abstraction (e.g., working hours, weekends, etc.).

Z6 - Location: Since most of a context model's elements represent objects from the physical world, which are arranged spatially, location is a major aspect of context information. The location dimension can include real or virtual locations (e.g., IP address in a network). Those locations can use absolute, relative, or symbolic location models.

Strang et al. [13] identified six different types of context modeling approaches: (1) Key-Value Models, (2) Markup Scheme Models, (3) Graphical Models, (4) Object Oriented Models, (5) Logic Based Models, and (6) Ontology Based Models. Depending on the specific requirements of the application domain, different advantages and disadvantages can be observed. They evaluated those types of context models regarding their ability to (a) be composed in a distributed computation environment, (b) the richness and quality of information, (c) the ability to handle incompleteness and ambiguity, (d) the level of formality, and (e) their applicability to existing environments. Considering those properties, Strang et al. conclude that ontologies are the best-rated modeling type, while Key-Value Models are the worst-rated. On the other hand, the construction and management of Key-Value Models is much simpler and the performance of analysis scales much better for simple requests. For the domain of service robots the properties (b), (c) and (e) are crucial because of the robots complex and unknown execution environment. The properties (a) and (d) become important when the sensors (e.g., temperature sensor) and actuators (e.g., door opening) are distributed across the environment, and, thus, multiple computational units have to share knowledge based on a shared interpretation. Hence, according to the provided evaluation, ontologies should be used for the modeling of contextual information in the domain of service robots. Furthermore, a context model for service robots should include the modeling elements *Z1 - Z6*, according to Zimmermann et al. [17].

3 Related Work

The research area of SAS is still very popular, resulting in thousands of publications each year. This is also true for research on context modeling and management. For our related work research, we searched for papers published between 2000 and 2013 and containing the words "*context model*" in their title, using Google Scholar[1]. The result of the indicated query was a set of 3469 papers. We filtered the result-set manually to exclude publications that were not intended for the application in SAS. The result was a reduced set of 1228 manually filtered papers. Among them, we investigated five context model survey papers [1–3, 6, 13]. We have chosen six representative context model publications, which consider the dimensions *time (Z3)* and/or *location (Z4)*. Four of those papers [5,12,14,15] were chosen based on the description in the context model surveys. Because the latest survey was published in 2010 by Bettini et al. [2], we have selected two additional publications [9, 16], published between 2010 and 2014. We have investigated their modeling capabilities w.r.t. the properties stated in Section 2. The results are summarized in Table 3.

In 2003 Strang et al. proposed the ontology-based context model **CoOL** (Context Ontology Language) [14]. CoOL provides the concept of an "Entity", while type information (e.g., Person, Place) must be expressed using the individuality dimension. *Individuality (Z2)* can be modeled using "Aspects" with different "Scales". *Relationships (Z3)* can be expressed using facts. Even though they show that the *time, place and activity dimensions (Z4 - Z6)* can be treated as an aspect as well, they do not provide a special interpretation semantic for time-bound, historical, location-, or activity-specific data.

Gu et al. presented an ontology-based context model for their service-oriented context-aware middleware **SOCAM** in 2004 [5]. They support several types of entities (e.g., Device, Network) as well as predefined and user-defined properties and relationships (*Z1 - Z3*). Activities are also treated as first-class-citizens (Z4). Time (*Z5*) is partially considered, but only as start and end. The model provides special nodes for locations (i.e., in- and outdoor locations) but does not show how locations can be related.

Wang et al. proposed the CONtext ONtology (**CONON**) [15] in 2004, by extending the SOCAM context ontology. In contrast to the previous model they provide means for describing location (*Z6*) in a more fine-grained manner and

[1] Google Scholar: http://scholar.google.de/, visited 20.05.2014

	CoOL	SOCAM	CONON	MUSIC	ERMHAN	CACOnt
(Z1) Entities	(+)	+	+	(+)	+	+
(Z2) Individuality	+	+	+	+	+	+
(Z3) Relationships	+	+	+	+	+	+
(Z4) Activities	(−)	+	+	(−)	+	+
(Z5) Time	(−)	(−)	(−)	(−)	−	−
(Z6) Location	(−)	(+)	+	(−)	+	+

Table 1. Evaluation of the related models w.r.t. to their modeling capabilities. (- not considered, (-) partially considered, (+) implicitly provided, + fully provided)

explain how those locations can be related to each other, to create hierarchical location models.

Reichle et al. described an ontology-based context model for the **MUSIC** project in 2008 [12]. Like the CoOL ontology, they provide an abstract type `Entity` which can be categorized using special type attributes (*Z1*). The model provides means for describing attributes and relationships (*Z2 and Z3*), but does not treat activities as special entities (*Z4*). Hence, activities can only be modeled by creating used-defined activity type attributes. The model contains basic types, such as `DateTime` or `GPS-Coordinate`, but does not provide a first-class-citizen interpretation semantics for time and location (*Z5 and Z6*).

In 2011, Paganelli and Giuli presented a context model for the **ERMHAN** service platform for Ambient Assisted Living (AAL) scenarios [9]. The model provides means for describing several entity types, attributes, relationships and activities (*Z1 - Z4*). Furthermore, they consider several types of interrelated locations (*Z6*), but do not consider time (*Z5*) within the model. They only consider time externally by tracking the change of context values. Based on the type of the changed value, they interpret a time-bound sequence of values.

In 2013, Xu et al. presented the Context-Aware Computing Ontology **CACOnt**. It predefines several types of entities, properties, relationships and activities (*Z1 - Z4*). The authors extensively investigate the location dimension (*Z6*), by providing different levels of abstraction for the specification of an entities location (e.g., GPS, location hierarchies). They do not consider the time dimension (*Z5*). Thus, a CACOnt model only provides information on the current context state. However, like in every model with extensible attributes, it is possible to express time information using attributes with a custom interpretation logic.

As shown in Table 3, the presented context models provide means for modeling the dimensions of entities, individuality, and relationships (Z1-Z3). The activity dimension (Z4) is either provided, directly or can be modeled separately, using an extensible entity-model. The most recent works consider the location (Z6) as an essential part of a context model, and, thus, provide means

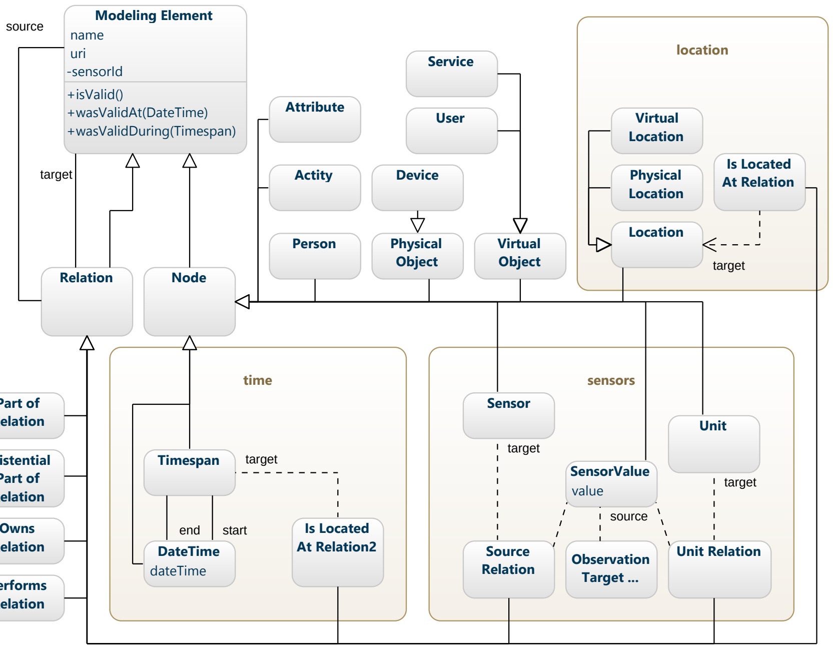


Fig. 2. The GroCoMo-Core metamodel.

for handling spatial information as a first-class-element of context models. The time dimension (Z5), however, is considered important in state-of-the-art literature, but current context models do not provide explicit modeling elements to handle time appropriately.

4 Context Modeling in Space and Time

In this section, we present our Graph- and Role-based Context-Model (**GRoCoMo**), a context model supporting all modeling dimensions stated in Section 2.

4.1 Structure

As depicted in Figure 2, the context model consists of `Nodes` and `Relations`. Both, `Node` and `Relation` inherit from the abstract type `Modeling Element`. Each element has a `name`, a `sensorId` to identify values created by the same sensor and a unique resource identifier (URI), to identify the individual element. A `Relation` connects exactly one `Source Element` to exactly one `Target Element`. Both, source and target, are of the type `Modeling Element`. Hence, it

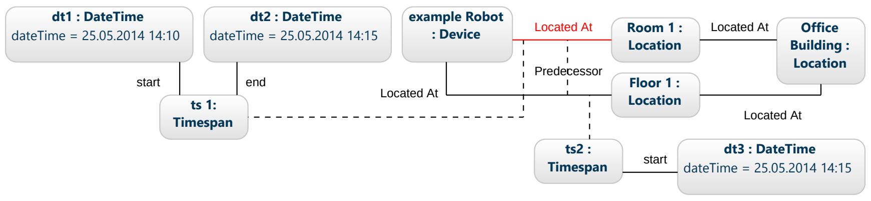


Fig. 3. Example model for time and location information representation.

is possible to define relations on relations. A node represents an entity of the context model. The model provides predefined node types (e.g., Person). However, the metamodel can be extended with domain-specific node types by subclassing. A `Relation` represents a typed, complex relationship between two entities. Furthermore, each relationship can contain several directly assigned attributes.

4.2 Handling Time

To cover temporal aspects, `Timespans` can be assigned to each modeling element (i.e., nodes and relations), to state when the validity of a modeling element started and stopped. Each modeling element with a validity timespan that has no associated end-time, is considered valid at the current time. By default the validity of a modeling element starts when it is created and can be invalidated by assigning an invalidation time. Because each element can have multiple validity times, a previously invalidated node or relation can be re-validated again. Each modeling element may have a `Predecessor Relation` to another modeling element of the same type that was replaced by the respective element w.r.t. its validity. In the scenario described in Section 1, the robot moves from a starting location `S` to a target location `T`. While the robot moves, a node, representing the robot, contains an outgoing `LocatedAt` relation. As shown in Figure 3, when the robot moves from `room1` to `floor1` the first `LocatedAt` relation is invalidated and replaced by a new relation. Because the invalidated relation is not deleted from the model, it is still accessible and can later be analyzed (e.g., by creating motion profiles). Furthermore, different representations of "time" can be modeled. As shown in Figure 3, date and time combinations can be represented as absolute timestamps in a given calendar. Hence, it is important to have an associated location to every timestamp representation, which is inferred in the provided example, because the timespan is assigned to a `LocatedAt` relation. Furthermore, other representations of time (e.g., weekdays or holidays) can be modeled and assigned to nodes and relations.

4.3 Handling Location

Locations are represented by special `Location` nodes (see Figure 2). The model separates physical (e.g., the main station) or logical locations (e.g., a folder in a file system). Physical locations can further be divided into sub-symbolic (e.g., GPS coordinates) or symbolic (e.g., Dresden Main Station) locations. To relate an entity to a location node, the GRoCoMo metamodel provides a generic `LocatedAt` relation. The target of such a relation must always be a location node. When the source node is a location node as well, the relation represents a part-of relation for a specified point in time (e.g., Dresden Main Station *Is Located At* Dresden). The `LocatedAt` relation is transitive. Lets consider for example a person is located in a car and this car is located in the city of Dresden. In this case, the person is located in Dresden as well. While the car changes its location when it is moving, the driver will not change its position relative to the car, but its location relative to the geographical location. The part-of relation on locations forms a graph that has no cycles.

4.4 Context Model Query

In order to query the context model, we have created a first prototype for a query language (GRoCoMo-QL) based on pattern matching in graphs. Listing 1.1 shows an example. Each query starts with a definition of roles. Each role has an `id` (e.g., node1) and represents a node with an optional type constraint (e.g., Location). Then, relations can be defined. Each relation has an `id`, an optional type constraint, a source and a target role, as well as a temporal constraint. The last part of the query is a restriction clause, where any restrictions on the structure of the previously defined roles and relations can be specified. The query from Listing 1.1 will return all tuples `(node1, node2, rel1)`, where `node1` is a location node, `node2` is a device node and the name attribute of `node2` has the value `‘‘Example Robot’’`. Furthermore, both nodes must be connected by a `LocatedAt` relation from `node1` to `node2`. As a temporal constraint, within all results it is guaranteed, that all nodes and relations were/are valid at the same time and only nodes and relations are considered, that were valid within the last 5 minutes.

```
nodes {
 node1 : Location [valid within last 5min];
 node2 : Device [valid within last 5min];
}
relations{
 rel1 : LocatedAt(node2,node1) [valid within last 5min];
}
where node2.name = "Example_Robot";
```

Listing 1.1. A GRoCoMo-QL example.

4.5 Context Model Manipulation

The context model can be changed using a sequence of the following basic graph rewrite operations:

Add Node/Relation: This operation adds a node/relation to the graph. The concrete type is specified on the client side.

Remove Node/Relation: This operation takes the `id` of a node/relation as an input and will remove the corresponding node. In contrast to the invalidation operation, a deletion will irreversibly remove the node.

Invalidate Element: The invalidation operation of a modeling element (i.e., nodes and relations) sets the end time of the corresponding element to the current time. Hence, it will be considered invalid.

Validate Element: Analogously to the invalidation, the validation operation will create a new valid timespan and sets the start time to either the current or the provided time.

Set Property: Some of the GroCoMo meta-classes (cf. Figure 2) define built-in properties (e.g., name). Those properties can be changed using this operation. The changes of built-in properties are not tracked (w.r.t. historical data).

From those basic operations, complex operations can be composed (e.g., replace node, set attribute, set location). The context model can be manipulated by (a) *sensors*, (b) *inference-*, and (c) *cleanup units*. Sensors observe the environment (physical or virtual) and update the context model accordingly. Inference units enrich the context model with new nodes and/or relations based on analysis of the available data in the context model. Cleanup units remove nodes and relations based on application- and hardware-specific rules in order to avoid memory overloads. To express the manipulation of those different manipulation units, we have created a prototypical manipulation language (*GRoCoMo-ML*), based on the Query Language sketched in Section 4.4. Listing 1.2 shows an example. A manipulation script consists of a set of labeled situations, where each situation contains exactly one query. Then, conditions on the results of the corresponding queries can be stated. The match/mismatch of situations can be combined using logical operators (e.g., `and`, `or`, etc.), as well as aggregation operations stated on the number of matches. In the provided example, the corresponding sequence of manipulation operations is executed, when the pattern, described in the situation `"PersonInRoom1"`, is matched more than 5 times.

The presented context model GRoCoMo provides a predefined set of node-types (e.g., Person, Activity, Location, Time), representing contextual entities (supporting modeling dimension *Z1*). The core model introduces specific nodes (i.e., `Attribute Node`) and specific relations (i.e., `HasAttribute` relation) to model the individuality of entities (dimension *Z2*). Through this approach dynamic complex types can be modeled by creating nested attributes. Relations represent relationships either between entities or between other relations (dimension *Z3*). Activities can be modeled using special `Activity` nodes (dimension *Z4*). In order to express temporal and historic data (dimension *Z5*), valid times by means of `Timespans` can be assigned to each node and relation, to express when the validity of a modeling element started and ended. Beside the represented timestamps, it is also possible to assign symbolic representations of

```
Situation "PersonInRoom1"{
 nodes {
  room1 : Location;
  person : Person;
 }
 relations{
  rel1 : LocatedAt(person,room1);
 }
 where room1.name == "kitchen";
}
ON Count(PersonInRoom1) > 5 {
 an = new AttributeNode(name = "is␣crowded", value=true);
 rel = new AttributeRelation(source = room1, target = an);
}
```

Listing 1.2. A GRoCoMo-ML Example

time (e.g., Monday, Holiday etc.). Finally, spatial information is captured by `Location` nodes (dimension *Z6*).

5 Implementation

To investigate the feasibility of the presented approach, we have created a prototypical implementation using the role-based self-adaptive system *Smart Application Grids* (SMAGs) [10]. SMAGs is a component-based modeling and execution approach for runtime reconfiguration. SMAGs defines a predefined implementation for a MAPE-K loop, which can be adapted at runtime [11] as well. We implemented the GRoCoMo as a special `Context Model` component and integrated the Query Language and the Manipulation Language in the Sensor, Inference and Adaptation Component. For the context model representation, we used the JUNG[2] graph framework. For the pattern matching, we used the GUERY[3] framework. GUERY defines a textual syntax for *Motifs*, representing patterns, which are either provided by simple text files or can be created using an object-oriented API. On top of GUERY, we defined two Domain-Specific Languages (DSLs) for the GRoCoMo-QL and -ML using the Eclipse-based DSL-framework Xtext[4]. Instances of GRoCoMo-QL, as well as the query parts from GRoCoMo-ML, are transformed to valid GUERY-queries. Based on the result propositions in the manipulation language, the results of the queries are investigated and based on the evaluation of the situation guards, the provided reconfiguration scripts are executed accordingly.

[2] JUNG: http://jung.sourceforge.net/ (visited 20.05.2014)
[3] GUERY: https://code.google.com/p/gueryframework/ (visited 20.05.2014)
[4] Xtext: http://www.eclipse.org/Xtext/ (visited 20.05.2014)

6 Conclusion and Future Work

The domain of service-robots highly requires software systems that adapt their behavior based on past, present and potential future situations of the involved system. The MAPE-K loop represents the adaptation process from data acquisition, to system reconfiguration, based on data stored in a shared knowledge base. An important part of this knowledge base is the context model, capturing environmental data. It was observed, that structured knowledge representations (e.g., ontologies) are best suited for modeling open and unknown environments. Current approaches, however, fail to support context data analysis over time and location simultaneously. In this paper, we have proposed the context model **GRoCoMo** (Graph- and Role-Based Context Model), using a typed, attributed and directed graph as a foundation. The model supports different predefined entities and relations, which can be extended for specific domains. The model treats activities, time and location as first-class-citizens. For temporal information, validity-timespans are attached to each modeling element, representing the timespan when an element is/was valid (w.r.t. a specific location). To model locations, the model provides specialized location nodes and relations, as well as a transitive semantics for those relations. We have outlined a first version of a pattern-based *Query Language*, as well as a *Manipulation Language* using pattern-based situation detection and a set of predefined manipulation operations, to change the context model. For future work, the provided prototypical implementations of the API and the corresponding languages have to be finished, stabalized and published. In addition, the presented approach has to be evaluated in real-world examples. Bettini et al. [2] described additional properties of context models that mainly focus on data quality. Those properties were already considered, but were not described in this paper. Those properties have to be investigated, covered and evaluated as well. Finally, it has to be investigated how pattern recognition techniques can be used to automatically detect situations in terms of context graph patterns, enabling machine-learning adaptation strategies, as well as situation specification guidance.

Acknowledgment

This work is supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" and the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

References

1. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. Int. J. Ad Hoc Ubiquitous Comput. 2(4), 263–277 (Jun 2007)
2. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive Mob. Comput. 6(2), 161–180 (Apr 2010)

3. Bolchini, C., Curino, C.A., Quintarelli, E., Schreiber, F.A., Tanca, L.: A data-oriented survey of context models. SIGMOD Rec. 36(4), 19–26 (Dec 2007)
4. Crowley, J.L.: Navigation for an intelligent mobile robot. Robotics and Automation, IEEE Journal of 1(1), 31–41 (1985)
5. Gu, T., Wang, X.H., Pung, H.K., Zhang, D.Q.: An ontology-based context model in intelligent environments. In: In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference. pp. 270–275 (2004)
6. yi Hong, J., ho Suh, E., Kim, S.J.: Context-aware systems: A literature review and classification. Expert Systems with Applications 36(4), 8509 – 8522 (2009)
7. IBM Corp.: An architectural blueprint for autonomic computing. IBM Corp., USA (Oct 2004)
8. Kawamura, K., Pack, R., Iskarous, M.: Design philosophy for service robots. In: Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on. vol. 4, pp. 3736–3741 vol.4 (Oct 1995)
9. Paganelli, F., Giuli, D.: An ontology-based system for context-aware and configurable services to support home-based continuous care. Trans. Info. Tech. Biomed. 15(2), 324–333 (Mar 2011)
10. Piechnick, C., Richly, S., Götz, S., Wilke, C., Aßmann, U.: Using role-based composition to support unanticipated, dynamic adaptation-smart application grids. In: ADAPTIVE 2012, The Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications. pp. 93–102. Nice, France (2012)
11. Piechnick, C., Richly, S., Kühn, T., Götz, S., Püschel, G., Amann, U.: Contextpoint: An architecture for extrinsic meta-adaptation in smart environments. In: ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications. Venice, Italy (2014)
12. Reichle, R., Wagner, M., Khan, M.U., Geihs, K., Lorenzo, J., Valla, M., Fra, C., Paspallis, N., Papadopoulos, G.A.: A comprehensive context modeling framework for pervasive computing systems. In: Proceedings of the 8th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems. pp. 281–295. DAIS'08, Springer-Verlag, Berlin, Heidelberg (2008)
13. Strang, T., Linnhoff-Popien, C.: A context modeling survey. In: In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp - Sixth International Conference on Ubiquitous Computing, Nottingham/England (2004)
14. Strang, T., Linnhoff-Popien, C., Frank, K.: Cool: A context ontology language to enable contextual interoperability. In: Distributed Applications and Interoperable Systems, Lecture Notes in Computer Science, vol. 2893, pp. 236–247. Springer Berlin Heidelberg (2003)
15. Wang, X., Zhang, D.Q., Gu, T., Pung, H.: Ontology based context modeling and reasoning using owl. In: Pervasive Computing and Communications Workshops. pp. 18–22 (March 2004)
16. Xu, N., Zhang, W.S., Yang, H.D., Zhang, X.G., Xing, X.: Cacont: A ontology-based model for context modeling and reasoning. Applied Mechanics and Materials 347, 2304–2310 (2013)
17. Zimmermann, A., Lorenz, A., Oppermann, R.: An operational definition of context. In: Modeling and Using Context, Lecture Notes in Computer Science, vol. 4635, pp. 558–571. Springer Berlin Heidelberg (2007)

Experiences with an Approach to Abstract Handling of Content for Human Machine Interaction Applications

Richard Schmidt, Johannes Fonfara, Sven Hellbach and Hans-J. Böhme[*]

Artificial Intelligence Lab, University of Applied Sciences, Dresden, Germany
`{schmidtr;fonfara;hellbach;boehme}@htw-dresden.de`

Abstract. Current robotic software frameworks lack a mean to aid in the generation, validation and presentation of high quality content for user interaction. This paper introduces a new approach to extend a basic robotic software framework with a layer for content management. This layer has capabilities for controlling the content presentation subsystems already integrated. We introduce abstract *dialog acts* as a centerpiece for creating and handling robot behavior including user interactions. A simple file format is used to edit the dialog act structure and it allows the delegation of the dialog creation to domain experts within the desired field. We demonstrate that the creation of different sets of dialog acts allows the implementation of completely different use cases without requiring any changes to existing software components.

Keywords: dialog content · content creation · corpus building · human machine interaction

1 Introduction

Within recent years, the field of human machine interaction (HMI) has drawn more and more attention within the robotics community. Interactions with human users play a key role in numerous disciplines such as robotic guidance, entertainment, and ambient assisted living.

There are plenty of ways for robotic platforms to communicate with human users. The most common ones are to display text, images, and videos on a mounted screen, and output speech – either prerecorded or generated by a text-to-speech system. Touch screens and automated speech recognition systems along with dialog management systems receive and process the input from the user. A schematic overview can be seen in Fig. 1.

For researchers and developers working in this field, the following three problems usually arise:

[*] This work was supported by ESF grant number 100076162.

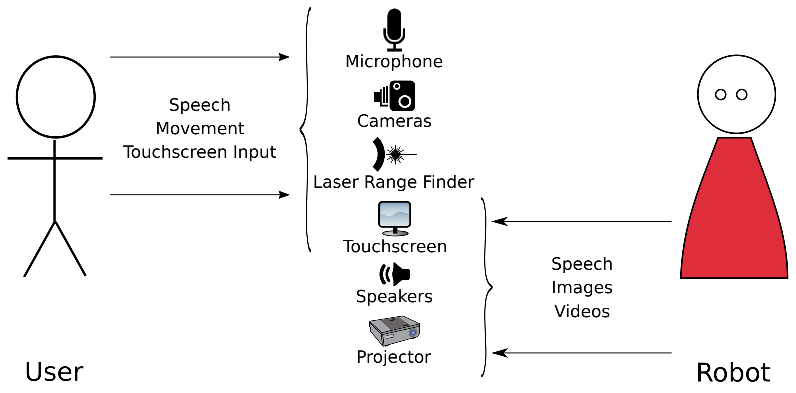


Fig. 1: Schematic overview with means of user communication for human machine interaction.

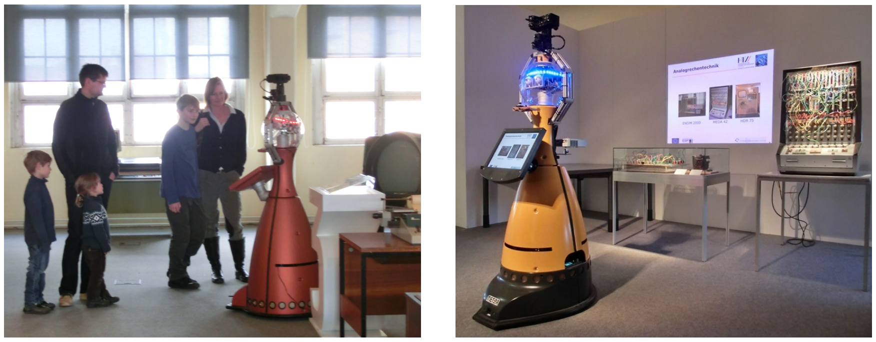
Missing Presentation Middleware In the robotics community several software packages like ROS [10], Player/Stage [7] and MIRA [5] aid the implementation process of real-world robotic applications. These frameworks are very helpful to abstract hardware access, interconnect software modules and develop algorithms even up to a behavioral level. Yet they all lack focus regarding the interaction with humans, as they were never designed specifically for this purpose.

Content Creation In real world applications, the *dialog content*[1] that the robot can present has to be gathered, edited, and evaluated. Unfortunately, the developers of a robotic application commonly do not have the expert knowledge to generate high quality dialog content for the robot's operational scenario. So *domain experts* should be enabled to author such content instead. Furthermore, the authored dialog content usually has to be tested and evaluated in real world scenarios, as this may reveal additional ways to improve the content.

Corpus Building Log data from previous deployments might be needed to create a data collection allowing further analysis, also known as a *corpus*. This corpus can be used to develop and tune speech recognition or dialog management systems and adapt them to the content. But as long as these systems are not yet capable of performing a user-satisfactory dialog autonomously, reliable data is hard to obtain.

In the following section, we formulate the requirements for a HMI-capable museum tour guide robot (see Fig. 2). In Sect. 3, we describe our proposed framework extension to fulfill the requirements. We continue with a discussion of how we applied our extension to multiple real world use cases in Sect. 4. Sect. 5 con-

[1] The term *dialog content* herein comprises everything that is used for user interaction such as speech, text, images, videos and even interactive applications like games.



(a) (b)

Fig. 2: (a) Tour guide robot leading a group. (b) Attached digital image projector is used to present content.

cludes this paper with an evaluation of our approach and gives an outlook for possible subsequent work.

1.1 Robotic Platform

For our experiments we used a Scitos G5 robot by MetraLabs GmbH[2], as shown in Fig. 2. Its anthropomorphic qualities, such as its life-sized proportions and a movable head, make this platform adequate for HMI applications. A sonar array, two laser range finders (front and back), microphones, a 360° camera array and a depth camera are the sensors on the platform. Speakers, a touchscreen and a digital video projector are the devices that allow presentation of information towards visitors.

1.2 Related Work

Several approaches for multimodal dialog management systems for robotic application like MuDiS [8] and the dialog system of the BIRON project [12] exist. Their goal is to enable a natural interaction with robot applications by interpreting input from different modalities, fusing the input and generating dialog output accordingly. Commonly not addressed are the aspects of dialog content authoring, evaluation and presentation that we focus on in this paper.

The Artificial Intelligence Markup Language (AIML) [13] is a markup language that serves as the knowledge base for HMI applications. It shares similarities with a markup language proposed in this paper, but lacks means of handling multimodal dialog content while being more complex. In [3,2] the Multimodal Interaction Markup Language (MIML) is introduced. The language abstracts

[2] `http://metralabs.com`

global tasks, means of interaction and low level modalities. MIML itself, beeing a language concept, does not solve the mentioned real world problems that we are going to address in this paper.

2 Requirements

Our goal is to use the robot as a tour guide in a museum. In this real world application, the robot has to inform and entertain visitors that were not trained for interaction with the device. Therefore, we think that spoken natural language is the best mean of interaction. The main reason why robots in public areas still lack complex dialog capabilities is that speaker-independent speech recognition is still a challenging task. Having this problem in conjunction with a dialog system still being in its development phase, a satisfactory spoken user interaction is not within short term reach. In order to still be able to gather dialog data for research and evaluate our already created subsystem under real world conditions, we decided to deploy a so called *Wizard of Oz* setup [11,9], in which a human operator remotely controls the application. This creates the illusion of an already completely operational system with spoken dialog interactions in a manner and quality that we aim to achieve eventually with a completely autonomous system.

For this Wizard of Oz extension to our existing platform the following main requirements were formulated:

Framework Integration The extension has to be implemented on top of an already employed robotic framework without requiring extensive modifications to the framework itself or existing subsystems. This enables the evaluation of these subsystems, for example navigation and people tracking, in a real – possibly crowded – environment.

Remote Operation A *remote operator* must be able to control certain high level aspects of the interaction and the robot behavior, for example triggering dialog reactions letting the robot navigate to waypoints. Therefore the operator has to take a remote location, where video and audio data are streamed from the robot's sensors via wireless connection.

Multimodal Dialog Presentation We intend to present dialog contents mainly by natural language outputs being generated by a text-to-speech system on the robot, together with a touch screen and a digital video projector presenting images, videos and text contents. The touch-capability of the screen should be used to allow browsing through a graphical user interface.

Content Creation In our scenario, the expert knowledge and media files about museum exhibits are not directly available to the developers. Therefore, the wish emerged to hand over certain aspects of the content authoring process to the domain experts of the exhibition. A template structure for all the content has to be created, easy enough to be filled by the experts without requiring background information about the software framework. The development of additional content authoring tools should be avoided for simplicity reasons.

Beside letting domain experts author the content, real world deployment sometimes requires the ability to adapt content without much effort, for example to react to unforeseen changes in the environment.

Contextual Dialogs It should be possible to provide a dialog text statement in different alternatives. This is necessary to adapt dialogs to the current operational context of the robot for an socially acceptable behavior. For example, groups should be addressed differently than a single person and facts should be explained more easily understandable for children.

Migration to Dialog System From the software developer point of view, the extension to our framework has to work independently of whether the dialog is directed by the remote operator or a dialog management system. A parallel deployment of a human operator and a dialog management system needs to be possible as well, which is desired for an iterative test-and-development cycle. Then more and more tasks of the human operator can be gradually taken over by an autonomous dialog management system.

Corpus Building A corpus of speech and interaction between robot and visitors is needed as the foundation to develop and train a dialog system as noted in [6]. The extension should aid in the process of building such a corpus.

3 Design and Implementation

We decided to base our dialog system on our *General-Robot* framework, whose design is heavily influenced by the actor model [1] and thus allows the concurrent processing of internal messages.

Depending on the design of the robot software framework in use, concurrent processing might not be necessary or even desired at the message passing and dispatching level. For an adaptation of our approach to different frameworks, a simple observer pattern, whereby messages are passed to observers, should be sufficient.

The General-Robot framework, maintains a set of *states*, *state containers* and *state processors*. Data objects representing messages, for example sensor data, are encapsulated into state objects. These encapsulated state objects are then enqueued into state containers which retain a certain constant number of states or alternatively all states within a certain time horizon. State processors can observe these containers, in which case they are notified about new states. We will discuss certain aspects of the design further in this section.

3.1 Dialog Acts

As mentioned in Sect. 1.1, our robot can present dialog content as uttered speech, on a touchscreen and as projected images or videos, of which speech is the most common and important mode. For natural language generation we use static text blocks. In our speech module, we use a third party text-to-speech system to transform text blocks on the robot directly into an audible signal. Whereas a system with prepared audio files would also be possible, we use this approach as it avoids the process of generating new audio files even after minor text changes.

Formally, a dialog is made up of a series of *dialog acts*. Every dialog act consists of one or more atomic system *commands* and has a unique label that also serves as a short description. When they are triggered by either the dialog system or the remote operator, the *dialog act dispatcher* sends the associated commands to the respective subsystems where they are processed accordingly (see Fig. 3). In order to formalize dialogs, we use a clear text markup language with a focus on human readability which is shown in Fig. 4. This allows to externalize the creation of the dialog text as mentioned in Sect. 2 and also circumvents the creation of additional tools for authoring.

The text blocks that the robot should utter are directly embedded into the dialog acts file. Although a stricter separation between structure and content – to which the text belongs – of a dialog act might appear desirable, we find that the convenience of being able to editing both in one common place is worth the structural breach and allows the required fast changes to text as mentioned in Sect. 2.

As shown with dialog act *OK* in Fig. 4, it is possible to offer several alternative texts for one statement. The dialog act dispatcher chooses randomly from the alternatives. This avoids a tedious listening experience to often repeated dialog acts like *YES* and *NO*.

To allow different text alternatives for different dialog contexts as required in Sect. 2, we added an extra layer of differentiation as shown on dialog act *WHERE_FROM* which is available in the alternations named *Text*, *TextGroup* and *TextFormally*. *Text* is the default and has to exist for all dialog acts where text utterance is desired. Other alternatives can be created and named freely. The dialog act dispatcher will choose the one preferred by the dialog management system.

3.2 Command Dispatching

A command represents a single task to be executed by the robot. Every command has a certain command type and may or may not carry arguments. There are command types for every aspect of our existing robotic platform that need to be controlled remotely in our museum scenario. An overview of the types is shown in Tab. 1. Within the framework, commands and arguments get encapsulated within a state.

Software components can instruct the dialog act dispatcher to trigger dialog acts by their label. The dispatcher then looks up in its in-memory representation of the dialog acts file and resolves the acts into commands. Submodules – which are also state processors – can listen to the dispatcher's commands state container in order to receive notifications when new commands arrive (see Fig. 3).

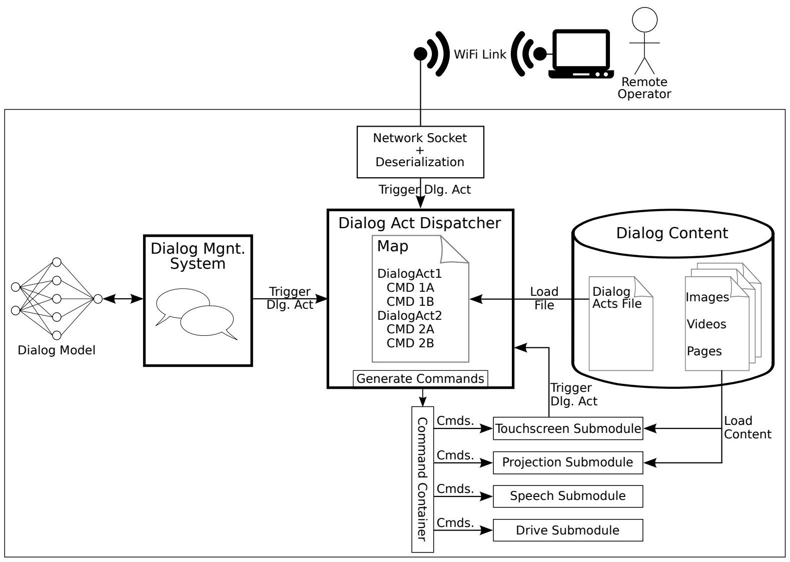


Fig. 3: Schematic overview of the usage of dialog acts in our system.

3.3 Media Content

We use the touchscreen and the projector to present visual dialog content to visitors. Similar to the speech submodule, the corresponding submodules are controlled by commands from the dialog act dispatcher. But here, the commands carry a file path to media files as argument. It is up to the submodules to render the files on the output devices appropriately.

The projection submodule takes care of finding projection regions and perspective correction which is further described in [4]. For touchscreen content, we decided to not only resort to plain images and videos, but also to use a HTML renderer. This allows the creation of interactive pages, through which users can browse by touch gestures. We extended the HTML render with the possibility

Table 1: Most commonly used commands for our robot applications.

Command	Argument	Submodule	Description
SAY	text	Speech	Let the robot speak the given text
PROJECT	file path	Projection	Let projector display image/video
DISPLAY_PAGE	file path	Touchscreen	Show HTML page on touchscreen
SET_WAYPOINT	coordinates	Drive	Set target coords. in environment map
DRIVE_START	none	Drive	Start/Continue drive to waypoint
DRIVE_STOP	none	Drive	Stop drive to waypoint

```
- Label: WELCOME
  Text:
        - Hello!
  Cmds:
        - DISPLAY_PAGE Welcome/Welcome.html

- Label: OK
  Text:
        - OK!
        - Great!
        - Splendid!

- Label: WHERE_FROM
  Text:
        - Where do you come from?
  TextGroup:
        - Where are you from?
  TextFormally:
        - May I mask from where you are?

- Label: RUN_VIDEO_EXHIBIT
  Text:
        - Let me show you a video about this exhibit.
  Cmds:
        - PROJECT Exhibit/video.mpg
```

Fig. 4: Example listing of a dialog acts file consisting of four dialog acts.

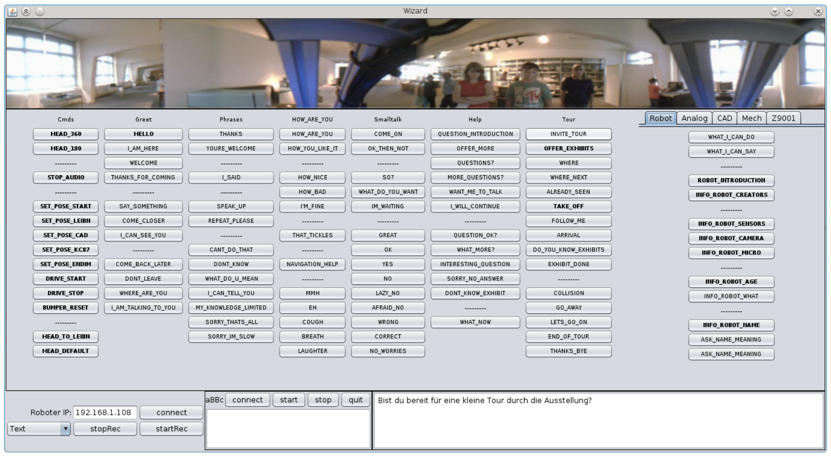
to trigger dialog acts, which enables more complex reactions to touch gestures, for example speech output.

It should be noted, that the capabilities of the projection and touchscreen submodules are not limited to preexisting media files, as we can display everything that could be rendered to a pixel buffer. Therefore, the modules allow the presentation of runtime generated media content – for example interactive games – which we plan to integrate in the future.

3.4 Client/Server Communication

We developed a remote control client that connects to a server component on the robot over the network. The server itself is an extension to an existing robot software stack, acquiring live camera images and audio from the sensors and streaming them over the network to the client, which plays them back to the remote operator.

On startup, the client loads and parses a dialog content file. The remote operator can trigger each dialog act by clicking the corresponding button. A screen shot can be seen in Fig. 5. Then the client forwards the triggered dialog act over network to the dialog act dispatcher on the robot. There, the dialog acts get resolved into commands which are sent to the listening subsystems. It should be noted that is does not matter for the subsystems where the commands come from, which allows a seamless migration between remote and autonomous dialog operation as required in Sect. 2.



Roboter IP: 192.168.1.108

Bist du bereit für eine kleine Tour durch die Ausstellung?

Fig. 5: Screen shot of the remote operation software. On the upper side, the camera image can bee seen. Below are the buttons to trigger dialog acts.

4 Discussion

In this section, we will discuss the usage of our dialog content extension in real world applications, regarding the tour guide use case and the building of a corpus for dialog management systems. Over time other use cases emerged, that we wanted to realize with our existing robot hard- and software. Our extension proved to be quite flexible and could also handle these new use cases with little to no modification to the existing setup. We will also discuss two additional use cases in the Sect. 4.3 and Sect. 4.4.

4.1 Tour Guide

We deploy our system in an exhibition of vintage computer hardware. As developers do not have access to all the resources and knowledge of the museum staff, a major part of the content authoring for the tour guide was done by the staff. To ease the process to them, we provided documentation and a template dialog acts structure that could be used as a building block for different exhibits.

We used the remote operation capability to give personalized tours to single visitors or smaller groups. In this setup, the remote workstation is located hidden from the visitors and connected to the robot via Wireless LAN. Our approach proved very suited to provide entertaining tours to visitors and gather real live data of dialog interactions. Also, we were able to evaluate already existing sub modules, like people tracking and path planning, in a candid real world environment.

Remote operating the robot has shown to be a complex task, as the operator has to choose an appropriate dialog option from a wide range of possibilities in a short period of time [9].

4.2 Corpus Building

To build a corpus, we use the logging capabilities of the robotic framework to record audio data and camera video streams. The dialog interactions triggered by the remote operator get recorded by a simple extension to the framework's logging capabilities. Due to lacking reliability of current speech recognition systems in our operational scenarios, it is unavoidable to resort to manual annotation of the recorded sensor data to comprehensively record the dialog interaction from the visitors towards the robot.

During several sessions we gathered a corpus of about six hours audio and video material, showing genuine interactions between robot and visitors. The corpus consists of 133 dialogs involving 378 test subjects. We annotated the corpus distinguishing about 30 different dialog situations, in which we transcribed all spoken utterances from the visitors. Additionally, major movement actions, the location and attention of the user were labeled.

The corpus analysis was very helpful in many ways. Firstly, it gave us a general feeling for the type of behavior to expect from visitors interacting with our system. We were able to designate four main classes of interaction behavior: *interested*, *chatting*, *passively interested*, and *not interested*. Surprisingly, most people reveal a chatting behavior, which included a lot of small talk before the interest shifts towards the museum exhibits.

Secondly, we computed various statistics of user behavior which were used to train a user simulator. The simulator model used is described in [6]. Using this simulator, we were able to reproduce versatile interactions of the tour guide scenario and train a dialog management system.

Thirdly, having all the user utterances annotated, we tested several algorithms for text classification. This allowed us to build a natural language processing module that can make use of a large-vocabulary speech recognizer to recognize broad range of speech inputs.

4.3 Info Terminal

We deployed a robot as an advanced info terminal at a variety of venues. There, the robot ought to provide basic information, such as schedules and maps of the location, to users using the touch screen. Remote dialog operation was not used.

This use case has been implemented without modifications to our software, only by creating dialog content. We had to write a dialog acts file and appropriate browsable pages for the touchscreen. The ability to trigger dialog acts from page elements (see Sect. 3.3) like buttons, allowed not only to let users progress between pages, but also to let the robot verbally utter descriptions of the pages.

We tested our info terminal application on various occasions and it behaved as expected. But to further improve this use case, a simple dialog system could be added, that employs data from our people tracker to allow the robot to react to nearby persons and automatically advertise itself as a source of information.

4.4 Poster Presenter

We also wanted to use our robot in an entertaining way as a presenter for posters at exhibitions, workshops and conferences. To present a poster, the robot highlights a certain area on a poster pinned to a wall using its projector, while uttering speech towards its audience. The touchscreen is used to show supplementary information. After a poster area has been explained, the robot proceeds to the next one.

For this setup, the projection submodule is used to simply project black images containing white patches matching the areas of the poster. Every poster section is represented by a dialog act. The dialog progress is controllable either remotely by an remote operator or automatically. For the automatic progression, we use JavaScript-Timers in the touchscreen HTML page to trigger the following dialog act.

Both variations were tested successfully on various occasions. However, the additional flexibility that currently only the operator can warrant, allows for a sometimes desirable variation from an otherwise static flow of information towards the user. Further information about the projection setup of this application are presented in [4].

5 Conclusion

In this paper, we proposed a concept to deal with the problem of dialog content handling in robotic applications. The described software stack did not only regard the organization of dialog content and its presentation, but also the authoring phase. By involving domain experts with the required domain knowledge in the authoring phase, the dialog content can become more useful and achieves a higher quality. In the end, this will increase the usefulness of the resulting robotic application to the user and might improve his impression of how interesting and pleasant the interaction with the robot turns out to be.

The presented approach proved highly versatile and flexible, as it allowed the realization of different applications by merely authoring additional content. Uttered texts are the main focus, but also on-screen content, images, and videos are considered.

By being able to remote control the dialog flow, the approach allows to build a corpus of real world dialog data needed for the further improvement of dialog systems. The migration to a completely autonomous dialog system can directly be done utilizing the existing implementation.

Further Work The extension of the framework towards our application is fairly complete. However, there is still room for improvement. As we are employing predefined text blocks as the foundation for spoken utterances, an extension towards less static representations of text might be desirable for further iterations of our dialog system.

In regards to the requirements noted in Sect. 2, the content creation phase, and especially the externalization aspect, could be optimized further. Even if we employed a very simple markup language to hold dialog actions, a special written editing software could still prove more user friendly. Such a software could easily be integrated into the current workflow.

References

1. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
2. Araki, M.: Proposal of a markup language for multimodal semantic interaction. In: Proceedings of the 2007 Workshop on Multimodal Interfaces in Semantic Interaction. pp. 58–62 (2007)
3. Araki, M., Tachibana, K.: Multimodal dialog description language for rapid system development. In: Proceedings of the 7th SIGdial Workshop on Discourse and Dialogue. pp. 109–116 (2006)
4. Donner, M., Himstedt, M., Hellbach, S., Böhme, H.J.: Awakening history: Preparing a museum tour guide robot for augmenting exhibits. In: Proceedings of the European Conference on Mobile Robots (ECMR). pp. 337–342 (2013)
5. Einhorn, E., Langner, T., Stricker, R., Martin, C., Gross, H.M.: Mira - middleware for robotic applications. In: Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS). pp. 2591–2598 (2012)
6. Fonfara, J., Hellbach, S., Böhme, H.J.: Learning Dialog Management for a Tour Guide Robot using Museum Visitor Simulation. In: Proceedings of the Workshop - New Challenges in Neural Computation 2012 (NC2). pp. 61–68 (2013)
7. Gerkey, B.P., Vaughan, R.T., Howard, A.: The player/stage project: Tools for multi-robot and distributed sensor systems. In: Proceedings of the 11th International Conference on Advanced Robotics. pp. 317–323 (2003)
8. Giuliani, M., Kaßecker, M., Schwärzler, S., Bannat, E., Gast, J., Wallhoff, F., Mayer, C., Wimmer, M., Wendt, C., Schmidt, S.: Mudis - a multimodal dialogue system for human-robot interaction. In: Proc. 1st Intern. Workshop on Cognition for Technical Systems (2008)
9. Poschmann, P., Donner, M., Bahrmann, F., Rudolph, M., Fonfara, J., Hellbach, S., Böhme, H.J.: Wizard of Oz revisited: Researching on a tour guide robot while being faced with the public. In: 21th IEEE Int. Symposium on Robot and Human Interactive Communication (RO-MAN). pp. 701–706 (2012)
10. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
11. Shiomi, M., Kanda, T., Koizumi, S., Ishiguro, H., Hagita, N.: Group attention control for communication robots with wizard of oz approach. In: Proceedings of Conference on Human-Robot Interaction (HRI). pp. 121–128 (2007)
12. Shuyin, I.T., Toptsis, I., Li, S., Wrede, B., Fink, G.A.: A multi-modal dialog system for a mobile robot. In: Proc. Int. Conf. on Spoken Language Processing. pp. 273–276 (2004)
13. Wallace, R.S.: The anatomy of a.l.i.c.e. In: Epstein, R., Roberts, G., Beber, G. (eds.) Parsing the Turing Test, pp. 181–210. Springer Netherlands (2009)

Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems

Jan Oliver Ringert[1,2*], Alexander Roth[1], Bernhard Rumpe[1], Andreas Wortmann[1]

[1] Software Engineering
RWTH Aachen University
http://www.se-rwth.de/
[2] School of Computer Science
Tel Aviv University
http://www.cs.tau.ac.il/

Abstract. Engineering software for robotics applications requires multi-domain and application-specific solutions. Model-driven engineering and modeling language integration provide means for developing specialized, yet reusable models of robotics software architectures. Code generators transform these platform independent models into executable code specific to robotic platforms. Generative software engineering for multi-domain applications requires not only the integration of modeling languages but also the integration of validation mechanisms and code generators. In this paper we sketch a conceptual model for code generator composition and show an instantiation of this model in the MontiArc-Automaton framework. MontiArcAutomaton allows modeling software architectures as component and connector models with different component behavior modeling languages. Effective means for code generator integration are a necessity for the post hoc integration of application-specific languages in model-based robotics software engineering.

1 Introduction

Software engineering for robotic systems is inherently complex due to the heterogeneity of the systems and their challenges from various domains (e.g., navigation, sensor fusion, manipulation). Thus, robotics software is usually developed by teams of domain experts with different views and understanding of the systems functionality. This leads to hardly reusable software limited to specific platforms [8, 17]. To enable the reuse of functionality and subsystems, the structuring and composition mechanisms of component-based software engineering have been applied to robotics software [4, 12]. These approaches are mainly based on the exchange of source code components and thus tied to specific platforms and general-purpose programming languages (GPL). The application of

* J. O. Ringert acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research.

GPLs often does not reflect the problems from heterogeneous domains faced in the development of robotics systems.

Model-driven engineering (MDE) is an approach to reduce the *conceptual gap* [5] between problem domains and software engineering. Models allow domain-specific software descriptions reflecting the heterogeneity of the developed system and its concerns. In combination with powerful code generators models may serve as primary development artifacts which increases the software's comprehensibility and reuse on different platforms.

We have combined MDE and software language engineering based approaches with concepts from generative software development in a versatile framework for robotics applications development. MontiArcAutomaton [14,15] is an extensible framework that allows to model robotics applications as hierarchically composable components with well-defined interfaces that embed problem specific modeling languages for component behavior. MontiArcAutomaton comprises powerful code generation facilities for the transformation of models into executable code for various robotics target platforms.

Language integration in MontiArcAutomaton is enabled by the MontiCore domain-specific language workbench [10]. MontiCore provides comprehensive language composition mechanisms supported by its symbol table and code generation frameworks [16,21]. We have presented the language composition mechanisms used by MontiArcAutomaton in [11].

The easy integration of modeling languages demands for integration mechanisms of corresponding code generators. Challenges are the coordination of multiple code generators each responsible for specific models or parts of models. This includes the selection of code generators supporting a common target platform, to handle language restrictions a code generator might impose, and to propagate necessary *generation context information* between generators. Integrating code generators should require no modification of the participating generators.

In this paper we sketch a conceptual model for code generator composition and show its instantiation in the MontiArcAutomaton framework. We introduce MontiArcAutomaton in Sect. 2 and state and illustrate the problem of code generator composition in Sect. 3. Section 4 describes our solution and Sect. 5 describes an implementation in MontiArcAutomaton. We discuss related work in Sect. 6 and future work in Sect. 7. Section 8 concludes this contribution.

2 MontiArcAutomaton

MontiArcAutomaton [14,15] is an extensible modeling language and framework for the generative model-driven engineering of robotics applications. The modeling language MontiArcAutomaton is a component and connector (C&C) architecture description language (ADL) [19] which extends the ADL MontiArc [7] with component behavior modeling. The logical architecture of robotics applications is described as the hierarchical composition of components that encapsulate the system's functionality. Components are either atomic or composed: atomic components define behavior via an embedded behavior modeling language or a component implementation in a general-purpose programming language. The behavior of composed components emerges from the subcomponents and their interaction. Components interact by sending messages via directed connectors that connect typed input and output ports of components. Types of ports are either defined via class diagrams or Java classes. Communication in MontiArcAutomaton is based on the Focus [3] framework for interactive distributed systems and supports different timing paradigms.

The concept of encapsulation from C&C ADLs allows not only a logically distributed development and a physically distributed computation model but also the composition of component behaviors independent of their behavior description. MontiArcAutomaton exploits the C&C encapsulation mechanism and allows the embedding of arbitrary modeling languages into components for providing the most suitable behavior description language per component.

We have developed MontiArcAutomaton using the domain-specific language workbench MontiCore [10] and its language integration mechanisms. The concrete and abstract syntax of a textual MontiCore modeling language is defined in an extended context free grammar format. From these grammars, MontiCore generates infrastructure to parse models of this language into their abstract syntax trees (ASTs). Checks of the well-formedness of models of a language, called *context conditions*, are implemented in Java [21]. MontiCore languages are textual modeling languages. An integration with the Eclipse Modeling Framework allows also the development of graphical editors for editing MontiArcAutomaton models.[3]

MontiCore supports language embedding, language extension, and language aggregation [11,21] to compose new languages from existing ones. These modular language composition mechanisms are supported by a sophisticated symbol table framework that enables the definition and adaptation of language symbols for integrating information and checking context conditions rules across embedded and imported models. MontiCore allows the easy development of code generators using the FreeMarker[4] template engine to process abstract syntax trees and code templates written in a target language [13, 16].

In previous works we have developed the MontiArcAutomaton modeling language with embedded I/O$^\omega$ automata and I/O tables [15]. Various code generators allow the deployment of MontiArcAutomaton models to different robotics platforms [14, 15]. With the integration of additional languages to model component behavior the post hoc composition of code generators has become a prevalent challenge.

3 Problem Statement and Example

MontiArcAutomaton allows to embed *application-specific* behavior modeling languages into components to facilitate the development of flexible, reusable, yet

[3] Video of an editor for synchronous graphical and textual editing of MontiArcAutomaton models: `http://www.monticore.de/robotics/`

[4] Website of the FreeMarker Java template engine: `http://freemarker.org/`

specific robotics applications. While the ability to use specific behavior modeling languages allows to develop specific applications, the encapsulation of models in components with well-defined and stable interfaces allows to modify component internals easily, e.g., to replace the specific behavior modeling language, while retaining a stable architecture.

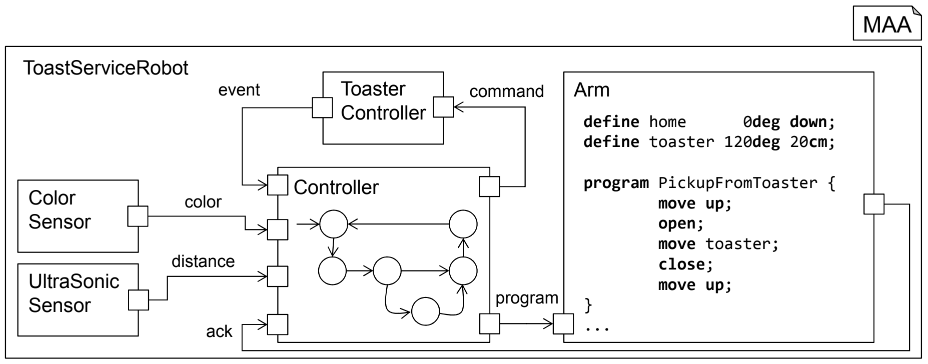
Engineering C&C applications with the flexibility of arbitrary embedded behavior modeling languages demands for approaches to generate code from heterogeneous models. As languages and code generators can be integrated into MontiArcAutomaton post hoc, code generators have to be composable to allow black-box integration. Each composable code generator produces only parts of the overall generated software system. A framework to support code generator composition has to provide a mechanism to configure C&C applications with different code generators. Realizing composition of code generators requires support for code generator reuse, the ability to handle code generators that are agnostic of any component structure specifics (e.g., how port or connectors work), and dependency management between different code generators.

3.1 Example

A software engineer is responsible for the development of a controller for a robotic arm. The robot assists a physically disabled person in a kitchen environment to operate a toaster. The robot is supposed to place bread in a toaster, operate the toaster, and deliver the toast to a nearby plate. The software engineer models the architecture and controller behavior platform independently using MontiArcAutomaton with embedded I/O$^\omega$ automata and RobotArm (RA) programs. The latter describe motion of the arm in terms of defined locations and gripper commands.[5] The engineer embeds the existing language RA into the MontiArcAutomaton framework using the language integration mechanisms of MontiCore. The software architecture of the robot is depicted in Fig. 1. The component `Controller` receives distances and toast color from attached sensors. The I/O$^\omega$ automaton modeling the behavior of `Controller` translates these inputs into commands for the `ToasterController`, which starts and stops the actual toaster, and the component `ArmController`, which actuates the robotic arm to pick up and deliver toast. The behavior of component `ArmController` is modeled as a set of RA programs.

To generate executable code from the architecture, the software engineer has to provide a code generator for the embedded RA language which translates RA commands into code for the target platform. This code generator can be selected from a library of existing code generators or newly developed. Finally, the generator has to be integrated into the framework, such that it is executed whenever a component with RA programs is processed.

[5] A video of the robotic arm: `http://www.monticore.de/robotics`



```
define home      0deg down;
define toaster 120deg 20cm;

program PickupFromToaster {
        move up;
        open;
        move toaster;
        close;
        move up;
}
...
```

Fig. 1: Architecture of the `ToastServiceRobot` with embedded RobotArm programs.

4 Code Generator Composition

In this section we propose an approach to code generator composition on a conceptual level. First, we describe code generator interfaces that support generator composition as motivated in Sect. 3. Second, we sketch the process of code generator composition and execution of the composed generators using information from code generator interfaces.

To achieve generator composition, each code generator explicates all information necessary within an interface. This interface is used during code generator orchestration to configure and execute the code generator. Definition 1 lists the elements of a code generator interface.

Definition 1 (Code Generator Interface). A code generator interface contains the following elements:

1. Input language: The language or language fragments the generator processes.
2. Input language constraints: A generator may restrict the processable models via generator-specific context conditions.
3. Output representation: The output representation states the language and format of the output.
4. Execution information: Defines how a generator is executed.
5. Artifact dependencies: A generator may produce code that depends on external libraries, runtimes, or code produced by another code generator. Such artifact dependencies have to be explicitly stated in order to satisfy dependencies of generated artifacts.
6. Generation context information: Additional information provided or required at generation time.

Multiple generators are composed to generate code for models of the software of a robotic system. The composition of code generators is described in an *application configuration* model which contains a selection of all code generators involved. It may also contain a configuration of generation context information

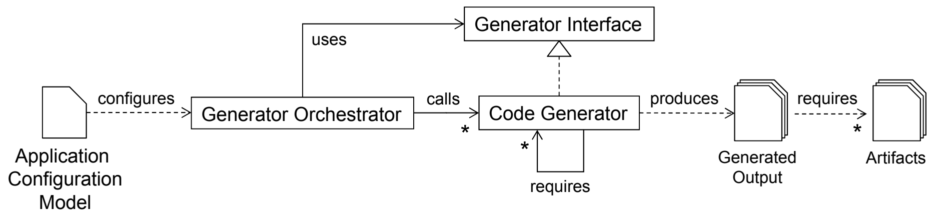


Fig. 2: Overview of code generator composition with generator interfaces.

for code generators. Composing generators according to a configuration model requires the orchestration of all selected code generators. Such an orchestration requires (a) to check that all required information is provided and (b) to compute an execution order of the code generators.

If for each code generator all required generation context information is provided by the selected code generators and an execution order can be computed, then the code generator composition can be performed. However, the execution order of the code generators is influenced by the dependencies described by the generation context information. There are two types of dependencies. First, a code generator may require generation context information from another code generator. Second, a code generator may use the output of another code generator. Both types of dependencies imply that the code generator providing required information or output is executed first. However, in some cases it is possible that an execution order cannot be computed. In this case the code generators cannot be composed.

Our concept of code generator composition is presented in Fig. 2. An application model configures a generator orchestrator. The generator orchestrator uses the generator interface of each code generator to check for dependencies and computes an execution order. Finally, the generator orchestrator calls each code generator according to the computed execution order.

5 Realization in MontiArcAutomaton

The MontiArcAutomaton implementation of the conceptual model presented above comprises an implementation of generator interfaces, which is facilitated by a configuration language that generates interface implementations, an application configuration to declare compositions of code generators, and an orchestrator performing the composition.

5.1 Generator Interfaces in MontiArcAutomaton

Based on the concrete requirements for code generators in MontiArcAutomaton we refine the code generator interfaces defined in Def. 1. The C&C nature of MontiArcAutomaton, suggests separate interfaces for component generators and component behavior generators. As MontiArcAutomaton relies on factories for

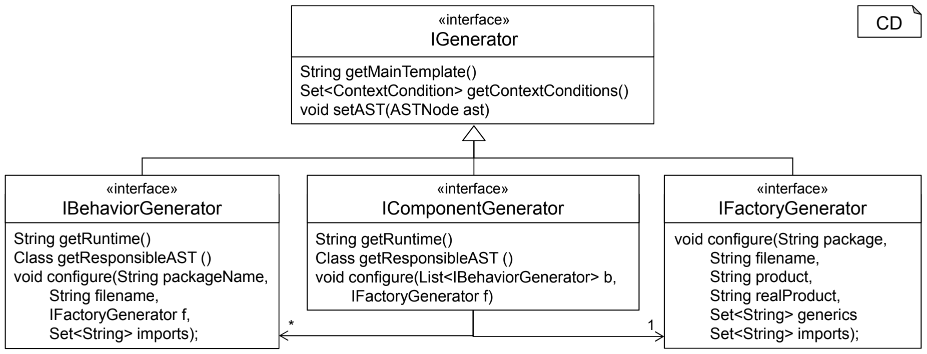


Fig. 3: Generator interface hierarchy of MontiArcAutomaton.

component and component behavior instantiation, factory generators are modeled as well. Component generators process the MontiArcAutomaton language and behavior generators process the respective embedded behavior modeling language (Def. 1, Item 1) possibly restricted by additional context conditions (Def. 1, Item 2). The classification in three generator kinds determines the output format of the generators (Def. 1, Item 3).

Generator execution information is provided by the generators in terms of the main template which the MontiCore code generation framework processes (Def. 1, Item 4). This template may call other templates and call Java code for complex calculations. All generators generate code conforming to a runtime environment they depend on (Def. 1, Item 5). The runtime environment determines, e.g., the scheduling of components. Generators in MontiArcAutomaton do not explicate further artifact dependencies as MontiArcAutomaton utilizes the delegator pattern [6] to integrate accordingly generated behavior implementations. Generation context information (Def. 1, Item 6) is provided to the generators at runtime and contains e.g., the AST of the processed model.

An overview of the concrete generator interfaces implemented for MontiArcAutomaton is displayed in Fig. 3. Every generator usable with MontiArcAutomaton implements an interface extending `IGenerator`. Thus, each generator can be parametrized with an AST node and provides at least its main template and its context conditions to the infrastructure. Generators for components and component behavior implement the interfaces `IComponentGenerator` and `IBehaviorGenerator` respectively. These interfaces explicate which AST types they can process.

Additionally, all generator interfaces define a method to `configure()` which is interface specific and defines the generation context information required. Generators for component behavior, e.g., expect to receive the package name of the containing component, the name of the artifact to be created, a factory generator, and the imported compilation units. The latter is required as embedding behavior into components produces integrated artifacts without distinction between the imports of the component and the imports of the behavior.

```
                                              GeneratorConfiguration
1 generator RobotArmPython {
2     interface generators.IBehaviorGenerator;
3     template robotarm.Main;
4     ast robotarm.ASTRobotArmProgram;
5     runtime runtimes.pythontimesync;
6 }
```

Listing 1: The generator configuration for the RobotArm generator describes that it implements the interface `IBehaviorGenerator` and provides static information.

5.2 Modeling Generator Interfaces

To facilitate the creation of code generator interfaces we have developed a modeling language for generator interfaces. Each code generator used with MontiArcAutomaton models how it is executed, which AST it processes, and which interface it implements in a single *generator configuration* model per generator. Listing 1 shows the model of the RA generator from the example in Sect. 3.1. This model describes that the generator implements the interface `IBehavior-Generator` and provides information accessable via this interface. The MontiArcAutomaton toolchain transforms these models into actual implementations implementing the interfaces.

The concrete implementation of the interface `IBehaviorGenerator` for the RobotArm generator from the example given in Lst. 1 provides implementations for all methods of `IGenerator` and `IBehaviorGenerator` and returns the static generator information from the model where applicable (e.g., `getRe-sponsibleAST()` returns an instance of the type specified behind `ast` in l. 4 of Lst. 1). The MontiArcAutomaton orchestrator can refer to these implementations via the implemented interfaces and compose generators as necessary.

5.3 Application Configuration and Generator Execution

Given a set of generators for component structure, behavior, and factories, an application has to specify which of these are to be used. This is modeled as the *application configuration* model. Listing 2 shows the application configuration for the toaster robot application. The model references a single component generator (l. 2), a single factory generator (l. 3), and two behavior generators - one for RA programs and one for I/O$^\omega$ automata (l. 4). An application configuration references at least a component structure generator and may reference additional behavior and factory generators.

Code generation in MontiArcAutomaton starts with the orchestrator processing the application configuration and loading the configuration of the referenced generators. As the order of generator execution is implicitly given by the C&C nature of MontiArcAutomaton, first the referenced behavior generators and the

```
                                          ApplicationConfiguration
application ToasterRobotApplication {
    componentgenerator ComponentsPython;
    factorygenerator FactoryPython;
    behaviorgenerators RobotArmPython, IOAutomatonPython;
}
```

Listing 2: Application configuration model for the toaster robot application using the RA generator for component behavior.

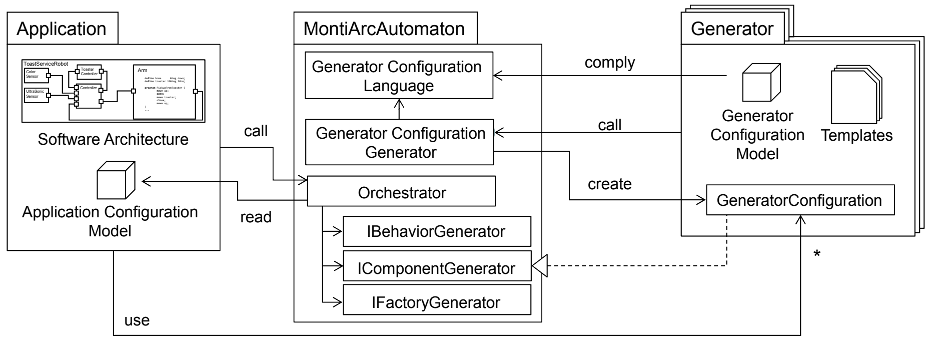


Fig. 4: Relations between applications using generators, the interfaces provided by MontiArcAutomaton, and the orchestrator performing the generator composition.

referenced factory generator are instantiated. Parametrized with these, the referenced component generator is instantiated. Afterwards, the orchestrator calls the main template of the component generator. The component generator traverses the AST of the architecture and thus also visits component behavior AST nodes. For each behavior AST node the responsible generator is configured with current AST generation context information and its main template is called with the AST node of the embedded behavior language.

Figure 4 shows the resulting relations: Applications consist of a software architecture, and an application configuration model. The application configuration model references the component, behavior, and factory generators required to build the software architecture of the project. To be processable by the orchestrator, referenced generators implement the appropriate generator interfaces.

6 Related Work

The presented approach for code generator composition is based on explicit generator interfaces, code generator orchestration, and application configuration. This approach is a first step towards a comprehensive approach for code generator composition and is closely related to modular code generator design.

The GenVoca model is an approach to build software systems generators based on composing object-oriented layers [1, 2]. Different layers can use control blocks to exchange information. In contrast to this approach, we do not focus on a layered architecture of a code generator but an infrastructure for code generators composition.

The application building center is a multi-purpose modular framework for modeling software systems [18]. Genesys is an extension that allows to develop service-oriented code generators [9]. Each code generator represents a service that can be composed with other services. Information exchange is managed by using shared memory communication. Our presented approach is similar if we consider code generators to be services with interfaces. However, our approach introduces a broader generator interface to regard input language, output representation, input language constraints, execution information, artifact dependencies, and generation context information. This information is used to manage the execution and composition of the code generators.

Code generator composition using aspect-orientation at the artifact level has been described in [22]. The authors assume that a code generator produces operationally complete code fragments that are merged by a code fragment weaver. Additionally, in feature-oriented model-driven development (FO-MDD), multiple code generators are used to produce a software product line [20]. Composition of code is achieved after code generation by manually writing glue code. In contrast, we do not consider manual artifact composition but focus on an infrastructure to compose code generators. We nevertheless consider composing generated artifacts relevant for reusing code generators and will address this topic in future work.

7 Discussion and Future Work

We have presented a conceptual approach for composition of code generators based on the notion of generator interfaces. The ideas are implemented within the MontiArcAutomaton toolchain to enable post hoc embedding and use of new component behavior modeling languages. To broaden its applicability this approach requires future work on syntax, methods, and technical solutions.

Composition of arbitrary code generators without assumptions on their actual integration is harder to realize than for C&C ADLs. In general, generator composition demands a more expressive composition configuration than the application configuration presented above. For instance, the orchestration of the code generation process may require a code generator to be executed multiple times for every input model or to fill extension points provided by another generator under certain conditions. Moreover, execution of a code generator may not be triggered by a model type but by selecting a code generator for a particular set of input models. A generic model to configure an application has to express such process information and constraints. Thus, future research will look into modeling these aspects.

The generator composition illustrated above assumes that the orchestration of generators reflects the language embedding for component behavior. Other language integration mechanisms, such as language aggregation or language inheritance [11] will require a more complex orchestration. The generator for an inheriting language might, for example, require the generator for the inherited language to be executed first, such that the latter only generates additional artifacts for the model elements introduced by the inheriting language. Future work will therefore examine the notion of generator extension points as well.

Finally, modeling language composition mechanisms have lead to language reuse and language libraries. We hope to gain similar libraries and advantages from facilitating code generator composition.

8 Conclusion

We have motivated the need for generator composition in robotics and sketched a concept for code generator composition. This concept is based on explicit code generator interfaces and configuration models. The interfaces enable code generators to define information required for composition. A code generator orchestrator composes and executes the code generators. We have illustrated our implementation for the C&C modeling language family MontiArcAutomaton. Although the implementation relies on various assumptions implied by the language workbench MontiCore and the C&C nature of MontiArcAutomaton, we belief that these translate well into other contexts. There are however open issues in arbitrary generator composition and we have identified possible extensions of generator interfaces and generator orchestrators to be applied in more complex scenarios.

References

1. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. 1(4), 355–398 (Oct 1992)
2. Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B., Sirkin, M.: The genvoca model of software-system generators. IEEE Softw. 11(5), 89–94 (Sep 1994)
3. Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg (2001)
4. Brugali, D., Brooks, A., Cowley, A., Côté, C., Domínguez-Brito, A., Létourneau, D., Michaud, F., Schlegel, C.: Trends in Component-Based Robotics. In: Brugali, D. (ed.) Software Engineering for Experimental Robotics, Springer Tracts in Advanced Robotics, vol. 30, chap. 8, pp. 135–142. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
5. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. Future of Software Engineering (FOSE '07) (2), 37–54 (May 2007)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)

7. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Tech. Rep. AIB-2012-03, RWTH Aachen (february 2012)
8. Hägele, M., Blümlein, N., Kleine, O.: Wirtschaftlichkeitsanalysen neuartiger Servicerobotik- Anwendungen und ihre Bedeutung für die Robotik-Entwicklung. Tech. rep., BMBF (2011), `http://www.ipa.fraunhofer.de/`
9. Jörges, S.: Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach. LNCS sublibrary: Programming and software engineering, Springer Berlin Heidelberg (2013)
10. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. STTT 12(5), 353–372 (2010)
11. Look, M., Perez, A.N., Ringert, J.O., Rumpe, B., Wortmann, A.: Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In: Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC). Miami, Florida, USA (2013)
12. Niemueller, T., Ferrein, A., Beck, D., Lakemeyer, G.: Design Principles of the Component-Based Robot Software Framework Fawkes, Lecture Notes in Computer Science, vol. 6472, chap. NFB+10, pp. 300–311. Springer, Darmstadt, Germany (2010)
13. Ringert, J.O., Rumpe, B., Wortmann, A.: A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. In: Giese, H., Huhn, M., Philipps, J., Schätz, B. (eds.) Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme. pp. 30–43 (2013)
14. Ringert, J.O., Rumpe, B., Wortmann, A.: From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In: Software Engineering 2013 Workshop Proceedings. p. to appear (2013)
15. Ringert, J.O., Rumpe, B., Wortmann, A.: MontiArcAutomaton : Modeling Architecture and Behavior of Robotic Systems. In: Workshops and Tutorials Proceedings of the International Conference on Robotics and Automation (ICRA). Karlsruhe, Germany (2013)
16. Schindler, M.: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band 11, Shaker Verlag (2012)
17. Schlegel, C., Steck, A., Lotz, A.: Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model. In: Chugo, D., Yokota, S. (eds.) Introduction to Modern Robotics. iConcept Press (2011)
18. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-driven development with the jabc. In: Bin, E., Ziv, A., Ur, S. (eds.) Hardware and Software, Verification and Testing, Lecture Notes in Computer Science, vol. 4383, pp. 92–108. Springer Berlin Heidelberg (2007)
19. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley, 1st edn. (2009)
20. Trujillo, S., Batory, D., Diaz, O.: Feature oriented model driven development: A case study for portlets. In: Proceedings of the 29th International Conference on Software Engineering. pp. 44–53. ICSE '07, IEEE Computer Society, Washington, DC, USA (2007)
21. Völkel, S.: Kompositionale Entwicklung domänenspezifischer Sprachen. Aachener Informatik-Berichte, Software Engineering Band 9. 2011, Shaker Verlag (2011)
22. Zschaler, S., Rashid, A.: Towards modular code generators using symmetric language-aware aspects. In: Proceedings of the 1st International Workshop on Free Composition. pp. 6:1–6:5. FREECO '11, ACM, New York, NY, USA (2011)

Empirical Study of Planning and Execution for Large Teams of Robots

Daniel Saur, Tareq Razaul Haque, and Kurt Geihs

Distributed Systems Group, University of Kassel 34121, Germany
{saur,haque,geihs}@vs.uni-kassel.de

Abstract. Large teams of robots can substantially increase the effectiveness of planning by acting as coordinated team. Our focus is on the planning of activities of a team of autonomous, mobile robots by distributed planning coordinated by one robot. With arising number of agents the communication increases rapidly. Our goal is to minimize communication much as possible. Modeling needs to be combined with planning to describe complex activities in intuitive way. The main contribution of the paper is the optimization of the planning process while using every agent as a planning resource and aiming at low communication needs. We evaluated our distributed planning for teams of up to 75 agents in the transport domain of the International Planning Competition[1] (IPC). We optimized the planning process compared to state-of-the-art approaches (last winners of IPC in the transportation domain) by up to 23%.

Keywords: Autonomous robots, Mobile robots, Distributed planning

1 Introduction

Recent advances in autonomous robot technology have opened up great opportunities in an exciting new application potential. Autonomous mobile robots can act individually while using an intuitive goal description for the team. This creates an enormous potential for innovative applications that intelligently support environmental monitoring, disaster management, logistics operations, and many other practices.

However, several challenging research questions have to be solved before we can harvest the benefits of such kinds of multi-agent systems. Increasing the number of agents also enourmously increases the overhead for maintenance, modeling, and testing. In our work we concentrate on planning for large teams with a high number of agents. The planning process calculates a plan, which describes the activities of all agents from a global perspective. This plan is divided into tasks, which denote the activity of a specific agent within the plan. The plan format is defined with ALICA (A language for interactive cooperative Agents)[15], which offers support for task allocation and coordination. Finally, we use every agent

[1] http://ipc.icaps-conference.org/

as a planning resource. The goal is to optimize the search time in distributing different seeds for the search tree, where we expect an acceleration of the search. We report on the results of an ongoing research project where we are developing a framework which supports distributed planning for a team of robots. We accelerated the search time by as much as 23% for autonomous mobile robot teams, consisting of as many as 75 agents, using a linearly scalable communication of agents.

The reminder of this paper is organized as follows. In the next section we outline the requirements of multi-agent planning for teams of up to 75 agents. In section 3, we start to discuss related works. In section 4, we introduce the basics of ALICA, which is used to describe team activities/plans. Furthermore, we sketch the basics of the planning framework pRoPhEt MAS [13]. Finally, in section 5 we evaluate the planning framework on relevant scenarios from the International Planning Competition.

2 Requirements

The requirements for multi-agent teams with a high number of agents in a team are:

- Modeling the global and local activities
- Automatic plan creation
- Low communication overhead

The description of team activities for autonomous mobile robots requires a suitable and intuitive description, instead of providing only single agent programs [15]. Furthermore, we would like to support task allocation and coordination instead of using predefined task-specific mapping to certain robots. With an increasing number of agents, manual modeling and task mapping takes a lot of time, and is hard to maintain. Hence, multi-agent systems require an intuitive method to control robot activities, and one that offers easy integration.

The communication bandwidth is limited. Hence, the planning process must also aim at keeping the number and size of messages low, particularly for large agent teams.

Finally, describing activities of autonomous mobile robot teams with an increasing number of agents requires a combination of modeling and planning.

3 Related Work

Heuristic search has become the predominant feature of problem solving for several years. The Fast Downward Planner [7] is a classical planning system based on heuristic search. Fast Downward is a best-first search planner that utilizes the information from domain transition graph as the heuristic to guide the search. Thus, it can deal with general deterministic planning problems encoded in the propositional fragment of PDDL2.2. The basic idea for the development of PDDL

[6] was to define a common interface to describe this problem class. PDDL defines a language to describe the existing world, actions to execute by agents and the goal state. The International Planning Competition (IPC) takes place every year, where newly developed planners evaluate difficult planning problems.

Helmert et al. [8] have proposed a concrete strategy for abstraction to derive better heuristics, and have empirically demonstrated the power of the merge-and-shrink abstraction heuristics. In particular, the empirical evaluation of the merge-and-shrink abstractions by Helmert et al. [8] suggests that, for many tasks, using a set of abstractions improves the overall heuristic guidance.

Brenner and Ivana [3] presented a new algorithmic framework in which situated dialogue is modeled as Continual Collaborative Planning (CCP). They showed how mixed-initiative dialogue that interleaves physical actions, sensing, and communication between agents occurs naturally during CCP. Thus, they introduced the language MAPL [4]. Their article describes a continual planning algorithm realized with MAPL. For the proof-of-concept, Brenner and Nebel evaluated MAPL in the grid world domain, where a team of four robots must find their position in the grid.

HPLAN-P [1] performs forward search using heuristics designed for propositional preferences. These are based on the relaxed planning graph (RPG) structure and use techniques such as summing the layers in which goals/preference facts appear (rather than relaxed plans) to estimate goal distance and preference satisfaction potential.

LAMA [11] is a classical planning system based on heuristic forward search. The system uses two heuristic functions in a multi-heuristic state-space search: a cost-sensitive version of the FF heuristic, and a landmark heuristic guiding the search towards states where many subgoals have already been achieved. Action costs are employed by the heuristic functions to guide the search to cheap goals rather than close goals, and iterative search improves solution quality while there is time remaining.

Burns et al. [5] developed parallel versions of best-first search to harness modern multicore machines. They showed that a set of previously proposed algorithms for parallel best-first search can be much slower than running A* sequentially. They presented a hashing function for parallel retractin A* (PRA*) that takes advantage of the locality of a search space and gives superior performance. They also presented another algorithm, PBNF, which approximates a best-first search ordering while trying to keep all threads busy.

Nissim et al. [9] developed a distributed planning system which uses a heuristic forward search. This system is evaluated for different IPC problems. The main disadvantage to this system is that communication effort increases rapidly as the number of agents increases.

The main contribution of most of state-of-the-art planning systems is to optimize the search heuristic. However, the quality of the search heuristic depends on the test domain. Our focus is to optimize planning independent of the search heuristic. Distributed planning often relies on high communication as in [9]. In real world applications like RoboCup[2] low communication approaches are required [14].

4 Planning Framework

In this section, we briefly introduce the planning framework. We will first introduce the basics of ALICA [15], and then we will sketch the basics of pRoPhEt MAS [13].

4.1 ALICA

ALICA is a language for describing team activities of interactive mobile agents from a global perspective. Originally, it was developed for the RoboCup Middle Size League. However, it has also been shown to be a viable and effective solution for other application domains, such as exploration robots [12] and autonomous vehicles in traffic [10].

The core elements of the language [14] are shown in Table 1.

$(\mathcal{A}, \mathcal{L})$	the domain signature	The domain signature consists of the set of possibly interacting agents and the logic with which the world is represented.
$\mathcal{R}$	a set of roles	This set contains all availables roles any agent can be assigned to.
$\mathcal{B}$	a set of behaviours	Behaviours are atomic action programs that form the means to interact with the environment.
$\mathcal{P}$	a set of plans	Each plan describes a specific cooperative activity.
$\mathcal{P}_\vee$	a set of plantypes	A plantype is a set of alternative plans.
$\mathcal{O}$	a set of planning problems	Defines a goal condition and a set of $\mathcal{P}$, to achieve the goal condition.
$\mathcal{T}$	a set of tasks	Each task intuitively describes a function or duty within plans, meant to be fulfilled by one or more agents.
$\mathcal{Z}$	a set of states	A state occurs within a plan as a step during an activity. It can contain plantypes and behaviours.
$\mathcal{W}$	a set of transitions	Each transition (z_1, z_2, ϕ) relates a predecessor state z_1 with a successor state z_2 and a condition $\phi \in \mathcal{L}(Pred, Func)$.

Table 1. Elements of a ALICA Program

The individual logic elements $\mathcal{L}$ defined by $\mathcal{L}(Pred, Func)$ are structured using the functions listed in Table 2.

[2] http://www.robocup.org

States: $\mathcal{P} \mapsto 2^{\mathcal{Z}}$	States maps plans to the set of contained states.
Tasks: $\mathcal{P} \mapsto 2^{\mathcal{T}}$	Tasks maps plans to the set of related tasks.
$\xi \colon \mathcal{P} \times \mathcal{T} \mapsto \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\})$	ξ defines the upper and lower bound of agents assignable to a task τ in plan p.
Pre: $\mathcal{P} \cup \mathcal{B} \mapsto \mathcal{L}_S$	Pre(p) denotes the precondition of plan or behaviour p.
Run: $\mathcal{P} \cup \mathcal{B} \mapsto \mathcal{L}_S$	Run(p) denotes the runtime condition of plan or behaviour p.
PlanTypes: $\mathcal{Z} \mapsto 2^{\mathcal{P}_\vee}$ Behaviours: $\mathcal{Z} \mapsto 2^{\mathcal{B}}$	PlanTypes(z) denotes the set of plantypes to be executed in state z. Behaviours(z) denotes the set of behaviours to be executed in state z.
Post: $\mathcal{Z} \mapsto \mathcal{L}_S$	Post(z) is a partial function, that maps terminal states of a plan to postconditions.
$\mathcal{U} \colon \mathcal{P} \mapsto 2^{\mathcal{L}_S} \mapsto \mathbb{R}$	$\mathcal{U}(p)$ is the utility function of p, evaluating p with respect to a set of formula.

Table 2. Structure Definitions of a ALICA Program

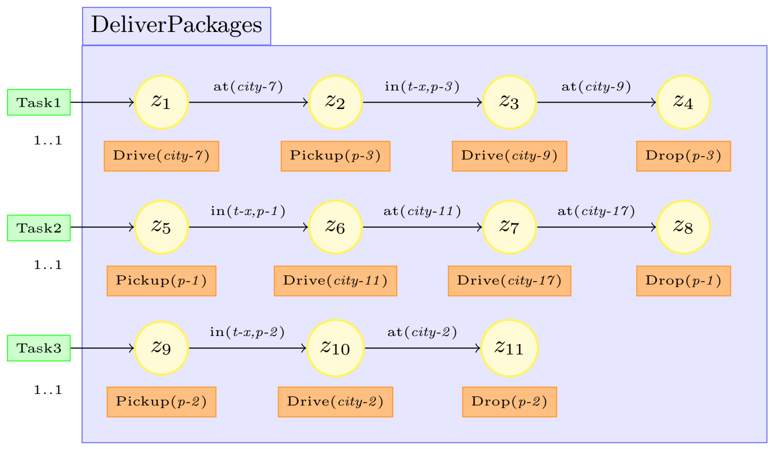


Fig. 1. Example ALICA plan for delivering packages by multiple agents

Figure 1 shows an example ALICA plan using the core elements of the language. This figure shows an example from the transport domain[3]. We defined roles $\mathcal{R}$ that are suitable for the task $\mathcal{T}$ dependent on the robot capabilities. Every agent in the team can assign to one of the tasks with respect to the minimum and maximum cardinalities (ξ) 1..1. The "DeliverPackages" plan $\mathcal{P}$ contains a state machine for every agent in team with several states $\mathcal{Z}$. Every state machine contains a plan, which in turn contains a state machine of basic behaviours $\mathcal{B}$. These plans represent the basic skills from the transportation domain. The basic skills of the agents are "Pickup", "Putdown" and "Drive". The agents can switch states with conditional transitions. The plan realizes the delivery of three packages.

In order to model plans ALICA offers a "PlanDesigner" which is a graphical tool based on the Eclipse Development Platform [2]. It supports modelling of all parts of an ALICA program, i.e., roles, tasks, plans, plantypes, utility functions, and conditions, as well as generating code from the models in a model-driven development fashion. The Ecore model is shown in figure 2. Modelled plans are stored in the XMI format and loaded afterwards by the runtime engine. However, for efficiency reasons, the tool provides mechanisms for generating platform-specific code for the evaluation of conditions and utilities. Since these evaluations happen very frequently during runtime, the generation of platform-specific code, which can be executed directly, results in enormous efficiency benefits. In order to facilitate an intuitive understanding, language elements are represented graphically.

4.2 pRoPhEt MAS

The planning framework pRoPhEt MAS (Reactive Planning Engine for Multi Agent Systems) is divided into two major parts (see Figure 3). The first part consists of "World" and "ALICA-Engine" and represents the basic ALICA components. The "ALICA-Engine" is the implementation of the language elements for section 4.1. In addition ALICA offers further algorithms for task allocation, role-task-mapping, supports coordinated execution in dynamic environments [14]. The "PlanBase" contains all modeled ALICA-plans that the team can access. Dependent on the actual world situation, ALICA will then select a suitable plan while reacting quickly to world changes. The second part consists of "ISharing" and "IPlanner". These components are used to expand the basics ALICA by a planning engine. "ISharing" is used to communicate plans after creation, and electing a leader, which starts the planning process. The election criteria can be defined by implementing the ISharing interface. At this time the robot with lowest id will be leader.

If the "PlanSelector" selects a plan containing a planning problem $\mathcal{O}$ (see language elements of section 4.1), which is briefly defined by basic actions and a goal description, the leader will start the planning process by the "PlannerBase". The resulting plan from the "PlannerRealization" will be communicated to all

[3] http://ipc.icaps-conference.org (IPC 2011)

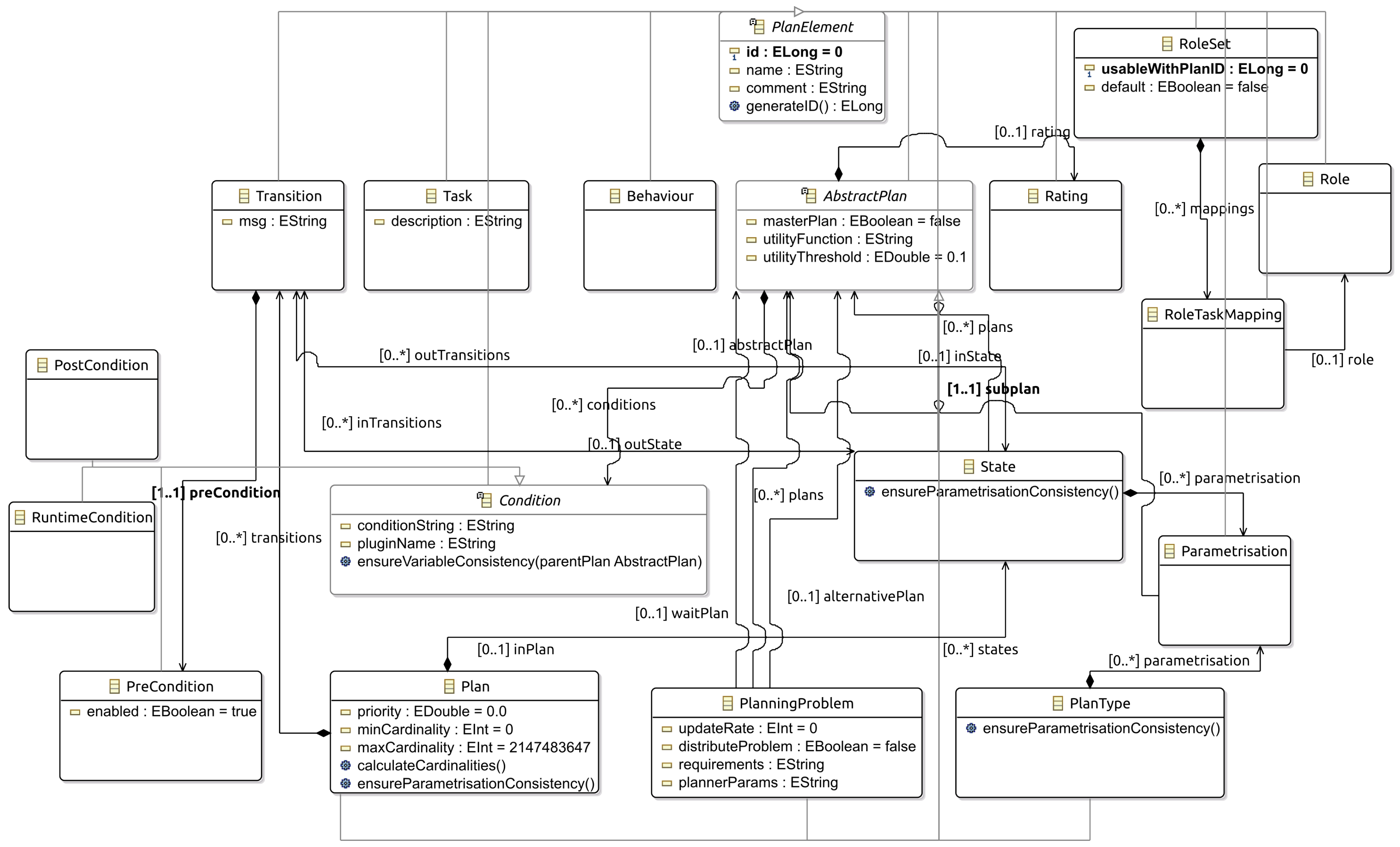


Fig. 2. Ecore model of ALICA's basic elements

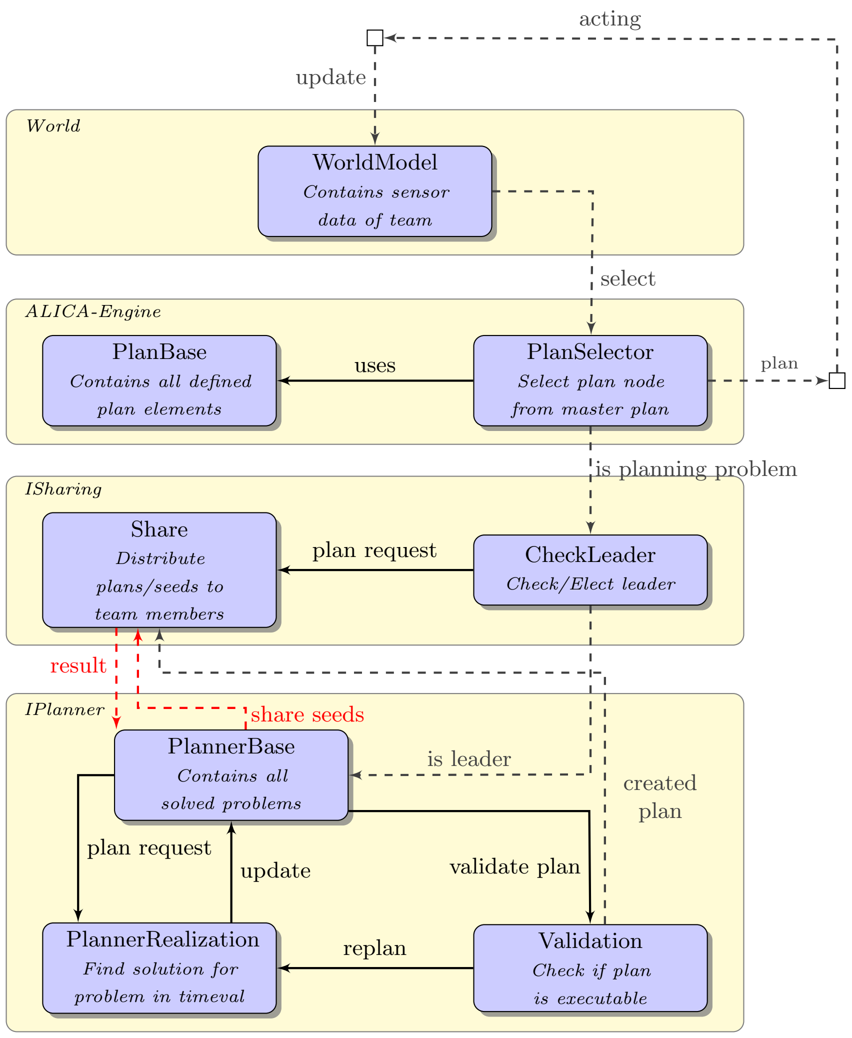


Fig. 3. Planning framework consisting of ALICA for describing team activities expanded by a planning engine

agents, if this plan is validated correctly by the "Validation" component. Hence, ALICA can react quickly in dynamic environments as evaluated in real world scenarios [13], though this is not the focus of this paper.

In order to decrease the search time and save memory for the planning process, the leader is able to distribute seeds of the search space to teammates using the "PlannerBase", which is shown in Figure 4. If an agent receives a seed, it will start the search, and send the solution back to the leader. If the leader receives the first result, he will share this solution to all other members, which will stop the search.

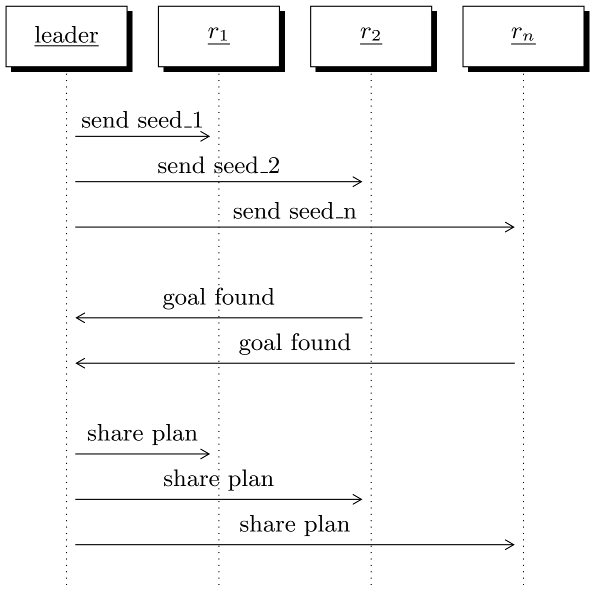


Fig. 4. Share seeds to accelerate search

5 Evaluation

In order to evaluate our framework, we use the defined problem of IPC 2011, as described in Section 3, and compare our results to state-of-the-art approaches. We used the planner "seq-sat-fdss-1", based on a Fast Downward Planning System [7], which we modifed to allow seed sharing with the entire team. The planner "seq-sat-fdss-1"participated in IPC 2011 and came in 2nd place in the transportion domain. Experiments were run on an Intel i7-2630QM CPU 2.00GHz processor, where we were allowed to use maximum one core. The results are shown in Figure 5, which shows calculation time and costs for different problems. These problems differ in map size, number of agents and packages to deliver, and can found on the IPC website. The time shows the total search time.

In the transportation domain the costs are defined by the total traveled distance. On average our approach decreases the costs by 1.3%. In addition, we were able to reduce the calculation time on average by 28.3%. For Problem 14, we reduced the calculation time by 65%.

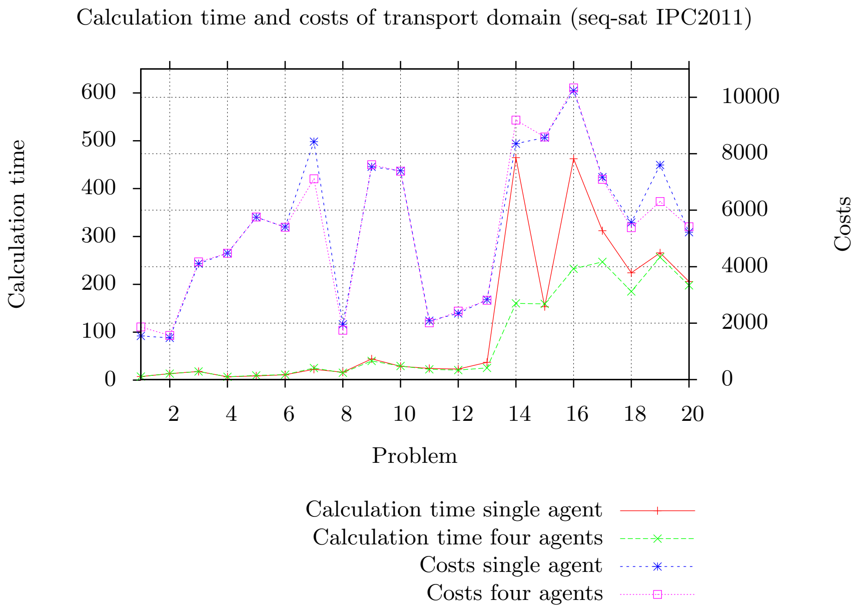


Fig. 5. Calculation time and costs for transport (IPC2011)

We later extended the problem and created a random map with 100 locations as shown in Figure 6. A distributed team with n members has to deliver n packages. Imagine a mail service group in a city that has around 100 different locations. This mail service wants to exchange packages between these locations via mobile, autonomous agents like copters or cars. The problem is how the agents should be assigned to deliver all packages. The results are shown in Figure 7. In this Figure the x represents the number of agents and the y shows calculation time and costs. The costs increased on average by 0.01%, but the execution time decreases on average by 10.17%.

The distributed planning scales linearly as $3 * (n - 1)$ with the number of agents. In a first round, we distribute seeds to all team members, which takes $(n - 1)$ messages. Next, in the worst case we wait for $(n - 1)$ results. Finally, we distribute the result to all $(n - 1)$ members. After receiving the result, every robot will stop the search.

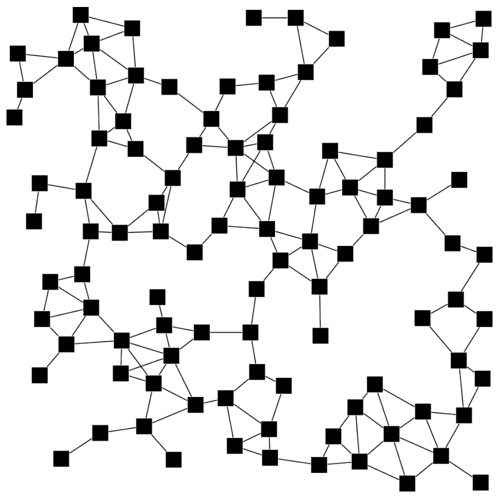


Fig. 6. Example map of transportation scenario

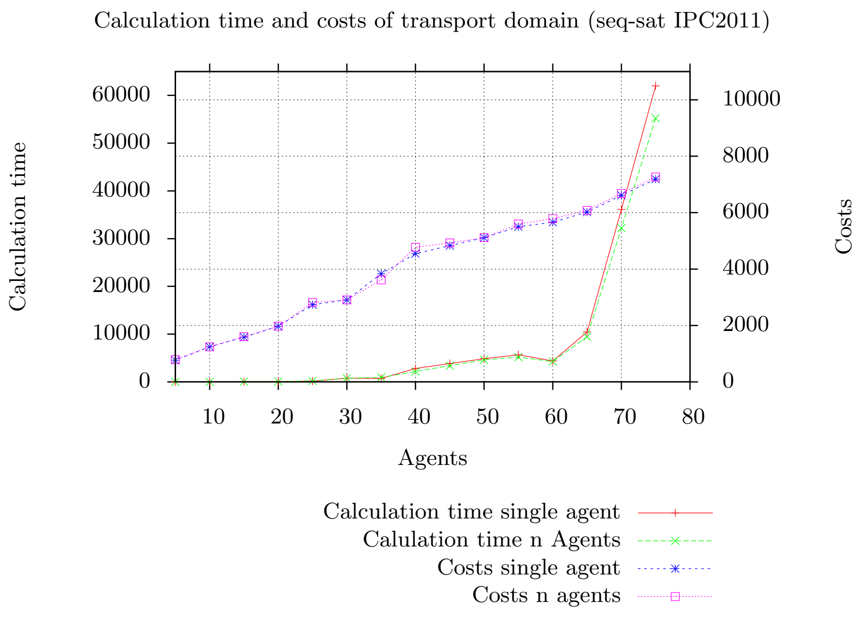


Fig. 7. Calculation time and costs of the map in Figure 6

6 Conclusions

The task planning for teams with a large number of mobile autonomous robots still offers improvements in research. The major problem is that cooperative distributed planning increases the communication rapidly as the number of agents increases. On the other side, severe resource limitations apply to the strategy of central planning, if complex planning problems shall be dealt with. Hence, it creates an opportunity to optimize planning for scenarios like disaster management, logistics operations, and many more.

However, planning is an increasingly complex task in multi-agent systems for an increasing number of robots. The state space grows tremendously with the number of robots. Moreover, in such domains, automatic plan generation reduces the overhead for maintenance, modeling, and testing enormously. Thus, planning is an important part of describing the activities of multi-agent systems.

The strategy of our framework is to *divide and conquer* to cope planning problems regarding resources like memory and communication bandwidth. Therefore, we use all robots as planning resources to reduce the planning time and divide the memory usage. The found solution of the robots will be shared, intermediately. Moreover, the communication burden scales linearly with an increasing number of agents.

In our evaluation, we took the transport scenario of the IPC2011 to compare our planning system to the state-of-the-art planner. Furthermore, we created more complex scenarios for the transport domain with up to 75 agents. We were able to improve the search time by up to 65% and 19.3% on average in the IPC problems. The costs in both scenarios were nearly the same (1.3% difference).

Our next steps are to evaluate the framework in the RoboCup domain using additional computational units to realize a set play in this dynamic environment.

References

1. Jorge A. Baier, Fahiem Bacchus, and Sheila A. McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 173(5-6):593–618, 2009.
2. W. Beaton and J. d. Rivieres. Eclipse Platform Technical Overview. Technical report, The Eclipse Foundation, 2006.
3. M. Brenner and I. Kruijff-Korbayov. A Continual Multiagent Planning Approach to Situated Dialogue. In *Proceedings of the LONDIAL (The 12th SEMDIAL Workshop on Semantics and Pragmatics of Dialogue)*. LONDIAL, 6 2008.
4. Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, June 2009.
5. Ethan Burns, Sofia Lemons, Wheeler Ruml, and Rong Zhou. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39(1):689–743, 2010.
6. M. Ghallab, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

7. M. Helmert. *The Fast Downward Planning System.* Journal of Artificial Intelligence Research 26, 2006.
8. Malte Helmert, Patrik Haslum, and Jrg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, pages 176–183, 2007.
9. R. Nissim and R. I. Brafman. Distributed Heuristic Forward Search for Multi-Agent Systems. *Computing Research Repository (CoRR)*, abs/1306.5858, 2013.
10. Stephan Opfer, Andreas Witsch, and Kurt Geihs. A Formal Multi-Agent Language for Cooperative Autonomous Driving Scenarios. nov 2014.
11. Silvia Richter and Matthias Westphal. LAMA is a classical planning system based on heuristic forward search. *Journal of Artificial Intelligence Research*, (39):127177, 2010.
12. D. Saur, T. R. Haque, R. Herzog, and K. Geihs. MAGiC : Multi-Agent Planning using Grid Computing concepts. In *12th International Symposium on Artificial Intelligence, Robotics and Automation in Space - i-SAIRAS 2014*, Quebec Canada, 2014.
13. Daniel Saur and Kurt Geihs. pRoPhEt MAS: Reactive Planning Engine For Multi-agent systems. In *13th International Conference on Intelligent Autonomous Systems*, 2014.
14. H. Skubch. *Modelling and Controlling of Behaviour for Autonomous Mobile Robots.* Westdeutscher Verlag GmbH, 2013.
15. H. Skubch, M. Wagner, R. Reichle, and K. Geihs. A modelling language for cooperative plans in highly dynamic domains. *Mechatronics*, 21:423–433, 2011.

3DVFH+: Real-Time Three-Dimensional Obstacle Avoidance Using an Octomap

Simon Vanneste, Ben Bellekens, and Maarten Weyn

CoSys-Lab, Faculty of Applied Engineering
Paardenmarkt 92, B-2000 Antwerpen
{ben.bellekens,maarten.weyn}@uantwerpen.be

Abstract. Recently, researchers have tried to solve the computational intensive three-dimensional obstacle avoidance by creating a 2D map from a 3D map or by creating a 2D map with multiple altitude levels. When a robot can move in a three-dimensional space, these techniques are no longer sufficient. This paper proposes a new algorithm for real-time three-dimensional obstacle avoidance. This algorithm is based on the 2D VFH+ obstacle avoidance algorithm and uses the octomap framework to represent the three-dimensional environment. The algorithm will generate a 2D Polar Histogram from this octomap which will be used to generate a robot motion. The results show that the robot is able to avoid 3D obstacles in real-time. The algorithm is able to calculate a new robot motion with an average time of 300 μs.

Keywords: Three-Dimensional Obstacle Avoidance, Octomap, Navigation and Planning, ROS, Robotics.

1 Introduction

Robotic applications such as airborne or underwater robots need to avoid obstacles in a 3D environment. These robots need to create a 3D map from the environment to locate obstacles. This can, for example, be accomplished by using a 3D laser scan Simultaneous Localization and Mapping (SLAM) algorithm [1] or a RGB-D SLAM algorithm [2,3]. These algorithms will build a 3D map from an unknown environment while at the same time locate the robot within this map. The 3D maps can be represented more efficiently by the octomap framework [4]. In this research we will use this octomap to determine the locations of obstacles so the robot can avoid them.

Researchers [5–7] have tried to solve the 3D obstacle avoidance problem with a 2D robot by creating a 2.5D height map from the original 3D map or by projecting the 3D map on a 2D map. This technique can only be used if the robot can move in 2D. When the robot can move in 3D, researchers [8,9] solved the 3D obstacle avoidance problem by planning and re-planning a 3D path by using the A* [10] and D* lite [11] algorithm. These techniques have the disadvantage that they are computational expensive. In this paper, we presents the Three

Dimensional Vector Field Histogram (3DVFH+) algorithm. This is an real-time obstacle algorithm, that will generate a robot motion to avoid obstacles in 3D.

The 3DVFH+ algorithm is based on the 2D VFH+ algorithm [12]. It will make a 2D polar histogram from an octomap of the environment. The algorithm consists of five stages to calculate a new 3D robot motion.

This research paper is organised in the following order. To begin with, the related researches will be described in Section 2. Then in Section 3, we will discuss the octomap framework. Section 4 will describe the multiple stages of the 3DVFH+ algorithm. Section 5 follows with an analysis of the results of this research. Next, we will come to a conclusion in Section 6. Finally, Section 7 proposes future improvements to the 3DVFH+ algorithm.

2 Related Work

Ulrich et al. [12] presented the VFH+ algorithm on which the 3DVFH+ algorithm is based. This obstacle avoidance system is able to plan local paths for a robot that can move in 2D. The VFH+ algorithm generates a smooth trajectory in real-time and takes the physical characteristics such as size and turning speed of the robot into account.

Hrabar [13] described a probabilistic roadmap planner. The planner generates a probabilistic roadmap which will be used to plan or re-plan a path with the D* lite algorithm [11]. When a path needed to be re-planned, it took on average 0.15s.

Burgard et al. [9] described a method for an autonomously quad-copter for indoor use. This research will plan a 3D path by using the D* lite algorithm [11] but only consider 2D actions of the robot. For these 2D points the multi-level map will be used to find the surface elevation under the robot to determine the change in altitude cost.

Maier et al. [5] developed a system that allows a Nao humanoid robot [14] to plan its path in a 3D environment and map new obstacles. The path planning algorithm uses a part of the octomap (vertical size of the robot) and project it to a 2D map. Next the path planning algorithm will calculate a path in two dimensions with an A* algorithm [10].

Nieuwenhuisen et al. [8] presents a local multiresolution path planning system. This system will use a multiresolution grid map that will be used for local path planning. This planner will plan a new path to the intermediate goals generated by the global path planner. This path will be planned by using an A* algorithm [10].

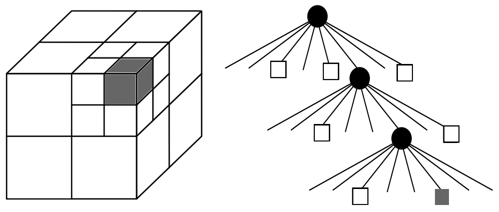


Fig. 1. Octree structure: white octree leaf nodes have status free, gray octree leaf nodes have status occupied

3 Octomap

The octomap [4] data structure is an efficient way to represent a 3D environment. The octomap can, for example, be accomplished by using a 3D laser scan SLAM algorithm [1] or a RGB-D SLAM algorithm [2,3]. With the octomap the 3DVFH+ algorithm can calculate a robot motion. This section will describe the basic principles of the octomap. The octomap framework is a 3D occupancy grid mapping framework based on the octree structure. The octree data structure is a hierarchical structure containing multiple nodes. These voxels (also called nodes) are cubic volumes in space. Every voxel can contain 8 or 0 child voxels. When the voxel contains 8 occupied child voxels the octomap will reduce these 8 child voxels to one parent voxel that is occupied. If no child voxels are occupied the voxels will be reduced to one parent voxel that is free. All the combinations in between will be stored by using the 8 child voxels. The number of layers in this hierarchical structure (also called the resolution of the octree) determine the precision and size of the octree (see Figure 1). This way the entire environment is represented by voxels.

Robot applications often use a probabilistic representation of the environment because input sensors introduce noise and therefore uncertainty into the system. The octomap leaf voxels contains a probabilistic number. This allows the octomap to have a probabilistic representation of the environment. This likelihood will be used to determine if te voxel is occupied or if voxel is free.

The octomap framework does not directly include information on the voxel location. The location can be calculated when descending the octomap tree. The root voxel has a fixed location and the number of the child voxel that is explored determines the location of the child voxel.

4 3DVFH+

3DVFH+ is an enhanced version of the two-dimensional VFH+ algorithm [12] which is used to work in a 3D environment. The algorithm uses an octomap to determine where the obstacles are located given the robot pose in a 3D environment. The 3DVFH+ algorithm uses five stages to calculate a new robot motion. These stages are described in the following subsections.

4.1 First Stage: Octomap Exploring

The octomap data structure makes use of the Octree data structure (see Figure 1). When the robot moves around in a large environment it is not possible to explore all the voxels due to computational limits. So the Octomap's exploring stage will only research voxels that lie within a bounding box around the robot. This bounding box has a size of $w_s * w_s * w_s$ (width, height and depth) and contains the Vehicle Center Point (VCP) as centroid. The VCP is the center point of the robot and the whole robot will be represented by this point. When a voxel lies within the bounding box, the voxel is an active cell $C_{i,j\ ,k}$ with i, j, k coordinates within the active region C_a. The exploring stage only needs to find voxels that lie within the boundaries of the bounding box. The algorithm needs to use the location of the voxels to determine which voxels need further exploration or which voxels can be ignored. But the location of the voxels is not implemented in the octomap data structure to reduce memory overhead [4]. These locations will be calculated when exploring the octomap tree. Voxels that are too far from the bounding box will not be explored. This principle also works on high level voxels, which will result in entire branches that can be ignored and improve the exploring speed without losing relevant information. When a voxel is found by the first stage, the second stage will use the information from the located voxel to make a 2D primary polar histogram

4.2 Second Stage: 2D Primary Polar Histogram

The second stage will add the information from the voxel (which has been found by the first stage) into the 2D primary polar histogram. This histogram (as shown in Figure 2) is a polar histogram where the location of an active voxel is determined by two angles. These angles are determined by the position of the voxel and the VCP. A weight that is calculated based on the voxel will be inserted into the 2D primary polar histogram with the two angles as coordinates.

First, the list of active cells will further be reduced by making a bounding sphere within the bounding box. This can be accomplished by calculating the euclidean distance $d_{i,j,k}$ between the VCP and each voxel. When the euclidean distance is larger then the radius of the bounding sphere $d_{i,j,k} > w_s/2$, the active cell will be ignored. A boundary sphere is necessary to create a rotation independent 2D polar histogram.

Next, the algorithm will calculate the two angles to determine the coordinates within the 2D primary polar histogram H^p. These angles are the azimuth angle β_z (x-axis of the 2D primary polar histogram) and the elevation angle β_e (y-axis of the 2D primary polar histogram). These angles are shown in Figure 3

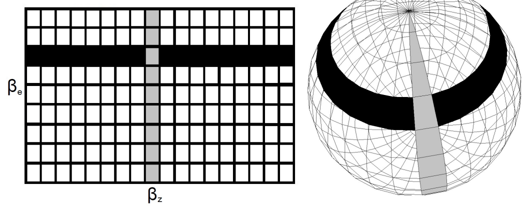
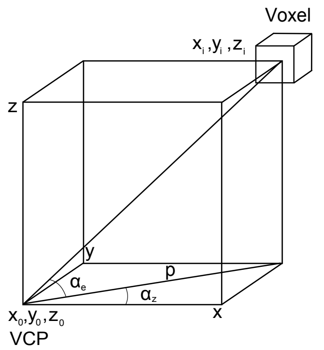


Fig. 2. 2D Polar Histogram



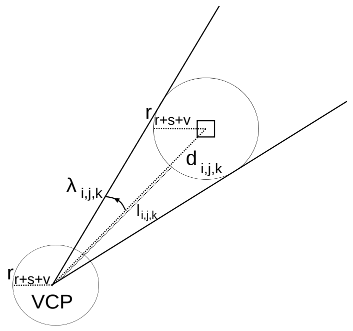


Fig. 3. An image that clarifies the angle calculations

Fig. 4. Enlargement of the voxels

as α_e and α_z.

The azimuth angle β_z is calculated by using (1) which is also used in the VFH+ algorithm [12]. The combination of the floor function and the partition by the resolution of the 2D polar histogram α will create a natural number (requirement for 2D primary polar histogram). The resolution is determined by the difference between the largest angle and the lowest angle that will lead to the same cell within the 2D polar histogram.

$$\beta_z = \text{floor}\left(\frac{1}{\alpha}\arctan\frac{x_i - x_0}{y_i - y_0}\right) \tag{1}$$

The elevation angle needs to be calculated in a different way (see (2) and (3)) because we need the elevation angle independent of the azimuth angle. This can be implemented by calculating p based on the x and y coordinates of the VCP and the voxel, see Figure 3.

$$\beta_e = \text{floor}\left(\frac{1}{\alpha}\arctan\frac{z_i - z_0}{p}\right) \qquad p = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} \tag{2}$$

this results in

$$\beta_e = \text{floor}\left(\frac{1}{\alpha}\arctan\frac{z_i - z_0}{\sqrt{(x_i - x_0)^2 + (y_i - y_0)^2}}\right) \tag{3}$$

The previous calculations used the VCP and the center of the voxel to calculate the two angles. The size of the robot is implemented by enlarging all the active voxels in the 2D primary polar histogram with r_{r+s+v} to compensate for the robots size, see Figure 4. This enlargement factor is calculated by adding the radius of the robot r_r, the safety radius r_s and the voxel size r_v. The voxels are enlarged so the algorithm can calculate the maximum and minimum angle in which the voxel lies, see (4).

$$\lambda_{i,j,k} = \text{floor}\left(\frac{1}{\alpha}\arcsin\frac{r_{r+s+v}}{d_{i,j,k}}\right) \tag{4}$$

The minimum distance between the voxel and the VCP can be calculated by subtracting the enlargement from the euclidean distance, see (5).

$$l_{i,j,k} = d_{i,j,k} - r_{r+s+v} \tag{5}$$

Now the algorithm has calculated which cells in the 2D primary polar histogram are influenced by the voxel, the algorithm needs to calculate the weight that the voxel will add to the 2D primary histogram. The weight of a voxel is calculated based on the euclidean distance $l_{i,j,k}$ and their occupancy certainty $o_{i,j,k}$, see (6).

$$H^p_{z,e} = \sum_{i,j,k \in C_a} \begin{cases} (o_{i,j,k})^2(a - bl_{i,j,k}) & \text{if } e \in [\beta_e - \frac{\lambda}{\alpha}, \beta_e + \frac{\lambda}{\alpha}] \\ & \text{and } z \in [\beta_z - \frac{\lambda}{\alpha}, \beta_z + \frac{\lambda}{\alpha}] \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Two constants a and b are calculated by using (7) to obtain a balanced a and b. The value of these constants is unimportant, only the relative size or fraction is important.

$$a - b\left(\frac{w_s - 1}{2}\right)^2 = 1 \tag{7}$$

The weight of a voxel will be used during stage 4 to obtain the 2D binary polar histogram.

4.3 Third Stage: Physical Characteristics

After the second stage, the third stage will calculate and add new information into the 2D primary polar histogram based on the physical characteristics of the robot and location of the voxel. When the robot has a certain heading θ retrieved by the RGBD-SLAM algorithm and speed, it cannot change direction instantly,

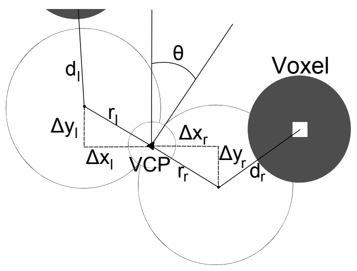


Fig. 5. Turning circles of the physical characteristics stage

so it will need to turn. When the robot is moving slowly, this stage will be ignored because it can change its direction instantly. The robot's turning trajectory depends on the robot's forward velocity, turning speed and the climbing speed. The turning speed and robot velocity are implemented in the VFH+ algorithm [12] so the same method can be used. The climbing acceleration has no influence on this part of the calculation.

First, the two center points of the turning circle need to be calculated (see Figure 5). This can be achieved in the same way as in the original VFH+ algorithm [12] (see (8) and (9)).

$$\Delta x_r = r_r \cdot sin\theta \qquad \Delta y_r = r_r \cdot cos\theta \tag{8}$$

$$\Delta x_l = -r_l \cdot sin\theta \qquad \Delta y_l = -r_l \cdot cos\theta \tag{9}$$

Secondly, the algorithm needs to check every active cell $C_{i,j,k}$ to see if it lies on the turning circle. This will be achieved by calculating the distance $d_{l,r}$ between the centre of the turning circle and the voxel, see (10). To detect if the voxel lies on the turning circle, the algorithm will compare this distance with the safety range r_{r+s+v} and the diameter of the turning circle $r_{l,r}$, see (11). The function $\Delta x(i)$ will calculate the x distance between the voxel and the VCP.

$$\begin{aligned} d_r &= \sqrt{(\Delta x_r - \Delta x(i))^2 + (\Delta y_r - \Delta y(j))^2} \\ d_l &= \sqrt{(\Delta x_l - \Delta x(i))^2 + (\Delta y_l - \Delta y(j))^2} \end{aligned} \tag{10}$$

$$d_r < (r_r + r_{r+s+v}) \qquad d_l < (r_l + r_{r+s+v}) \tag{11}$$

Thirdly, the algorithm also needs to take the climbing motion of the robot into account. The algorithm will calculate which cells within the 2D masked polar histogram are blocked by the voxel. This will be achieved by calculating for all the following α_e angles after the object which α_z angle is blocked by the voxel. First, the climbing motion constant f needs to be calculated. This is achieved by calculating the turning distance t to the obstacle, see (12). Next, the climbing motion constant is calculated with the altitude difference $z_i - z_0$ and the turning distance, see (13).

$$t = \frac{2.\pi.r(2.\alpha_z)}{360} \tag{12}$$

$$f = \frac{z_i - z_0}{t} \tag{13}$$

Next, a new unreachable altitude $z_{\alpha z}$ needs to be calculated for every α_z angle after voxel. The turning distance $t_{\alpha z}$, see (14) to the next α_z needs to be calculated to determine the unreachable altitude, see (15).

$$t_{\alpha z} = \frac{2.\pi.r(2.\beta_z.\alpha)}{360} \tag{14}$$

$$z_{\alpha z} = f t_{\alpha z} \tag{15}$$

Finally, the unreachable elevation angle β_e will be calculated using the altitude difference and the euclidean distance $l_{\beta z}$, see (16) and by using (17).

$$l_{\alpha z} = \sqrt{r^4 - 2.r^2.\cos(270 - 2.\alpha_z)} \tag{16}$$

$$\beta_e = \frac{1}{\alpha}\arctan(\frac{z_{\alpha z}}{l_{\alpha z}}) \tag{17}$$

These calculations can be extended so that the algorithm will calculate unreachable cells with a given forward speed and maximum climb speed.

4.4 Fourth Stage: 2D Binary Polar Histogram

The previous stages will generate a 2D primary polar histogram based on the voxels of the octomap. The fourth stage will reduce the information further by creating a 2D binary polar histogram based on the 2D primary polar histogram. This will be accomplished by comparing every cell in the 2D primary polar histogram with two thresholds τ_{low} and τ_{high}. When a value is higher than τ_{high} the point will be 1 in the 2D binary polar histogram. When a value is lower than τ_{low} the point will be 0 in the 2D binary polar histogram. If a point lies between the two thresholds the value next to the point will be used in the 2D binary polar histogram. The two thresholds allow the algorithm to distinguish real obstacles and measurement errors.

Because the 3DVFH+ algorithm uses a 2D polar histogram, the thresholds τ_{high} and τ_{low} need to change when using a different elevation angle β_e. The cells of the 2D polar histogram do not have the same size (as shown in Figure 2) so to compensate for this, different thresholds are required for different elevation angles, see (18). The size of these thresholds depends on the robot, the robot's speed, the window size of stage five, the octomap resolution and the bounding sphere size.

$$H^b_{\beta_z,\beta_e} = \begin{cases} 1 & \text{if } H^p_{\beta_z,\beta_e} > \tau_{\beta_e high} \\ 0 & \text{if } H^p_{\beta_z,\beta_e} < \tau_{\beta_e low} \\ H^p_{\beta_{z-1},\beta_p} & \text{otherwise} \end{cases} \tag{18}$$

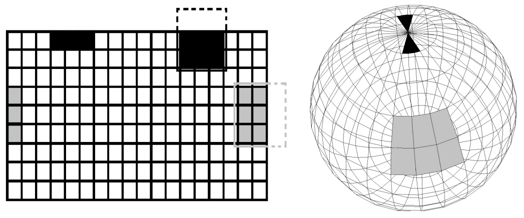


Fig. 6. Moving window of the fifth stage

4.5 Fifth stage: Path detection and selection

The fifth stage searches for available paths in the 2D binary polar histogram and selects the path with the lowest path weight. To determine which paths are available the algorithm will detect openings in the 2D binary polar Histogram by moving a window around the 2D binary polar histogram. This window will mark the path passable, if all the elements in the window are equal to 0. This way only large openings will be threaded as available paths. When implementing this window the algorithms needs to take the polar properties of the histogram into account. When the window crosses the boundaries of the histogram, the window will use elements that are connected by the rules of a 2D polar histogram.

When the window crosses the upper or lower boundary of the histogram (see black markings in Figure 6) the cells that need to be checked are located in the same elevation angle but 180 degrees rotated over the azimuth angle. When the window crosses the left or right border the window will check the cells on the other side of the histogram.

Next, three path weights will be calculated and combined into a single path weight for the candidate direction, see (19). The first path weight will be calculated based on the difference between the target angle k_t and the candidate direction v. The second path weight is the difference between the rotation of the robot θ and the candidate direction v. The last path weight is the difference between the previous selected direction k_{i-1} and the candidate direction v. The variable μ is used to select which of the three path weight has a larger impact on the final path weight. We used for our goal-oriented robot $\mu_1 = 5$, $\mu_2 = 2$ and $\mu_3 = 2$ as proposed by Ulrich et al. [12]. The path weight function can be adapted to enable a different behavior. For example the path weight function can give a preference to a turning motion instead of a climbing motion.

The function $\Delta(v_1, v_2)$ will calculate the difference between the two elevation angles and the two azimuth angles. From these differences the function will generate a weight based on the two calculated angle differences.

$$k_i = \mu_1.\Delta(v, k_t) + \mu_2.\Delta(v, \frac{\theta}{\alpha}) + \mu_3.\Delta(v, k_{i-1}) \tag{19}$$

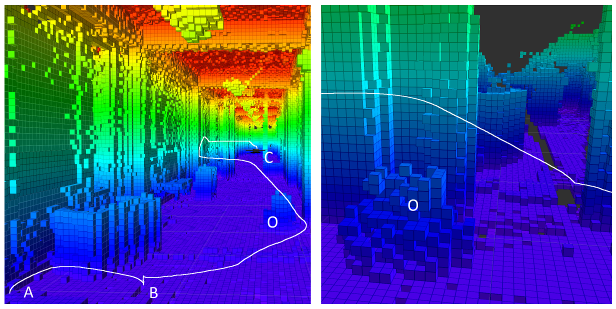


Fig. 7. Left: The full traveled path from point A to C. Right: A part of the traveled path from B to C. The robot avoided an obstacle by flying over the obstacle *O* and by turning.

When the path weight of all the candidate directions is calculated the algorithm can select the direction with the lowest weight. The chosen direction will be converted into a robot motion. The conversion can easily be implemented by making a decision tree that will generate a robot motion based on the coordinates of the calculated direction.

5 Results

In this section we will evaluate the 3DVFH+ algorithm. The requirements for the 3DVFH+ algorithm are that it needs to avoid obstacles in a 3D environment and perform these calculations in real time. These results are generated by using the Robot Operating System (ROS) [15] framework and the simulation software gazebo [16].

5.1 3D Obstacle Avoidance Result

The 3DVFH+ algorithm is a 3D obstacle avoidance algorithm, so to test the algorithm we simulated a quad-copter in gazebo. The quad-copter needed to fly from point A to point B and from point B to point C without bumping into obstacles (as shown in Figure 7). For this simulations, the dataset FR-079 corridor [17] is used. This octomap has a resolution of 0.05 m and is publicly available.

The left image in Figure 7 shows how the robot will go from point A to point B and from point B to point C. The right in Figure 7 image shows how the robot meets an obstacle *O* and will avoid it by a combination of flying over the obstacle and turning to avoid the wall.

5.2 Performance Result

This section will describe the performance of the 3DVFH+ histogram is measured by using the same dataset and the same intermediate points as in the previous section, see Figure 7. For these experiments we used a standard laptop with an Intel i7-3720QM processor with a NVIDIA GeForce GT 650M. The simulations were executed on a different machines to ensure that the simulation software did not influence the results. When performing the simulation the robot was flying around at a low speed, so the third stage was ignored. The third stage did not impact the performance measurements. The following parameters has the most influences to the performance of the algorithm: the bounding box size, the moving window size, the thresholds of stage four, the robot speed, and the octomap resolution. Due to the increasing complexity and the amount of voxels, the performance will raise or slow down.

We found that the average time that the algorithm needs to calculate a new robot motion is 326 μs. As a result, that the algorithm is capable of calculating on average a new robot motion 3061 times per second. The first three stages mainly determine the speed of the algorithm because these stage speed depends on the number of voxels within the bounding sphere. Based on multiple experiments we found that it takes 300 ns for every voxel to calculate the 2D primary polar histogram. The speed of the fourth and fifth stage are negligible with a speed of 0.2 μs for every robot motion. In practice the algorithm is limited to the other algorithms that the robot uses to calculate its pose and the octomap.

6 Conclusion

The 3DVFH+ algorithm is a real-time three-dimensional obstacle avoidance algorithm that uses an octomap to determine the obstacles locations. The algorithm can determine the location of these obstacles in real-time because the algorithm will only take obstacles into account that are located close to the robot. From the location of the obstacles the algorithm will make a 2D primary polar histogram based on the pose of the robot and location of the obstacles. Next, the algorithm will take the physical capability of the robot into account. In this 2D binary polar histogram the algorithm will find multiple paths, give them a path weight and determine the path with the lowest path weight. This path will be used to calculate a motion for the robot. By using these techniques the algorithm is able to calculate the robot motion real time in 3 dimensions.

7 Future work

Before the 3DVFH+ algorithm can be used, the algorithm needs to be configured based on the robot's size, robot's speed, octomap's resolution and multiple levels of thresholds to generate a 2D binary polar histogram. These parameters are chosen empirically. In future work this process could be automated.

In this system the thresholds that generate a 2D binary polar histogram are static. But when the robot is moving with a different speed, different thresholds could and should be used.

References

1. D. M. Cole and P. M. Newman, "Using laser range data for 3d slam in outdoor environments," in Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on. IEEE, 2006, pp. 1556-1563.
2. F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard,"An evaluation of the rgb-d slam system," in Robotics and Automation (ICRA), 2012 IEEE International Conference on. IEEE, 2012, pp. 1691-1696.
3. P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "Rgb-d mapping: Using depth cameras for dense 3d modeling of indoor environments," in the 12th International Symposium on Experimental Robotics (ISER), vol. 20, 2010, pp. 22-25.
4. A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," Autonomous Robots, vol. 34, no. 3, pp. 189-206, 2013.
5. D. Maier, A. Hornung, and M. Bennewitz, "Real-time navigation in 3d environments based on depth camera data," in Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on. IEEE, 2012, pp. 692-697.
6. J. Chestnutt, Y. Takaoka, K. Suga, K. Nishiwaki, J. Kuffner, and S. Kagami, "Biped navigation in rough environments using on-board sensing," in Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on. IEEE, 2009, pp. 3543-3548.
7. J.-S. Gutmann, M. Fukuchi, and M. Fujita, "3d perception and environment map generation for humanoid robot navigation," The International Journal of Robotics Research, vol. 27, no. 10, pp. 1117-1134, 2008.
8. M. Nieuwenhuisen and S. Behnke, "Hierarchical planning with 3d local multiresolution obstacle avoidance for micro aerial vehicles," in Proceedings of the Joint Int. Symposium on Robotics (ISR) and the German Conference on Robotics (ROBOTIK), 2014.
9. S. G. G. G. W. Burgard, "A fully autonomous indoor quadrotor."
10. P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," Systems Science and Cybernetics, IEEE Transactions on, vol. 4, no. 2, pp. 100-107, 1968.
11. S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," Robotics, IEEE Transactions on, vol. 21, no. 3, pp. 354-363, 2005.
12. I. Ulrich and J. Borenstein, "Vfh+: Reliable obstacle avoidance for fast mobile robots," in Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on, vol. 2. IEEE, 1998, pp. 1572-1577.
13. S. Hrabar, "3d path planning and stereo-based obstacle avoidance for rotorcraft uavs," in Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on. IEEE, 2008, pp. 807-814. Real-Time Three-Dimensional Obstacle Avoidance Using an Octomap 13
14. D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier, "Mechatronic design of nao humanoid," in Robotics and Automation, 2009. ICRA'09. IEEE International Conference on. IEEE, 2009, pp. 769-774.
15. "http://www.ros.org/", 2014, Page retrieved on 21/05/2014.
16. "http://gazebosim.org/", 2014, Page retrieved on 21/05/2014.
17. "http://ais.informatik.uni-freiburg.de/projects/datasets/octomap/", 2013, Page retrieved on 18/05/2014.