Preliminary Proceedings
DSLRob 2013

Editors: Christian Schlegel, Ulrik Pagh Schultz, Serge Stinckwich

November 4, 2013

Abstract

These are the preliminary proceedings of DSLRob 2013. The papers contained in this proceedings have all been accepted for presentation at DSLRob 2013, and have initially been revised according to reviewer comments, but are not yet in their final form. The final proceedings (containing the final papers) will be published on arxiv.org, as was the case for the previous iterations of DSLRob.

For more information, please see the DSLRob 2013 web page at `http://www.doesnotunderstand.org/public/DSLRob2013`

Table of Contents

Towards A Domain-specific Language For Pick-And-Place Applications

Thomas Buchmann[1], Johannes Baumgartl[2], Dominik Henrich[2] and Bernhard Westfechtel[1]

***Abstract*— Programming robots is a complicated and time-consuming task. A robot is essentially a real-time, distributed embedded system. Often, control and communication paths within the system are tightly coupled to the actual physical configuration of the robot. Thus, programming a robot is a very challenging task for domain experts who do not have a dedicated background in robotics. In this paper we present an approach towards a domain specific language, which is intended to reduce the efforts and the complexity which is required when developing robotic applications. Furthermore we apply a software product line approach to realize a configurable code generator which produces C++ code which can either be run on real robots or on a robot simulator.**

I. Introduction

A robot is essentially a real-time, distributed embedded system. Robot systems consist of different hardware components and different sensors which results in a very complex and highly variable system architecture. Often, control and communication paths within the system are tightly coupled to the actual physical configuration of the robot. As a consequence, these robots can be assembled, configured, and programmed only by experts. While this is the state of the art for robot programming nowadays, it is evident that using model-driven software engineering, and domain specific languages in particular, could provide great benefits to this domain by raising the level of abstraction and reducing complex and recurring programming tasks.

Model-driven software engineering [1], [2] puts strong emphasis on the development of high-level models rather than on the source code. Models are neither considered as documentation nor as informal guidelines how to program the actual system. In contrast, models have a well-defined syntax and semantics. Moreover, model-driven software engineering aims at the development of *executable* models. *Code generators* are used in model-driven software engineering, to transform the specification of higher-level models into source code. A *domain-specific language (DSL)* is a programming or specification language which is dedicated to a particular problem domain.

Software product line engineering (SPLE) [3], [4], [5] deals with the systematic development of products belonging to a common system family. Rather than developing each instance of a product line from scratch, reusable software artifacts are created such that each product may be composed from a library of components. Furthermore, it provides means to capture and manage the variability of a particular application domain. In common approaches, *feature models* [6] are used for that purpose.

In this paper, we present the work in progress of our domain-specific language for pick-and-place applications and especially the configurable code generator which produces C++ code.

II. A Domain Specific Language for Pick-And-Place Applications

As stated in Section I, programming a robot is a very complex task. Resulting programs highly depend on the robot's hardware and the environment in which the robot is being operated. Thus, our approach - whose basic ideas are presented in [7] - aims at enabling programmers without dedicated knowledge in the robotics domain to specify robot applications.

```
1  program pack_box
2
3  color RED (255,0,0)
4  object redbox {
5      objectConfig boxcfg {..
9      geometry {..
12     color:=RED
13 }
14
15 /*depth camera */
16 sensor cam {..
22
23 robot rb1 {..
31
32 mylist : RSet<ObjectDeclaration> := cam::perceive
33
34 foreach (ding : ObjectDeclaration in mylist) {
35         ding.pick(rb1)
36         ding.place(redbox)
37 }
```

Fig. 1. A small example of our DSL code.

The core part of our approach is a declarative domain-specific language for pick and place applications. We chose to start with this domain, since it covers basic robotic tasks like moving robots, grasping objects and placing them at a different location. These tasks, which sound easy at first glance, include inherently complex subtasks like object modeling, path planning, grasp planning, and placement planning. To empower users without dedicated background in those tasks, we abstracted from those concepts. Instead, modeling and planning operations are implemented in a C++ framework, which is used by the code generator presented

[1]T. Buchmann and B. Westfechtel are with Chair of Applied Computer Science I (Software Engineering), University of Bayreuth, 95447 Bayreuth, Germany firstname.lastname@uni-bayreuth.de

[2]J.Baumgartl and D. Henrich are with Chair of Applied Computer Science III (Robotics and Embedded Systems), University of Bayreuth, 95447 Bayreuth, Germany firstname.lastname@uni-bayreuth.de

in Section III. As a consequence, the DSL code can be kept simple and human readable as shown in Figure 1.

A. Design Decisions

The most difficult task when designing a domain-specific language is to find the right level of abstraction as well as the required keywords. A basic question is whether object and hardware declarations are required in the DSL or not.

The sample DSL code in Figure 1 contains various declarations of different types. In its current status the DSL allows declarations, for e.g. colors, objects, sensors, and robots as well as object and robot configurations (c.f. lines 2 - 41 in the sample).

While declarations of sensors and robots are useful when the generated code is meant to be run using several robots, declarations might be obsolete when using an educated distribution planner to assign a subtask to a specific sensor or robot. However, in this paper we focus on one robot with one sensor network capable to model objects to manipulate with, where those hardware declarations are just convenient.

Dependent on used hardware, requirements for the algorithms might vary. Those dependencies should be implemented as constraints on the feature model of the product line and not be part of the DSL, since the concrete algorithms are transparent to the user, likewise the interaction with the concrete hardware.

The second design decision is concerned about the keywords that should be provided by the DSL. The current version of the DSL comprises keywords for object, sensor, and robot declarations and configurations. Furthermore keywords for built-in data types and control structures are included. The keyword `object` in the declarative part can be used to define static environment or known objects. Following an object-centered approach, objects are linked with hardware by keywords for object manipulation (`pick` and `place`), robot movement (`move`), and operations on sensors (e.g. `perceive`). A concrete (planning) algorithm must be available for each of the keywords. However, different realizations concerning one keyword might exist. Those are selected depending on the hardware.

B. Implementation

We decided to use the Xtext[1] framework for our textual DSL. Xtext allows the specification of a context-free grammar of a language. It uses ANTLR[2] as a parser generator, which means that is able to parse LL(*) grammars. Furthermore, the Xtext framework allows to enrich the Xtext grammar specification with context-sensitive information, which is used to unparse a text into an Ecore-based tree representation. The resulting, automatically generated editor comprises full-fledged support for syntax highlighting and code completion.

[1]http://www.eclipse.org/xtext
[2]http://www.antlr.org

III. Configurable Code Generator

The DSL code needs to be compiled into executable code in order to be run on real robots or within a simulator. To this end, we use the Accelo[3] framework which provides an implementation of the *OMG MOF Model to Text standard* [8]. Acceleo can be used to specify code generators for arbitrary Ecore-based metamodels.

The target platform of the code generator is GeNBot - a C++ framework which comprises different algorithms for path planning, grasp planning and placement planning as well as a modular interface to different robots (Kuka LWR, Kuka KR16-2, Stäubli RX130) and simulators.

Acceleo provides a template-based code generation engine equipped with its own template language MTL. OCL[4] is used to retrieve model information dynamically, which is used in the templates to generate code.

Figure 2 shows a small cutout of our code generation templates which is used to initialize a LWR robot controller. The code formatted in blue color between square brackets depicts dynamic code fragments which are extracted from the model (e.g. the DSL code) at runtime. Text formatted in black color is static text which is used as it stands each time the template is invoked.

The code which is produced by the template above is shown in Listing 1. The code contains fragments which are neccessary to initialize a Kuka LWR robot controller in the GeNBot framework. This code is necessary in every application which is intended to be run on this type of hardware. But it also contains some variable parts like the number of joints for example so that it could not be reused by plain copy and paste in “traditional” programming approaches.

As stated in [7], our approach aims at integrating a product line approach to cover the variability which may occur in the target domain due to changing hardware (robots, sensors) and software (algorithms used for planning tasks etc.). Thus, we started to integrate our FAMILE environment, which is dedicated to the model-driven development of software product lines [9], [10].

[3]http://www.eclipse.org/acceleo
[4]Object Constraint Language

```
181 [template private robotPhysicalInitializer(aRobot : RobotDeclaration)]
182 /* Instantiate the robot controller */
183 [RobotPrefix()/]::BaseType::Ptr [aRobot.name/] =
184     GeNBot::Factory::buildRobotController<
185         [RobotPrefix()/],
186         [HaltInterpolator()/]<[RobotNumberOfJoints()/]>,
187         [JointInterpolator()/]<[RobotNumberOfJoints()/]>,
188         [NSAInterpolator()/]<[RobotNumberOfJoints()/]>
189         (
190             [RobotPrefix()/]::BaseType::RobotIKPtr
191                 (new [InverseKinematics(KinematiksFile())/]),
192             [RobotPrefix()/]::BaseType::RobotFKPtr
193                 (new [ForwardKinematics(KinematiksFile())/])),
194                 [IpoCycle()/]
195         );
196
197 [RobotPrefix()/]::BaseType::RobotIKPtr
198     [aRobot.name/]_inverse_kinematics_ptr
199         (new [InverseKinematics(KinematiksFile())/]);
200 [/template]
```

Fig. 2. A cutout of the code generator templates.

Listing 1. Cutout of the generated C++ code

```
/* Instantiate the robot controller */
GeNBot::LWRRobotController::BaseType::Ptr rb1 =
     GeNBot::Factory::buildRobotController<
    GeNBot::LWRRobotController,
    GeNBot::HaltInterpolator<7>,
    GeNBot::ReflexxesJointInterpolator<7>,
    GeNBot::ReflexxesNSAInterpolator6D<7>
    (
      GeNBot::LWRRobotController::BaseType::
          RobotIKPtr
        (new GeNBot::LWR_ik_AC()),
        GeNBot::LWRRobotController::BaseType::
            RobotFKPtr
          (new GeNBot::LWRFK(std::string("/path/
              to/kinematicsFile.xml"))),
        0.034f
    );

GeNBot::LWRRobotController::BaseType::RobotIKPtr
  inverse_kinematics_ptr
    (new GeNBot::LWR_ik_AC());
```

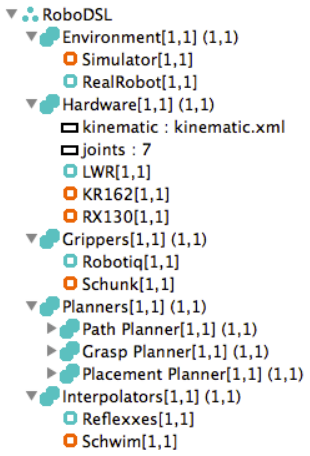


Fig. 3. A sample feature configuration.

Currently, we are addressing the variability which concerns the code generator. Depending on the target platform (simulator, real robot) different building blocks of the C++ framework are used in the generated C++ code. Furthermore, three types of robots (Kuka LWR, Kuka KR16-2, Stäubli RX130) and different planning algorithms are supported. Our FAMILE toolchain uses feature models [6] to capture commonalities and variabilities of the product line. Figure 3 shows a sample feature configuration of the product line, which is used as an input during the code generation process to bind the variability. Elements marked with cyan colored circles depict features which are included in the current feature configuration, while orange colored circles mark excluded ones. Features also may have attribute values, e.g. feature `Hardware` contains the attribute `joints`. In its current state, the configuration (e.g. selecting / deselecting) the appropriate features in the feature model has to be done manually.

IV. Future Work

In its current state, our approach already covers variability on the level of the code generator. E.g. different code is being generated depending on the feature configuration which is passed to it. But variability in robotics hardware does not only affect the generated code (by initializing and using appropriate building blocks of the GeNBot framework), it may also concern the language itself. The presence / absence of hardware or software might result in the inclusion or exclusion of language constructs. In this case, the end user cannot specify DSL programs which cannot be run on the target hardware. As a consequence, variability on the level of the grammar of the DSL is required. Our toolchain FAMILE was built to support model-driven software product line engineering for arbitrary Ecore-based domain models. As an Xtext grammar is parsed into an Ecore-based syntax tree (as the Xtext editor is specified with Xtext itself) FAMILE can be used to map features from the feature model to grammar production rules. As a result, language elements decorated with feature expressions evaluating to false (in case the respective features are excluded from the current feature configuration) are omitted.

Furthermore, Acceleo also provides an Ecore-based model for its abstract and concrete syntax. While the connection between feature model and code generation templates is realized via Acceleo queries at the moment, we are currently working on using the feature mapping capabilities of FAMILE for it as well. Unfortunately it does not work out of the box, as Acceleo (contrastingly to Xtext) does not use a parser which automatically creates instances of this Ecore model in memory.

In order to support automatic configuration depending on the used hardware, a dedicated interface to the hardware as well as a protocol providing the required information (which will be used for an automatic configuration of the feature model) is necessary. This will also be addressed in future work.

Finally, plan to extend the language to address other robotic application domains apart from pick and place as well.

V. Related work

In [7], we present the basic ideas behind our approach, which is intended to provide textual DSLs for robotic applications, which can be adapted at runtime according to the actual robot configuration. While [7] offers a conceptual overview, we present the first version of a DSL for pick-and-place applications and a configurable code generator which creates C++ code in this paper.

In [11], the authors present an approach which uses a DSL to handle run-time variability in programs for service robots. The approach presented by Inglés-Romero et al. aims to support developers of robotic systems (e.g. experts in the robotics domain) while our approach is not restricted to robotic experts only. Even regular programmers without a dedicated background in robotics are able to write robotic programs with our DSL. Furthermore, the DSL is only able

to express variability information. It is not possible to specify the behavior of the robot.

Steck et al. present an approach [12] that is dedicated to a model-driven development process of robotic systems. They present an environment called *SmartSoft* [13] which provides a component based approach to develop robotics software. The SmartSoft environment is based on Eclipse and the Eclipse Modeling Project[5]. It uses Papyrus[6] for UML modeling. By using a model-driven approach, the authors focus on a strict separation of roles throughout the whole development life-cycle. Again, experts in the robotics domain are addressed with this approach while our approach doesn't require expert knowledge in robotics.

RobotML [14], a modeling language for robot programs also aims to provide model-driven engineering capabilities for the domain of robot programming. RobotML is an extension to the Eclipse-based UML modeling tool Papyrus. Papyrus puts strong emphasis on UML's profile mechanism, which allows domain-specific adaptations. RobotML provides code generators for different target platforms, like Orocos, RTMaps, Arrocam or Blender/Morse. The approach presented by Dhouib et al. addresses developers of robot programs or algorithms, while our approach can also be used by regular programmers (of course robotic experts can use it as well and may gain an increase in productivity).

Bubeck et al. present in [15] an overview about best practices for system integration and distributed software development in service robotics. Furthermore, the authors develop *BRIDE*[7], a graphical DSL for ROS developers. Using BRIDE, new ROS nodes or ROS-based systems can be specified in a graphical way and corresponding C++ or Python code may be generated. In addition, the required launch files for the ROS environment including the relevant parameters and dependencies are generated as well, similar to the approach which we used in our case study as described in [7]. Similar to the approaches discussed above, BRIDE also addresses robot experts only.

In [16], Schultz et al. present an approach for a domain-specific language intended for programming self-configurable robots. The DSL is targeted towards the ATRON self-reconfigurable robot. Like all other approaches mentioned in this section, it aims to provide a higher-level of abstraction for robot experts.

VI. Conclusion

In this paper we presented our approach towards easy robot programming for personal robots. We demonstrated the feasibility of our approach by presenting a small and declarative domain-specific language for pick and place applications. Furthermore, a product line approach was used to realize a configurable code generator for C++.

[5]http://www.eclipse.org/modeling/

[6]http://www.eclipse.org/papyrus

[7]http://ros.org/wiki/bride

References

[1] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Indianapolis, IN: Wiley Publishing, 2003.

[2] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[3] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Boston, MA, 2001.

[4] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer Verlag, 2005.

[5] D. M. Weiss and C. T. R. Lai, *Software Product Line Engineering: A Family-Based Software Development Process*, Boston, MA, 1999.

[6] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University, Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[7] J. Baumgartl, T. Buchmann, D. Henrich, and B. Westfechtel, "Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines," in *Proceedings of the 8th International Conference on Software Paradigm Trends (ICSOFT-PT 2013)*, J. Cordeiro and M. van Sinderen, Eds., Reykjavik, Iceland, Jul. 2013, pp. 147–157.

[8] OMG, *MOF Model to Text Transformation Language, Version 1.0*, formal/2008-01 ed., OMG, Needham, MA, Jan. 2008.

[9] T. Buchmann and F. Schwägerl, "FAMILE: tool support for evolving model-driven product lines," in *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*, ser. CEUR WS, H. Störrle, G. Botterweck, M. Bourdells, D. Kolovos, R. Paige, E. Roubtsova, J. Rubin, and J.-P. Tolvanen, Eds. Building 321, DK-2800 Kongens Lyngby: Technical University of Denmark (DTU), Jul. 2012, pp. 59–62.

[10] T. Buchmann and F. Schwägerl, "Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability," in *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, ser. FOSD '12. New York, NY, USA: ACM, 2012, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/2377816.2377822

[11] J. F. Inglés-Romero, A. Lotz, C. V. Chicote, and C. Schlegel, "Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties," in *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-12, co-located with SIMPAR 2012)*, E. Menegatti, Ed., Tsukuba, Japan, 2012.

[12] A. Steck, D. Stampfer, and C. Schlegel, "Modellgetriebene Softwareentwicklung für Robotiksysteme," in *AMS*, ser. Informatik Aktuell, R. Dillmann, J. Beyerer, C. Stiller, J. M. Zöllner, and T. Gindele, Eds. Springer, 2009, pp. 241–248.

[13] A. Steck and C. Schlegel, "Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development," in *Proceedings of the First International Workshop on Domain-Specific Languages and models for ROBotic systems (IROS - DSLRob)*, Taipei, Taiwan, 2010.

[14] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, I. Noda, N. Ando, D. Brugali, and J. Kuffner, Eds. Springer Berlin Heidelberg, 2012, vol. 7628, pp. 149–160. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34327-8_16

[15] A. Bubeck, F. Weisshardt, T. Sing, U. Reiser, M. Hagele, and A. Verl, "Implementing best practices for systems integration and distributed software development in service robotics - the care-o-bot robot family," in *System Integration (SII), 2012 IEEE/SICE International Symposium on*, 2012, pp. 609–614.

[16] U. P. Schultz, D. J. Christensen, and K. Stoy, "A Domain-Specific Language for Programming Self-Reconfigurable Robots," in *Workshop on Automatic Program Generation for Embedded Systems (APGES)*, 2007, pp. 28–36.

Towards a Domain Specific Language for a Scene Graph based Robotic World Model

Sebastian Blumenthal and Herman Bruyninckx

Abstract— **Robot world model representations are a vital part of robotic applications. However, there is no support for such representations in model-driven engineering tool chains. This work proposes a novel Domain Specific Language (DSL) for robotic world models that are based on the Robot Scene Graph (RSG) approach. The RSG-DSL can express (a) application specific scene configurations, (b) semantic scene structures and (c) inputs and outputs for the computational entities that are loaded into an instance of a world model.**

I. Introduction

Robots interact with the real world by safe navigation and manipulation of the objects of interest. A digital representation of the environment is crucial to fulfill a given task. Although a world model is a central component of most robotic applications a *Domain Specific Language* (DSL) has not been developed yet. One reason for this is the lack of a common world model approach. The scene graph based world model approach *Robot Scene Graph* (RSG) [1] tries to overcome this hurdle. It acts as a shared resource for a full 3D environment representation in a robotic system. It accounts for dynamic scenes by providing a short-term memory, allows to hierarchically organize scenes, supports uncertainty for object poses, has semantic annotations for scene elements and can host computational entities. While other approaches have a stronger focus on certain world modeling aspects like probabilistic tracking of semantic entities [2] or hierarchical representations for geometric data [3], the RSG emphasizes a holistic view on the world modeling domain. This work extends the RSG approach by a *RSG-DSL* to model the structural and computational aspects of the scene graph. It is accompanied by a model to text transformation to generate code for an implementation of the RSG which is a part of the BRICS_3D C++ open source library [4].

A DSL is a formal language that allows to express a certain aspect of a problem domain. It creates an abstraction in order to quickly create new applications and it imposes constraints on a programmer to prevent from programming errors. A structured development of a new DSL is organized in four levels of abstractions M0 to M3 [5]:

- **M0**: The M0 level is an instantiation of a DSL model. Typically this results in generated code for a (generic) programming language that can be compiled and executed.
- **M1**: The M1 level comprises models that conform to a certain DSL that is defined on the M2 level.
- **M2**: A meta model on the M2 level specifies the DSL in a formal way. This definition has to conform to the meta meta model of M3.
- **M3**: The M3 level defines the meta meta model which is a generic model to describe DSLs.

The goal of the RSG-DSL for a robotic world model is manifold. This DSL can describe the structural and behavioral parts of a scene that are part of a specific robotic application. It allows to combine the required elements at design time into an executable instance of a world model. The a priori known structure of a system can include the involved robots with their kinematic structures and their geometries, previously known parts in the environment or the places in the structure where to store online sensor data. Results of the behavioral *function blocks*, which can contain any kind of computation, are stored in the scene graph as well. The selection and the configuration of the function blocks has an important influence on how the world model will appear at runtime. For example, the presence of an object recognition function block can enrich the scene graph with task-relevant objects. The above items can be specified on the code level. However, the RSG-DSL reduces the required number of lines of code to encode the scene graph. The API assumes a correct order of creation of scene primitives, while the RSG-DSL does not have this restriction.

The proposed DSL allows to express input and output scene structures for the function blocks. For instance, a segmentation algorithm module that consumes a point cloud and generates a set of new point clouds with associated spatial relations pointing to the center of the segments. In addition, the RSG-DSL is able to express prior semantic knowledge about a scene. It is possible e.g. to encode a generic version of a table that consists of a table plate and four legs. This can serve as input for a function block that analyzes the perceived scene to recognize that particular structure.

The remainder of the paper is organized as follows: Section II summaries related work and Section III gives a brief introduction to the world model concept. Details of the RSG-DSL are explained in Section IV and its capabilities are illustrated with examples in Section V. The paper is closed with a conclusion in Section VI.

II. Related Work

Recently interest has been risen in robotics to create DSLs for various sub-aspects of robotic systems. The Task Frame

All authors are with the Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium. Corresponding author: sebastian.blumenthal@mech.kuleuven.be

Formalism DSL [6] has been proposed to describe the control and coordination aspects of robotic software systems. A DSL to express geometric relations between rigid bodies [7] helps to correctly set up spatial relations as constraints will be automatically evaluated on the M1 level. Two DSL variants are discussed: one version is embedded into the Prolog programming language and the second one uses the Eclipse Modeling Framework (EMF) [8]. The Prolog approach results in a directly executable code while the EMF variant benefits from the Eclipse tool chain including an editor that supports syntax highlighting and auto-completion. The Grasp Domain Definition Language [9] is developed in the EMF framework as well. It demonstrates that multiple dedicated robotic languages can be further composed into more complex ones.

DSL approaches in the 3D computer animation domain have been recently developed for 3D scenes. The streaming approach for 3D data [10] uses a meta model for scene elements to cope with various 3D scene formants. In a similar way the SSIML [11] approach tries to abstract from the existing 3D formats and APIs method calls. It is meant as a DSL for development of 3D applications. However, in contrast to a robotic world model the complete access to the world state is given. To the best of the authors knowledge a DSL for a robotic world model does not exist yet.

III. World Model Primitives

The goal of the world model is to act as a shared resource among multiple involved processes in an application. Such processes could be related to the various robotic domains like planning, perception, control or coordination. To be able to satisfy the needs of the different domains the world model has to offer at least the following set of properties: It appears as shared and possibly distributed resource. It it takes the dynamic nature and imprecision of sensing of real-world scenes into account, allows for multi-resolution queries and supports annotations with semantic tags.

The scene graph based world model RSG consists of objects and relations among them [1]. These relations are organized in a *Directed Acyclic Graph* (DAG) similar as for approaches used in the computer animation domain. The directed graph allows one to structure a scene in a hierarchical top-down manner. For instance, a table has multiple cups, whereas multiple tables are contained in a room, multiple rooms in a building and so on. Traversals on such a hierarchical structure can stop browsing the graph at a certain granularity to support multi-resolution queries. The graph itself supports four different types of nodes. All node types have in common that each instance has a unique ID, a list of attached attributes for semantic tags and one or more parent nodes. Details of the four node types are given below:

- **Node**: The `Node` is a generic leaf in the graph. It can be seen as a base class for the other node types.
- **GeometricNode**: A `GeometricNode` is a leaf in the graph that has geometric data like a box, a cylinder, a point cloud or a triangle mesh. The data is time stamped and immutable i.e. once inserted the data connote be altered until deletion to prevent inconsistencies in case multiple processes consume the same geometric data at the same time.
- **Group**: The `Group` can have child nodes. These parent child relations form the DAG structure.
- **Transform**: The `Transform` is a special `Group` node that expresses a rigid transform relation between its parents and its children. Each transform node in the scene graph stores the data in a cache with associated time stamps to form a short-term memory.

The `Transforms` are essential to capture the dynamic nature of a scene as changes over time can be tracked by inserting new data into the caches. Moreover, such a short-term memory enables to make predictions on the near future. This requires dedicated algorithms to be executed by the word model as described later. In contrast to the `Transforms`, geometric data is defined to be immutable. Hence, changes on those data structures do not have to be tracked. In case a geometry of a part of a scene does change over time a new `GeometricNode` would have to be added. The accompanying time stamps still allow to deduce the geometric appearance of a scene at a certain point of time. All temporal changes in the world model are explicitly represented.

The RSG approach uses a graph structure. Thus, it is possible to store multiple paths formed by the preceding parents to a part of a scene. This case expresses that multiple information to the same entity is available. For example, an object could be detected by two sensors at the same time. Different policies for resolving such situations are possible. Selection of the most promising path like the latest path denoted by the latest time stamps associated with the transforms is one possibility, while choosing a path with the help of the semantic tags is another one. Probabilistic fusion strategies [12] are an alternative, given covariance information on the transform data is available. This kind of uncertainty data can be stored in the temporal caches as well. The details of representing uncertainty and fusion strategies are planned as future work.

Besides the structural and temporal aspects, the world model contains *function blocks* to define any kind of computations. A function block consumes and produces scene graph elements. Algorithms for estimating near future states are one example of such a computation. A function block can be loaded as a plugin to the world model and is executed on demand. This allows to move the computation near to the data to improve efficiency of the executed computations. Conceptually the scene graph is a shared resource among all function blocks. Concurrent access to the scene is possible since geometric data is defined to be immutable. `Transforms` provide a temporal cache such that inserting new data will not affect retrieval of transform data by another function block as long as queries do not go beyond the cache limits.

The RSG can be used as a shared resource within a multi-threaded application. However crossing the system boundaries of a process or a computer requires additional

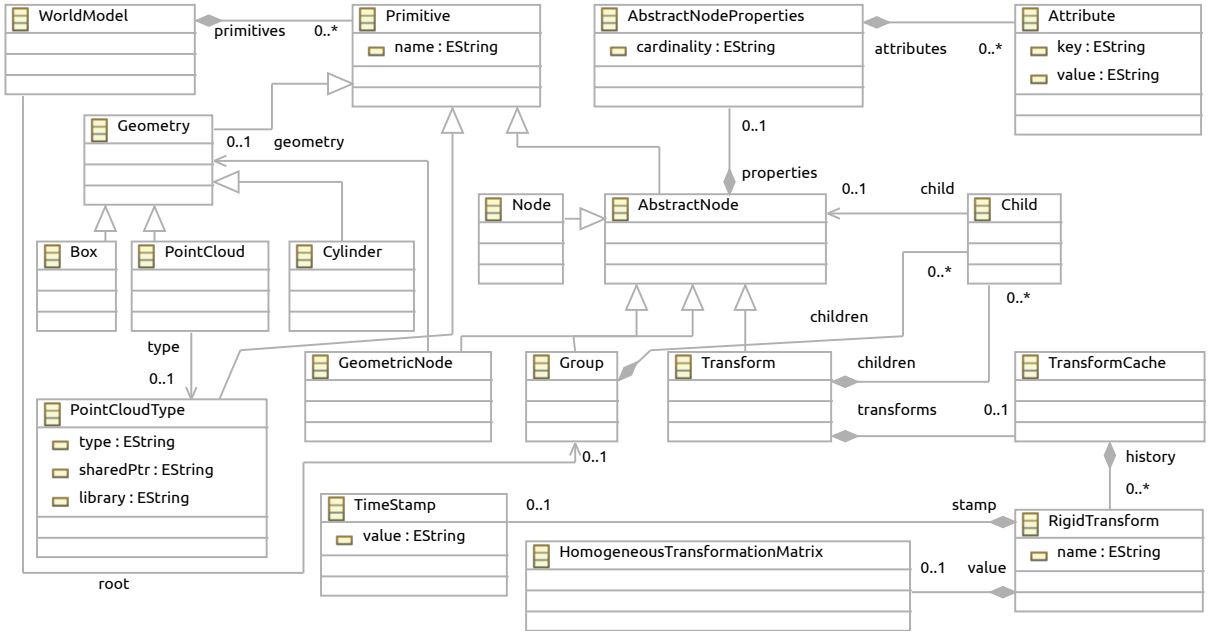


Fig. 1. Excerpt from Ecore model of the structural part of the world model. For the sake of readability some elements are not shown: the quantities for the geometries and transformation matrix are omitted and the `Mesh` definition is skipped because it is defined analogous to the `PointCloud` type.

communication mechanisms. Many component-based frameworks in robotics including ROS [13], OROCOS [14] and YARP [15] provide a communication layer for distributed components. These frameworks are mostly message-oriented and do not support a shared data structure like a world model well. Thus, we allow the RSG to create and maintain local copies of the scene graph [16]. Subsequent graph updates need to be encapsulated in the framework specific messages. Further details on the RSG world model and its primitives can be found in [1].

IV. A DSL for a Scene Graph based World Model

A. Choice of modeling framework

This work uses the Eclipse Modeling Framework (EMF) [8] as DSL framework. For two reasons: first, it allows one to make use of the Eclipse tool chain to generate an editor with syntax highlighting. Second, other robotic DSLs that already exist in this framework could be potentially re-used. A candidate is the geometric relations DSL [7]. The integration into the world model DSL is left as future work.

In addition to the proposed DSL, a model to text transformation is provided that generates code to be used in conjunction with the C++ implementation of the RSG which is part of the BRICS_3D library. Hence, this work mainly contributes to the M2 and M0 levels.

B. M2: DSL definition

The RSG-DSL for the scene graph based world model is defined with the Xtext grammar language [17]. The corresponding Ecore meta model representation as part of the EMF is completely generated from that Xtext definition. The RSG-DSL re-uses an existing DSL for units of measurements that is defined with Xtext as well. This is achieved via *grammar mixins*.

The central elements of the RSG-DSL are the node types as shown in Section III. As depicted in Fig. 1 the Ecore model represents the shared properties for all node types within the `AbstractNodeProperties` that can have a list of `Attributes`. An attribute is a key value pair. The `Group` and the `Transform` are the only node types that have *children* by referencing to the `AbstractNode`. The other two node types `Node` and `GeometricNode` are thus leaves in the scene graph.

The RSG-DSL identifies and references all node types by their names. This seems to be in conflict with the requirement that nodes have unique IDs but the description of a world model on M1 level can be seen as a generic template for a scene of an application [11]. The constraint of unique IDs has to hold on the M0 instance level and can be considered as an implementation detail. The world model implementation has facilities to provide and maintain unique IDs.

Each geometric data that can be contained in a `GeomtricNode` has its dedicated representation within the RSG-DSL. Special attention has to be paid to the `PointCloud` and `Mesh` types as legacy data types shall be supported on M0 level. The `PointCloudType` collects all necessary information to be able to generate code for any point cloud representation used in an application. The `Mesh` representation follows analogously. On the M0 level this variability is mapped to a template based class.

The temporal cache for the `Transform` node is modeled by the `TransformCache`. It consists of a list of `RigidTransforms` while a single entry is formed by a

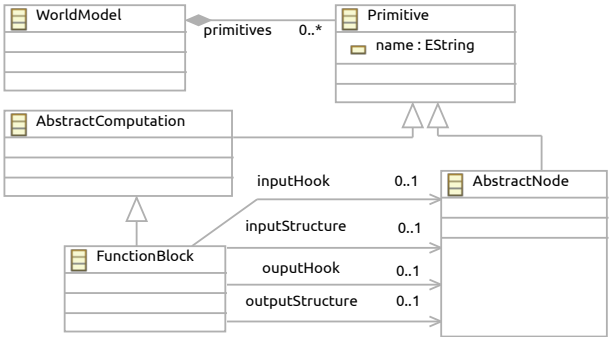


Fig. 2. Ecore model of function blocks for data processing. Input and output data is specified via *hooks* and *structure* definitions. Hooks describe where the data is located at run time, while structure definition describes how the data looks like.

`HomogeneousTransformationMatrix` and an associated `TimeStamp`. The values for the geometric data, the transformation matrix and the time stamps are accompanied with units of measurements.

The `FunctionBlock` model (cf. Fig. 2) to represent the behavioral aspect of the world model consists of four references to `AbstractNodes`: An `inputHook`, an `inputStructure`, an `outputHook` and an `outputStructure`. The *hooks* reefer to a subgraph at run-time that is to be consumed for further processing or it defines where to add the results of a computation to the scene graph. The *structure* property represents at design time the expected structure of a scene that is required for an encapsulated algorithm. For example, a function block that implements an algorithm for segmentation of point cloud data can have a `PointCloud` node as structural input. As output structure it provides *any number* of `Transform` nodes pointing to the centroids of the segmented point clouds. Each transform has a `PointCloud` as child node to represent a single segment. To be able to express such multiplicities in the input and output structures the DSL foresees a *cardinality* attribute that is available in the `AbstractNodeProperties`.

Function blocks can be used to create processing chains. All intermediate results of such a chain are stored in the scene graph. The input and output structures allows to check on the M1 level if the output of one function block matches as input for a successor function block. A trigger mechanism that can execute function blocks based on changes in the scene or based on signaling by other function blocks is planned as future work.

C. M0: Code generation

Xtend is used to realize the model to text transformation from the M1 to the M0 level. As the world model primitives are available in the RSG implementation the code generation for them is a straight forward mapping to the respective API calls. The RSG-DSL has no assumptions on the order of primitives. On the implementation level, children can not be added to parents that will be created afterwards. To overcome this hurdle the transformation uses a depth-first search based graph traversal for the model primitives to ensure correct order of creation.

Adding a new primitive with the help of the API will return a unique ID which will be kept in a variable that is labeled with the same name as in the model. These corresponding variables improve readability of the generated code on the one hand and keep the unique ID property on the other hand.

The primitives that are in the subgraph of the `root` node of the `WorldModel` will be stored in a *SceneSetup.h* file to represent the application specific scene. This file can be included and used within the application.

All `FunctionBlocks` result in dedicated header files for each generated interface. An implementation for a function block has to inherit from such an interface. This strategy is inspired by the *Implementation Gap Pattern* [18] and it separates generated code from hand-written code via inheritance.

V. Examples

To illustrate the capabilities of the RSG-DSL a set of examples on the M1 and the M0 level is given below.

A. A robot application scene

Listing 1 demonstrates an application scene that consists of a subgraph for a sensor and a kitchen table attached to the *scene objects* `Group`. The `root` keyword defines the application scene subgraph. For the sake of readability the structure for the robot carrying the sensor is omitted. The sensor `Group` is supposed to be the place where online sensor data will be hooked in. Note that the transform data is accompanied by units of measurements (cf. lines 20 to 22). In case of a moving sensor with respect to the world frame further transform data has to be inserted into the cache. Here the provided information given by the RSG-DSL can be seen as an initial value.

An excerpt of the resulting model to text transformation is presented in Listing 2. The respective API method invocations for *group1* `Group` and *worldToCamera* `Transform` are shown. Lines 2 to 4 indicate the mapping of M1 level node names to IDs on the M0 level.

Listing 1. Application scene setup represented with the RSG-DSL.

```
1  root rootNode // application scene
2
3  Group rootNode {
4    child group1
5    child worldToCamera
6  }
7
8  Group group1 {
9    Attribute ("name", "scene_objects")
10   child kitchenTable
11 }
12
13 Transform worldToCamera {
14   Attribute ("name", "wm_to_sensor_tf")
15   child sensor
16   transforms {
```

```
    RigidTransform t1 {
      stamp TimeStamp ( 0.0 s)
      value HomogeneousTransformationMatrix (
      [1.0, 0.0, 0.0, 0.0 m ],
      [0.0, 1.0, 0.0, 0.0 m ],
      [0.0, 0.0, 1.0, 1.0 m ],
      [0.0, 0.0, 0.0, 0.0   ])
    }
  }
}

Group sensor {
  Attribute ("name", "sensor")
}
```

B. Scene structure for a semantic entity

As an example for a semantic entity a table is defined in Listing 3. It relates the geometric parts into a scene structure. All legs have a spatial relation from the center of the `tablePlate` defined by the transform node that is a child of the `kitchenTable` group node. The example shows only one table leg but the other definitions follow analogously. The results are depicted in the Fig. 3 and Fig. 4. The used visualization functionality for the graph structure and the 3D visualization are part of the RSG implementation and demonstrate that the model to text transform of the example works as expected.

Listing 3. Kitchen table represented with the RSG-DSL.

```
Group kitchenTable {
  Attribute ("name", "kitchen_table")
  Attribute ("affordance", "pushable")
  child tablePlate
  child leg1tf
  child leg2tf
  child leg3tf
  child leg4tf
}

GeometricNode tablePlate {
  Attribute ("name", "table_plate")
  geometry tablePlateGeometry
}

Box tablePlateGeometry {
  sizeX 1.80 m
  sizeY 0.90 m
  sizeZ 5.0 cm
}

Box tabelLegGeometry {
  sizeX 0.1 m
  sizeY 0.1 m
  sizeZ 0.76 m
}

Transform leg1tf {
  Attribute ("name" , "plate_to_leg1_tf")
  child leg1geom
  transforms {
    RigidTransform t1 {
      stamp TimeStamp ( 0.0 s )
      value HomogeneousTransformationMatrix (
      [1.0, 0.0, 0.0, 0.85 m ] ,
      [0.0, 1.0, 0.0, 0.40 m ] ,
      [0.0, 0.0, 1.0, -0.38 m ] ,
      [0.0, 0.0, 0.0, 0.0   ])
    }
  }
}

GeometricNode leg1geom {
  Attribute ("name", "leg_1")
  geometry tabelLegGeometry
}

// The other three table legs
// are set up analogously.
```

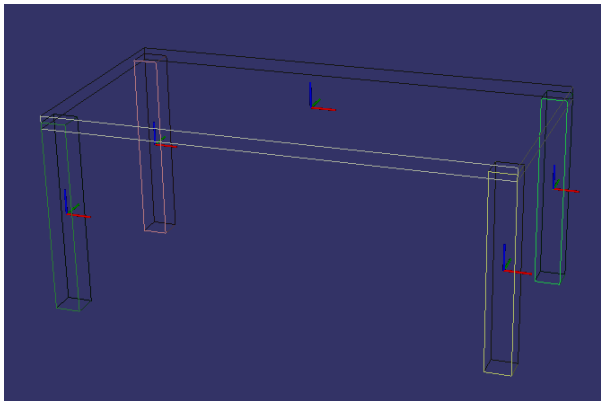


Fig. 3. 3D visualization of the kitchen table.

C. Interface definition for a function block

A `FunctionBlock` definition for a point cloud based segmentation algorithms is depicted in Listing 4. The input structure reefers to a point cloud node that contains an internal representation based on the Point Cloud Library (PCL) [19]. Input and output point clouds are of the same type as shown in lines 7 and 8. As output structure a *planes* `Group` node is specified that can have zero or more `Transforms` that are supposed to point to the centroids of the calculated point cloud segments. Line 21 reflects this variability by using the optional `cardinality` keyword. In this case the "*" terminal symbol has the semantics of *any number*. According to the `outputHook` in line 44 all results will be inserted to the scene graph as child node of the *sensor* node (cf. Section V-A). An implementation of the function block can be achieved with functionality offered by PCL for instance. Algorithmic details are beyond the scope of this paper. Other point cloud processing libraries could have been chosen as well. Whatever choice the application programmer has been made, it is explicitly represented in the model on the M1 level.

Listing 4. A function block represented with the RSG-DSL.

```
PointCloudType PointCloudPCL {
  type "pcl::PointCloud<PointType>"
  sharedPtr "pcl::PointCloud<PointType>::Ptr"
  library "pcl"
}
```

Listing 2. Excerpt from generated code for M0 level. Some comments and additional line breaks have been added after generation.

```
std::vector<rsg::Attribute> attributes; // Instantiation of list of attributes.
unsigned int rootNodeId;                // IDs correspond to names in model on M1 level.
unsigned int group1Id;
unsigned int worldToCameraId;
// [...]

/* Add group1 as a new node to the scene graph */
attributes.clear();
attributes.push_back(Attribute ("name", "scene_objects"));
wm->scene.addGroup(rootNodeId, group1Id, attributes); // group1Id is an output parameter
// [...]                                              // and returns a unique ID.

/* Add worldToCamera as a new node to the scene graph */
attributes.clear();
attributes.push_back(Attribute ("name", "wm_to_sensor_tf"));
brics_3d::IHomogeneousMatrix44::IHomogeneousMatrix44Ptr worldToCameraInitialTf(
  new brics_3d::HomogeneousMatrix44(  // Instantiation of HomogeneousTransformationMatrix primitive.
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    0.0 * 1.0, 0.0 * 1.0, 1.0 * 1.0  // Values are scaled to SI unit [m].
));

wm->scene.addTransformNode(rootNodeId, worldToCameraId, attributes, worldToCameraInitialTf,
    brics_3d::rsg::TimeStamp(0.0, Units::Second) // Value is scaled to SI unit [s].
);
```

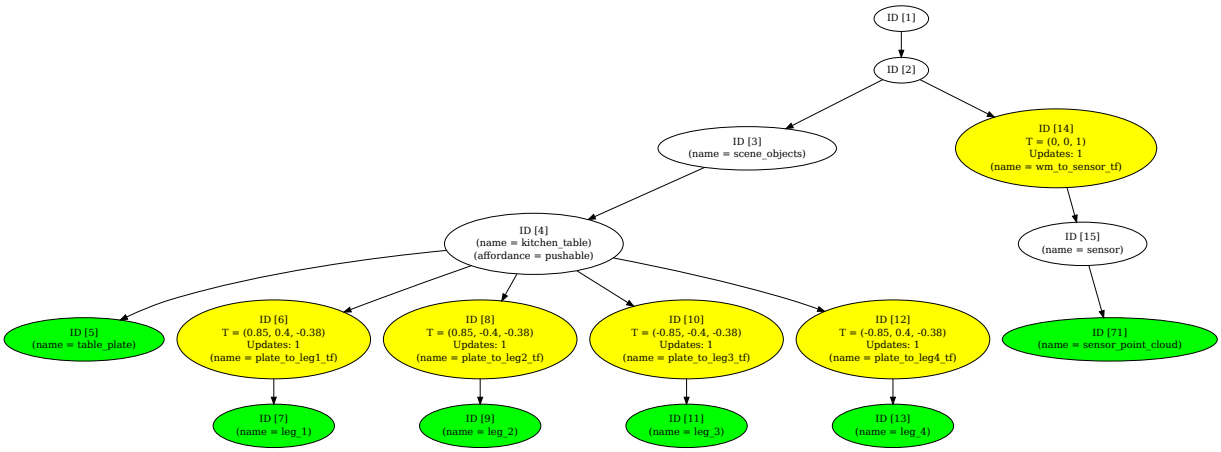


Fig. 4. Scene graph structure for the application scene including the kitchen table. Yellow nodes show `Transforms` while green nodes indicate `GeometricNodes`. The on M0 level generated IDs are shown in square brackets. Attached attributes are given in brackets. In addition the `Transform` nodes indicate the translational values T = (x, y, z) and the size of the temporal cache via the *Updates* field.

```
PointCloud inputCloud type PointCloudPCL
PointCloud planeCloud type PointCloudPCL

GeometricNode pointCloud {
  Attribute ("name", "point_cloud")
  geometry inputCloud
}

Group planes {
  Attribute ("name", "planes")
  child tfToPlaneCentroid
}

Transform tfToPlaneCentroid {
  cardinality *
  child horizontalPlane
  transforms {
    RigidTransform t1 {
      stamp TimeStamp ( 0.0 s)
      value HomogeneousTransformationMatrix (
      [1.0, 0.0, 0.0, 0.0 m ],
      [0.0, 1.0, 0.0, 0.0 m ],
      [0.0, 0.0, 1.0, 0.0 m ],
      [0.0, 0.0, 0.0, 0.0   ])
    }
  }
}

GeometricNode horizontalPlane {
  Attribute ("name", "plane")
  geometry planeCloud
}

FunctionBlock horizontalPlaneSegmentation {
```

```
  inputStructure pointCloud
  inputHook sensorPointCloud
  outputStructure planes
  outputHook sensor
}
```

VI. Conclusion

This work has presented the RSG-DSL: a DSL for a robotic world model based on the Robot Scene Graph (RSG). It is grounded in real behavior as code can be generated to be used with an API for an existing implementation of the RSG approach. The RSG-DSL allows to express (a) application specific scene setups, (b) semantic scene structures and (c) inputs and outputs for the function blocks which are a part of the world model approach.

The RSG-DSL makes a contribution to improve the robot development work flow as world model aspects can be explicitly represented in a model-driven tool chain. Thus, a developer can create a robotic application quicker and less error prone.

Future work will include extension of the RSG-DSL approach by multiple levels of detail representations for geometries, uncertainty representations and trigger entities for function blocks. Currently the scene setup definition is centered around a single robot system. Language support for distributed and multi-robot applications are important improvements for the proposed DSL. The inclusion of other existing DSLs like the geometric relations DSL is a promising research direction with the goal of contributing to a *robotic DSL* that can be composed of a set of languages representing various robotic subfields like world modeling, planning, perception, reasoning or coordination.

Acknowledgements

The authors acknowledge the support from the KU Leuven Geconcerteerde Onderzoeks-Acties *Model based intelligent robot systems* and *Global real-time optimal control of autonomous robots and mechatronic systems*, and from the European Union's 7th Framework Programme (FP7/2007–2013) projects *BRICS* (FP7-231940), *ROSETTA* (FP7-230902), *RoboHow.Cog* (FP7-288533), and *SHERPA* (FP7-600958).

References

[1] S. Blumenthal, H. Bruyninckx, W. Nowak, and E. Prassler, "A Scene Graph Based Shared 3D World Model for Robotic Applications," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Karlsruhe, Germany*, 2013.

[2] J. Elfring, S. van den Dries, M. van de Molengraft, and M. Steinbuch, "Semantic world modeling using probabilistic multiple hypothesis anchoring," *Robotics and Autonomous Systems*, vol. 61, no. 2, pp. 95 – 105, 2013.

[3] K. Wurm, D. Hennes, D. Holz, R. Rusu, C. Stachniss, K. Konolige, and W. Burgard, "Hierarchies of octrees for efficient 3D mapping," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011, pp. 4249–4255.

[4] S. Blumenthal, "BRICS_3D Documentation pages," 2013. [Online]. Available: http://www.best-of-robotics.org/brics_3d/

[5] International Organization for Standardization, "ISO/IEC 19502: International Standard: Information technology - Meta Object Facility (MOF)," 2005.

[6] M. Klotzbücher, R. Smits, H. Bruyninckx, and J. De Schutter, "Reusable hybrid force-velocity controlled motion specifications with executable Domain Specific Languages," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, 2011, pp. 4684–4689.

[7] T. De Laet, W. Schaekers, J. de Greef, and H. Bruyninckx, "Domain Specific Language for Geometric Relations between Rigid Bodies targeted to robotic applications," *CoRR*, vol. abs/1304.1346, 2013.

[8] Eclipse Modeling Framework Project, "Eclipse Modeling Framework Project (EMF)," 2013. [Online]. Available: http://www.eclipse.org/modeling/emf/

[9] S. Schneider and N. Hochgeschwender, "Towards a Declarative Grasp Specification Language," in *Workshop on Combining Task and Motion Planning of the IEEE International Conferenceon Robotics and Automation*, 2013.

[10] J. Haist and P. Korte, "Adaptive streaming of 3D-GIS geometries and textures for interactive visualisation of 3D city models," 2006.

[11] M. Lenk, A. Vitzthum, and B. Jung, "Model-driven iterative development of 3D web-applications using SSIML, X3D and JavaScript," in *Proceedings of the 17th International Conference on 3D Web Technology*. ACM, 2012, pp. 161–169.

[12] R. Smith, M. Self, and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," *Autonomous robot vehicles*, vol. 1, pp. 167–193, 1990.

[13] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009.

[14] H. Bruyninckx, "Open robot control software: the orocos project," in *IEEE International Conference on Robotics and Automation*, vol. 3, 2001, pp. 2523 – 2528.

[15] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: Yet another robot platform," *International Journal on Advanced Robotics Systems*, vol. 3, no. 1, pp. 43–48, 2006.

[16] M. Naef, E. Lamboray, O. Staadt, and M. Gross, "The blue-c distributed scene graph," in *Proceedings of the workshop on Virtual environments 2003*. ACM, 2003, pp. 125–133.

[17] Xtext project, "Xtext - Language Development Made Easy! - Eclipse," 2013. [Online]. Available: http://www.eclipse.org/Xtext/documentation.html

[18] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[19] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

Towards a Robot Perception Specification Language

Nico Hochgeschwender, Sven Schneider, Holger Voos, and Gerhard K. Kraetzschmar

I. Introduction

Domestic service robots such as PR2 [1] and Care-O-bot 3[1] must be able to perform a wide range of different tasks ranging from opening doors [1] and making pancakes [2] to serving drinks [3]. A crucial precondition to achieve such complex tasks is the ability to extract *task knowledge* about the world from the data perceived through the robot's sensors. Examples are the localization of humans [4] for navigation and interaction purposes, or the detection and recognition of objects in grayscale images for the sake of manipulation by the robot. To perceive all the knowledge needed to safely and robustly perform a task, robots are equipped with a set of heterogenous sensors such as laser range finders, ToF cameras, structured light cameras and tactile sensors which provide different types of data such as distance measurements, depth images, $3D$ point clouds, and $2D$ grayscale or color images. To structure all the required processing steps on this data so called *Robot Perception Architectures* (RPAs) are required (see also Fig. 1). In general, RPAs are composed of functional components processing sensory input to output which is relevant for the task in hand. Thereby, heterogenous algorithms such as filters and feature detectors are integrated in components which are then assembled to make up an RPA [5]. However, despite recent algorithmic advancements in the field of vision and perception, the development of RPAs, designed to extract meaning out of the enormous amount of data, is still a complex and challenging exercise. There is little consensus on either how such an architecture is best designed for any particular task or on how to organize and structure robot perception architectures in general, so that they can accommodate the requirements for a *wide range* of tasks.

In this paper we present our work in progress towards a *Robot Perception Specification Language (RPSL)*. *RPSL* is a domain-specific language and its purpose is twofold. First, to provide means to specify the expected result (*task knowledge*) of a RPA in an explicit manner[2]. Second, to initiate the (re)-configuration process of an RPA based on the provided specification. Here, we focus on the first objective and discuss the core language concepts which have been composed in *RPSL*, namely the object, spatial, timing, dependency and composition domain.

Nico Hochgeschwender, Sven Schneider, and Gerhard Kraetzschmar are with the Department of Computer Science, Bonn-Rhein-Sieg University of Applied Sciences, Germany. Email: forename.surname@h-brs.de Nico Hochgeschwender and Holger Voos are with the Research Unit in Engineering Sciences, University of Luxembourg, Luxembourg. Email: holger.voos@uni.lu

[1] http://www.care-o-bot-research.org/

[2] Referring to the right-hand side of Fig. 1

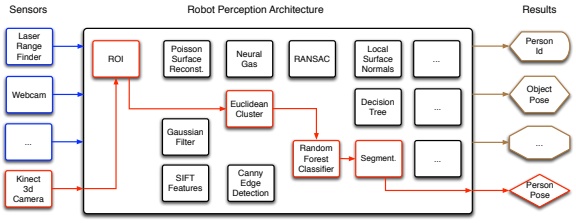


Fig. 1. Elements making up the design space of a robot perception architecture: i) heterogenous sets of sensors (blue boxes), ii) computational components (black boxes), and iii) task-relevant information and knowledge (brown boxes). The path which is visualized in red shows an instance of an existing RPA described in Hegger *et al.* [4]. *RPSL* is used to specify the task knowledge visualized in brown boxes on the right hand side.

II. Problem Statement and Motivation

Currently, robot perception architectures are developed by domain experts during design time. The design is significantly influenced by many decisions, which often remain implicit. These design decision concern the robot *platform*, the *tasks* the robot should perform, and the *environment* in which the robot operates. Some exemplary design decisions include:

- The sensor configuration (e.g., resolution or data frequency) of a particular sensor according to environment and task specifications.
- The general composition of an RPA, including the selection, configuration and organization of computational components (implementing the core sensor processing functionalities such as filters, classifiers, etc.) such that task and environment requirements are met.
- The configuration of a specific composition of an RPA for solving a particular task-relevant perception problem, e.g. determining the pose of a human. The pertains to particular dynamic connection of RPA components.

As long as task, environment, and platform specifications remain as assumed during design time, the RPA will operate properly. However, if an event concerning robot capabilities, task requirements, or environment features occurs, systematically ensuring an appropriate reaction by the RPA is a great challenge. Generally speaking, the vast majority of RPAs is static and inflexible and it is not possible

- to reconfigure parameters of computational components (e.g., the σ value of a Gaussian filter) during run time,
- to execute complete processing chains in a demand-driven manner,

- and to modify and reconfigure robot perception processing chains during run time.

To provide RPAs with the ability to reconfigure their structure and behavior one needs to model the design decisions mentioned above in an explicit and computable (during runtime available) manner. First of all the desired *task knowledge* needs to be specified. Depending on the functional component (e.g., manipulation, grasping, or decision-making) which requires the knowledge and the current task at hand this knowledge differs substantially. For instance, a decision-making component might be interested in the existence of an object whereas a grasping and manipulation component demands more sophisticated information such as spatial dimensions and shapes of an object. In both cases means to express the desired task knowledge are required. To the best of our knowledge in robotics there is no language available which allows us to encode such specifications. We observed that very often ad-hoc solutions e.g., in the form of message definitions (provided by the underlying robot software framework) are used which lack expressiveness.

III. RPSL: Robot Perception Specification Language

In the following we present the current status of *RPSL*. We identify first language requirements and then describe the different domain concepts which are part of the language. Those concepts have been identified through a domain analysis of existing RPAs and their application context in real-world scenarios.

A. Requirements and Assumptions

RPSL is aimed to be a specification language. Therefore, the language is not executable. Interestingly, from a planning point of view the specifications are comparable with goal specifications in the Planning Domain and Definition Language (PDDL) [6]. Similarly to PDDL a specification language for the perception domain should be independent of the underlying RPA just as PDDL is independent of concrete task planners. To be usable for a wide range of applications and systems, *RPSL* should be independent of

- the type of sensor data processed by the RPA, and
- the type of functional components which are assembled to make up an RPA.

To enable reuse and exchangeability of the domain concepts realized in *RPSL* (e.g., through concrete language primitives and abstraction) they should be orthogonal to each other as far as possible. Further, we assume that an environment is not actively observed (e.g., no active perception which involves movements of the robot). However, many so called table top situations in robotics are covered with our current status of *RPSL*.

B. General Approach

Based on our domain analysis we derived several core domain concepts described in the following. To model the domains we apply a model-driven engineering approach using the Eclipse Modeling Framework (EMF) [7]. Here, each domain is specified in the form of an Ecore model. Based on the Ecore models we developed an external domain-specific language (DSL) with Xtext [8]. As *RPSL* is work in progress we use the external DSL mainly to validate the domain concepts with experts. The next sections describe the domains and features that need to be captured by *RPSL* in more detail.

```
myConcepts: Namespace {
  myBox: Concept {
    use_domain Size
    p: Polytope {
      Point(Size.Height, 20mm)
      Point(Size.Height, 40mm)
      Point(Size.Width, 20mm)
      Point(Size.Width, 40mm)
      Point(Size.Length, 100mm)
    }
  }
}
```

Fig. 2. Concept definition of a box.

C. Object Domain

As exemplified in Fig. 1 and discussed in Section II there is a huge variability in the kind of *task knowledge* potentially provided by RPAs. Ranging from diverse objects such as persons and objects of daily use such as cups, bottles, and door handles to the information about these objects themself such as center of mass, poses, color and shapes. Here, the challenge is to use a representation which enables us to model the information about objects on various levels of abstraction. Ranging from raw sensor data to feature descriptors and high-level object information such as size. In *RPSL* the object domain is based on Conceptual Spaces (CS) which is a knowledge representation mechanism introduced by Gärdenfors [9]. A conceptual space is composed of several (measurable) quality dimensions. A concept in a conceptual space is a convex region in that space. Points (also called knoxels) in a conceptual space represent concrete instances (objects) of a concept. To decide whether an instance belongs to one concept or to another we can apply similarity measures such as Euclidean distances. In Fig. 2 an example is shown. Here, a `Concept` called *myBox* is specified. The concept belongs to the `Namespace` *myConcepts* which is simply a mechanism to organize different concepts as known in general-purpose programming languages such as Java or C++. The concept *myBox* uses the `Domain` *Size* which is composed of three quality dimensions, namely *Height*, *Width*, and *Length*. In *RPSL* quality dimensions with different scales such as continuous or ordinal scales are supported. A `Polytope` is further used to model the "borders" of the concept *myBox*. For instance, every box belonging to the concept *myBox* needs to have a height between $20mm$ and $40mm$. In contrast to the Conceptual Space Markup Language (CSML) introduced by Adams and Raubal [10] we use polytopes instead of a set of inequalities to define the concept region as they are easier to model. To enrich the concept *myBox* we

```
myConcepts: Namespace {
  myBox: Concept {
    use_domain Size
    use_domain RGB
    p: Polytope {
      // ...
      Point(RGB.Red, 0)
      Point(RGB.Green, 0)
      Point(RGB.Blue, 100)
      Point(RGB.Blue, 140)
    }
  }
}
```

Fig. 3. Concept definition of a box with color information.

```
use Namespace myConcepts

darkBlueBox: Prototype {
  use_concept myConcepts.myBox
  v: Values {
    // ...
    Point(myBox.RGB.Blue, 139)
  }
}
```

Fig. 4. Prototype definition of a dark blue box.

simply refer to another domain. For instance, in Fig. 3 the concept *myBox* is enriched with color information using the RGB color coding which includes three quality dimensions, namely, *Red*, *Green*, and *Blue*. This approach allows us to model very expressive concepts as we can reuse existing domains and corresponding quality dimensions. Once concepts are defined we can model concrete instances or speaking in the conceptual space terminology: "prototypes". In Fig. 4 a `Prototype` *darkBlueBox* is modeled. Instead of defining ranges as in the concept definition, prototypes have single values per quality dimension.

D. Spatial Domain

Very often it makes sense to specify the required object information with respect to the spatial surrounding. Assuming an egocentric view of the robot one could model for instance objects through spatial operators such as "behind", "next to", and "right of". In particular, for manipulation tasks it is crucial to have information not only about the object to manipulate, but also about their spatial surrounding in order to plan motions and to check for collisions. Currently, we investigate which spatial model we want to include in *RPSL* such as the region connection calculus (RCC) [11].

E. Timing Domain

With the timing domain we intend to enrich specifications about the *"when"*. More precisely, in many situations it is important to retrieve information about objects within a certain time frame e.g., to avoid a stucking robot behavior. We use the notion of a deadline to encode a particular point in time by which the specified information should be available. For instance, specification *s5* shown in Fig. 5 is enriched with a `Deadline` of $3s$. Here, `Deadline` can be parameterized with the value and an time unit. From an implementation point of view once the specification is received by the RPA it will obtain a time stamp which will be used to cope with the deadline. This imposes a certain protocol between the component which emits the specification and the RPA which will not be discussed here. In future we intend to extend the timing domain with policies allowing to model strategies with missed deadlines (e.g., "when deadline X is missed try to retrieve information Y or repeat it once").

F. Dependencies

Another feature in *RPSL* is to model dependencies among specifications. That is some information is required before some other information is available. In Fig. 5 specification *s4* is composed of two specifications which have a dependency. First, the amount of the *darkBlueBox* is retrieved and then the *Pose* of the *darkBlueBox* is retrieved. To model these situations the dependency meta-model is based on the concept of a directed acyclic graph (DAG). Interestingly, in the past we used the same dependency meta-model to model the sequence of component deployment [12] and robot action plots [13].

G. Composition Domain

The composition domain composes the previous domains in order to model a valid and complete specification. Some concepts such as timing and dependencies are optional whereas the object domain is mandatory. In Fig. 5 some examples are shown. First, the `Namespace` *myConcepts* is used. Further, in the first specification one is interested in the amount of objects (visible in the current scene) belonging to the concept *myBox* with certain properties concerning length and width. Here, *Amount* itself is a concept with one quality dimension, namely an ordinal integer scale. As seen in the example the syntax is inspired by SQL with the difference that the data model is based on Conceptual Spaces. Similarly to SQL we support logical operators such as `AND` and `OR` as well as relational operators such as $==$, $>$ and $<=$ known from general-purpose programming languages. In the second specification *s2* the previously modeled prototype *darkBlueBox* is used. After the `where` statement a condition is modeled. Here, the condition is that only objects which look exactly like the *darkBlueBox* (similarity measured with Euclidean distance) are counted. The idea is that with the `Similarity` operator several similarity measures are supported and that we can balance the expected result according to the features provided by the measure. In future we intend to support also weighting factors which can be applied to increase or decrease the importance of quality dimensions for the similarity measure.

IV. Conclusion

We presented the work in progress of using domain-specific languages for specifying robot perception architectures. Assessing the DSLRob workshop series showed

```
use Namespace myConcepts

s1: Specification {
  d: Data {
    get Amount from myBox where myBox.Size.Width >= 20mm and myBox.Size.Length > 100mm
  }
}

s2: Specification {
  d: Data {
    get Amount from darkBlueBox where Similarity(EuclideanDistance) == 0
  }

s3: Specification {
  d: Data {
    get Pose from darkBlueBox where Similarity(EuclideanDistance) == 0
  }
}

s4: Specification {
  dg: DependencyGraph {
    s2 before s3
  }
}

s5: Specification {
  d: Data {
    get Amount from darkBlueBox where Similarity(EuclideanDistance) == 0 ensure Deadline(3s)
  }
}

}
```

Fig. 5. Some example specifications.

that *RPSL* is the first attempt to use DSLs in the subdomain of robot perception. Even though, *RPSL* is work in progress, it helped already to identify and break down the crucial domains which are involved in specifying the result of RPAs. To achieve the second objective of our language, namely the initialization of a (re)-configuration based on the specification we are currently implementing a use case which is based on simple table top scene.

ACKNOWLEDGEMENT

Nico Hochgeschwender received a PhD scholarship from the Graduate Institute of the Bonn-Rhein-Sieg University of Applied Sciences which is gratefully acknowledged. Furthermore, the authors gratefully acknowledge the on-going support of the Bonn-Aachen International Center for Information Technology.

REFERENCES

[1] W. Meeussen, M. Wise, S. Glaser, S. Chitta, C. McGann, P. Mihelich, E. Marder-Eppstein, M. Muja, V. Eruhimov, T. Foote, J. Hsu, R. B. Rusu, B. Marthi, G. Bradski, K. Konolige, B. P. Gerkey, and E. Berger, "Autonomous door opening and plugging in with a personal robot," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2010.

[2] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth, "Robotic roommates making pancakes," in *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2011.

[3] T. Breuer, G. R. G. Macedo, R. Hartanto, N. Hochgeschwender, D. Holz, F. Hegger, Z. Jin, C. Müller, J. Paulus, M. Reckhaus, J. A. Álvarez Ruiz, P. G. Plöger, and G. K. Kraetzschmar, "Johnny: An autonomous service robot for domestic environments," *Journal of Intelligent Robotic Systems (JIRS)*, vol. o.A., pp. 245–272, 4 2012.

[4] P. G. P. Frederik Hegger, Nico Hochgeschwender and G. K. Kraetzschmar, "People Detection in 3d Point Clouds using Local Surface Normals," in *Proceedings of the 16th RoboCup International Symposium*, ser. Lecture Notes in Computer Science. Mexico City, Mexico: Springer, June 2012, to appear.

[5] G. Biggs, N. Ando, and T. Kotoku, "Rapid data processing pipeline development using openrtm-aist," in *System Integration (SII), 2011 IEEE/SICE International Symposium on*, 2011, pp. 312–317.

[6] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl - the planning domain definition language," Yale Center for Computational Vision and Control,, Tech. Rep. TR-98-003, 1998.

[7] Eclipse Modeling Framework Project, "Eclipse Modeling Framework Project (EMF)," 2013. [Online]. Available: http://www.eclipse.org/modeling/emf/

[8] Xtext project, "Xtext - Language Development Made Easy! - Eclipse," 2013. [Online]. Available: http://www.eclipse.org/Xtext/documentation.html

[9] P. Gärdenfors, *Conceptual spaces - the geometry of thought*. MIT Press, 2000.

[10] B. Adams and M. Raubal, "Conceptual space markup language (csml): Towards the cognitive semantic web." in *ICSC*. IEEE Computer Society, 2009, pp. 253–260.

[11] D. A. Randell, Z. Cui, and A. Cohn, "A Spatial Logic Based on Regions and Connection," in *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, B. Nebel, C. Rich, and W. Swartout, Eds. San Mateo, California: Morgan Kaufmann, 1992, pp. 165–176.

[12] N. Hochgeschwender, L. Gherardi, A. Shakhimardanov, G. Kraetzschmar, D. Brugali, and H. Bruyninckx, "A model-based approach to software deployment in robotics," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).*, 2013.

[13] M. Reckhaus, N. Hochgeschwender, P.-G. Plöger, and G. K. Kraetzschmar, "A platform-independent programming environment for robot control," in *Proceedings of the 1st International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob) held at the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

Towards Automatic Migration of ROS Components from Software to Hardware

DSLRob 2013 — Work-in-progress

Anders Blaabjerg Lange, Ulrik Pagh Schultz and Anders Stengaard Soerensen

I. Introduction

The use of the ROS middleware is a growing trend in robotics in general, in particular in experimental branches of robotics such as modular robotics, fields robotics, and the vast area of cyber-physical systems (for example applied to welfare technology). Our main area of interest is in experimental robotics and cyber-physical systems. When building "robot controllers" for the aforementioned systems there are numerous suitable technological platforms. Given specific requirements we can choose an appropriate standardized approach, for example emphasizing flexibility and ease of development by using a generic middleware — such as ROS — or emphasizing real-time performance and direct hardware access by using approaches based on dedicated, embedded hardware. So far ROS and hard real-time embedded systems have however not been easily uniteable while retaining the same overall communication and processing methodology at all levels.

In this paper we present an approach aimed at tackling the schism between high-level, flexible software and low-level, real-time software. The key idea of our approach is to enable software components written for a high-level publish-subscribe software architecture to be automatically migrated to a dedicated hardware architecture implemented using programmable logic. Our approach is based on the Unity framework, a unified software/hardware framework based on FPGAs for quickly interfacing high-level software to low-level robotics hardware. The vision of Unity is to enable non-expert users to build high-quality interface and control systems using FPGAs and to interface them to high-level software frameworks, thereby providing a framework for speeding up and increasing innovation in experimental robotics. This paper presents the overall vision and the initial work on the implementation of an architecture supporting a generative approach, based on a declarative specification of how software components are mapped to a hardware architecture; the actual language design is left as future work.

II. Context: Unity and FPGAs

The traditional approach to building a control system in experimental robotics is mainly based on microcontrollers (MCU's) and PC's. This approach has numerous advantages, mainly: (1) developers are familiar with the programming methodology; (2) good tools, libraries and frameworks from commercial vendors and the open-source community; and (3) the availability of cheap and simple MCU-based systems like the Arduino, as well as more powerfull ARM based systems. Despite the advantages of this approach, there are also inherent limitations to the sequential-style processing and fixed hardware (HW) architecture, which can significantly limit reuse of HW as well as real-time capabilities, design freedom and flexibility.

We prefer FPGAs and hybrid FPGA-MCU SoC systems over pure MCUs: we find FPGAs superior to MCUs in many performance areas relevant to experimental robotics, except for price and library support. FPGAs can provide deterministic hard real-time performance no matter the complexity or scale of the implemented algorithms [1], [2], [3]. On an FPGA the architecture is designed by the developer, providing increased flexibility that can reduce the need for costly software abstractions on higher levels [4], [5], [6], [7] and reduce or eliminate the need for external support logic. FPGAs are however not commonly used; we believe the reason to be partly historical: people stick to technologies they know. Moreover, FPGAs suffer from a lack of good, open-source, vendor-independent HDL-component libraries suited for robotics, and a high degree of complexity associated with FPGA programming, caused partly by complex tools and a different programming methodology compared to the traditional Von-Neumann style.

We have proposed the Unity framework as a means to facilitating FPGA-based development for experimental robotics [8], [9]. Unity is an open-source framework consisting of reference HW designs, gateware (GW, VHDL) and SW libraries, all targeted at providing a complete framework for easy development, with standard cases covered by model-based code generation of all the necessary FPGA GW and PC SW needed to interface electronics with a high-level software framework. The Unity framework is a work-in-progress: The modular HW designs include single nodes, distributed nodes, sensor interfaces and generic motor controllers. On the GW side we have a growing library of VHDL modules, including servo- and brushless DC motor controllers, a real-time network based on a shared memory model, a complete FPGA-based real-time operating system [10], as well as a modular and reconfigurable FPGA-PC interface called Unity-Link [9]. The use of Unity compared to a traditional MCU-based approach, exemplified with a PC connected to low-

A. B. Lange, U. P. Schultz and A. S. Soerensen are with the Maersk McKinney Moeller Institute, University of Southern Denmark, Odense, Denmark (e-mail: {anlan, ups, anss}@mmmi.sdu.dk)

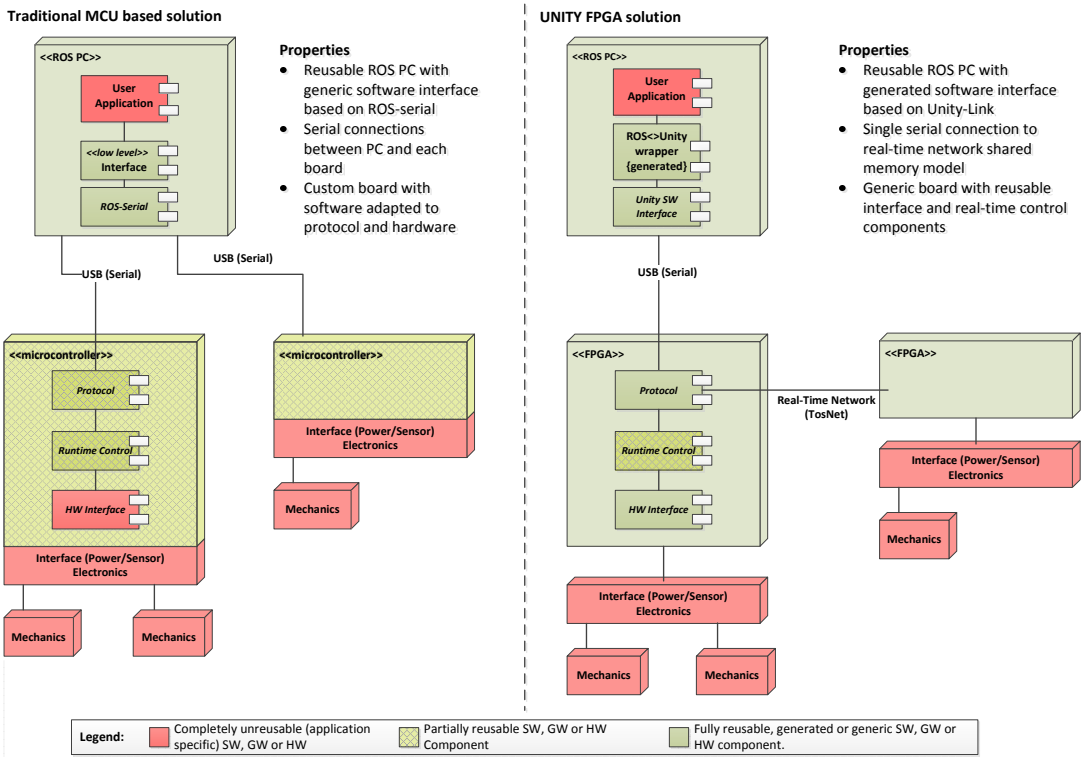


Fig. 1. Unity Link compared to a traditional MCU-based architecture (UML 2.0 component diagram notation)

level hardware using ROS-serial, is illustrated in Figure 1. We believe that a generic FPGA or FPGA-MCU SoC based module will be more flexible and therefore more easily reused for various tasks, compared to a standard off-the-shelf MCU system, since the various hardware interfaces needed are decoupled from (i.e., not locked to) specific pin locations, and therefore virtually only the pin count limits the number and types of interfaces that are possible when using programmable logic. Unity is an evolution of the TosNet framework, which is the basis for the real-time network and other specific components [4], [5], [6], [7].

III. Automatic Migration of ROS Components to FPGAs

We are currently investigating the idea of automatically migrating networks of ROS components[1] to our FPGA-based architecture. The Unity framework already provides a standardized platform on which gateware components can be interconnected, and Unity-Link provides automatic integration on ROS components with gateware components using a publish-subscribe infrastructure [9]. There is however no support for migrating a ROS component, or a set of ROS components, from the PC to the FPGA, without completely reimplementing the functionality of each of the components, and furthermore using the Unity framework to connect them internally on the FPGA. Note that we are not concerned with dynamic migration: we simply want to make it easy for the developer to statically change the deployment of functionality between the flexible PC platform and the real-time FPGA platform.

We propose that migration of a given ROS component from the PC-based platform to the FPGA can be done by recompiling the component to run on either a softcore or hard-IP CPU embedded in the FPGA. The HartOS real-time operating system [10] will be used to execute the threads of the component and to handle external events. A substrate that provides the ROS API and a few selected parts of the standard POSIX API[2] will be used on the embedded CPU, enabling a ROS component e.g. implemented in C++ to execute on the CPU after a simple recompilation. Publish-subscribe messages can be routed between the CPU and a PC running ROS using Unity-Link. A high performance hard-IP CPU, like e.g. the dual-core ARM-A9 in a Xilinx Zynq device, could as a second option also run a full linux system with ROS, and thereby support native (non-recompiled) ROS components. By providing the same memory-mapped publish/subscribe and service-call IP interfaces on both the small softcore and Hard-IP CPU's, no matter the software

[1]Throughout this paper we consistently use the term "ROS component" to refer to ROS nodes: we believe our approach is applicable to other component-based middlewares as well, and hence prefer the technology-independent term "component."

[2]Only a small subset of the POSIX API will be relevant, as well as feasible for a processing system utilizing the HartOS kernel. We assume our approach is primarily relevant for ROS components having a fairly small amount of interaction with the operating system.

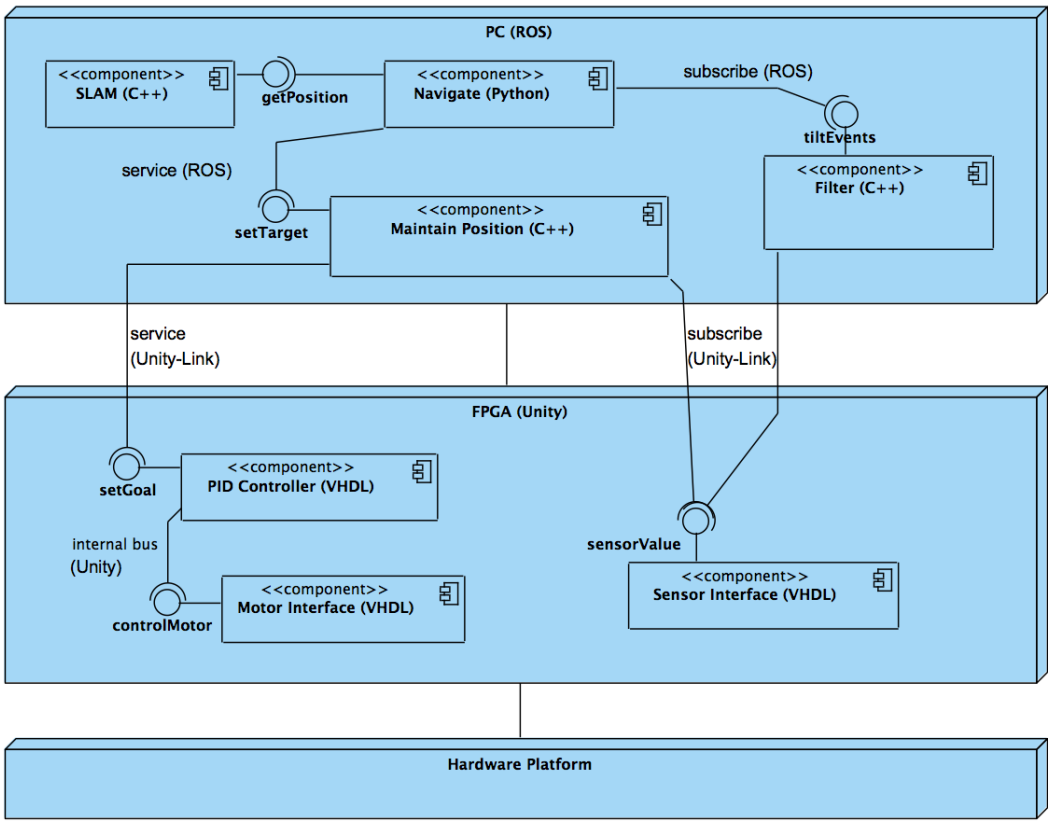


Fig. 2. Example: Robot software before migration of selected components, part of the control loop is implemented in ROS. (UML 2.0)

environment executed on them, Unity will allow both high and low performance processors, and PC's using Unity-link, to communicate with GW components directly utilizing ROS' own communication paradigm, thereby enabling easy migration between execution paradigms.

A set of ROS components that communicate using publish-subscribe can similarly be migrated to the FPGA. Each component is placed on a softcore CPU, depending on the performance requirements they can be placed on the same or different CPUs. If they are placed on the same CPU, HartOS is used for scheduling CPU-time between the components, and communication can be performed directly between the components (taking care to preserve communication semantics). If components are placed on different CPUs, a shared memory component is used to propagate publish/subscribe messages between the nodes: each topic uses a specific address in the shared memory, enabling a complete decoupling of the execution of publishers and subscribers. Service calls can be handled similarly, however rather than using a shared memory, a generic address-data bus can be used to provide a point-to-point connection between components that need to communicate.

As an example, consider a first revision of the robot software architecture for a two-wheeled balancing robot shown in Fig. 2. Low-level control and hardware interfacing is done in the FPGA using the Unity framework, and consists of low-level hardware interface components and a generic PID controller. Unity-Link connects low-level control and sensor interfaces to ROS using publish/subscribe and service calls. High-level control is implemented in ROS, and concerns navigation, movement, and balancing of the robot. Real-time operation of the "maintain position" component is ensured by using a suitably fast PC. Now assume that — although initial experiments showed that this worked fine — after experimenting with the robot in a realistic scenario it is found that control is unstable because real-time deadlines are sometimes missed. To solve the issue using our approach, the "maintain position" component is moved to a softcore CPU on the FPGA, as illustrated in Fig 3. Moreover, due to the use of standard interfaces that are interoperable between ROS and the FPGA, the software filter component is transparently replaced by a functionally equivalent gateware component from the Unity library. All components on the FPGA execute in hard real-time, making control of the robot predictable.

IV. Discussion and Status

The migration is intended to be automatic, in the sense that given a declarative specification of how a set of ROS components should be mapped to a real-time architecture, our system will generate substrate code, configuration files, and VHDL components such that the ROS components can be directly recompiled to run on the FPGA. This declarative

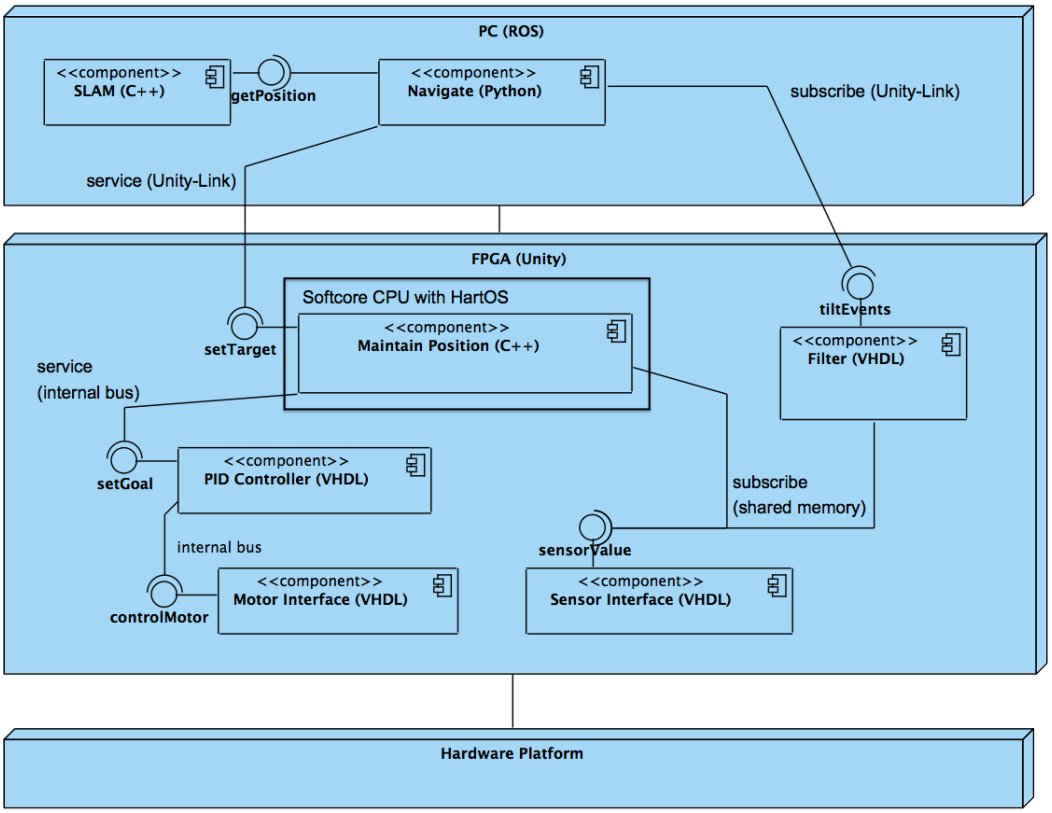


Fig. 3. Example: Robot software after migration of selected components, the "maintain position" and "filter" components have been migrated to the FPGA. (UML 2.0)

specification will thus need to model which components are to be deployed to which softcore CPU, how much time is to be assigned to each thread of each component, and how the components are to communicate with each other and with the rest of the ROS system.

We are currently extending the Unity framework to support multiple ROS components executing and communicating in real-time on one or more softcore CPUs on one or more FPGAs connected by a real-time network. Once this framework is complete, we will augment it with a model-based code generator than can automatically generate the complete set of code artifacts needed to support the execution of the ROS components on the FPGA. We expect that the task of implementing the framework and corresponding generator is significantly reduced by building on top of the standardized Unity architecture and using Unity Link to interface the FPGA-based ROS components to the rest of the ROS system.

References

[1] A. Fernandes, R. Pereira, J. Sousa, A. Batista, A. Combo, B. Carvalho, C. Correia, and C. Varandas, "HDL Based FPGA Interface Library for Data Acquisition and Multipurpose Real Time Algorithms," *Nuclear Science, IEEE Transactions on*, vol. 58, no. 4, pp. 1526–1530, Aug. 2011.

[2] M. Pordel, N. Khalilzad, F. Yekeh, and L. Asplund, "A component based architecture to improve testability, targeted FPGA-based vision systems," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, May 2011, pp. 601–605.

[3] S. Toscher, T. Reinemann, and R. Kasper, "An adaptive FPGA-based mechatronic control system supporting partial reconfiguration of controller functionalities," in *Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on*, June 2006, pp. 225–228.

[4] S. Falsig and A. Soerensen, "Tosnet: An easy-to-use, real-time communications protocol for modular, distributed robot controllers," in *Robot Communication and Coordination, 2009. ROBOCOMM '09. Second International Conference on*, March 31–april 2 2009, pp. 1–6.

[5] ——, "An FPGA based approach to increased flexibility, modularity and integration of low level control in robotics research," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, Oct. 2010, pp. 6119–6124.

[6] A. Soerensen and S. Falsig, "A system on chip approach to enhanced learning in interdisciplinary robotics," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, Oct. 2010, pp. 4050–4056.

[7] R. Ugilt, A. S. Soerensen, and S. Falsig, *A step toward 'plug and play' robotics with SoC technology*. World Scientific, 2010, ch. 52, pp. 415–422. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/9789814329927_0052

[8] A. B. Lange, U. P. Schultz, and A. S. Sorensen, "Unity: A unified software/hardware framework for rapid prototyping of experimental robot controllers using FPGAs," in *Proceedings of the Eigth full-day Workshop on Software Development and Integration in Robotics (SDIR VIII) at ICRA 2013*, 2013.

[9] ——, "Unity-Link: A software-gateware interface for rapid prototyping of experimental robot controllers on FPGAs," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, accepted for publication.

[10] A. B. Lange, K. H. Andersen, U. P. Schultz, and A. S. Sorensen, "HartOS - a Hardware Implemented RTOS for Hard Real-time Applications," *Proceedings., Eleventh IFAC/IEEE Conference on Embedded Systems and Programmable Devices, 2012.*, pp. 207–213, May 2012, iSSN: 1474-6670.

A Top-Down Approach to Managing Variability in Robotics Algorithms

Selma Kchir, Tewfik Ziadi, Mikal Ziane

UMR CNRS 7606 LIP6-MoVe
Université Pierre et Marie Curie, France
Email: firstname.lastname@lip6.fr

Serge Stinckwich

UMI UMMISCO 209 IRD/UPMC
Université de Caen-Basse Normandie, France
Email: serge.stinckwich@ird.fr

***Abstract*— One of the defining features of the field of robotics is its breadth and heterogeneity. Unfortunately, despite the availability of several robotics middleware services, robotics software still fails to smoothly handle at least two kinds of variability: algorithmic variability and lower-level variability. The consequence is that implementations of algorithms are hard to understand and impacted by changes to lower-level details such as the choice or configuration of sensors or actuators. Moreover, when several algorithms or algorithmic variants are available it is difficult to compare and combine them.**

In order to alleviate these problems we propose a top-down approach to express and implement robotics algorithms and families of algorithms so that they are both less dependent on lower-level details and easier to understand and combine. This approach goes top-down from the algorithms and shields them from lower-level details by introducing very high level abstractions atop the intermediate abstractions of robotics middleware. This approach is illustrated on 7 variants of the Bug family which were implemented using both laser and infrared sensors.

I. Introduction

The development of robotics software must deal with a large amount of variability from at least two sources. First, robots are very different from each other: "They have different locomotion mechanisms, different onboard computational hardware, different sensor systems, and different sizes and shapes." [1]. Second, robots are used for very different tasks with varying constraints which leads to a large variety of algorithms.

Robotics middleware, among other improvements, has been a significant attempt at addressing the first kind of variability by decoupling robotics application from lower levels details. "It is designed to manage the heterogeneity of the hardware, [...] simplify software design [...]. A developer needs only to build the logic or algorithm as a component" [2].

The task is huge however, as well explained by W.D. Smart who considers the hypothetical case of middleware providing a obstacle-avoidance routine for a mobile robot [1]. According to him "writing a generic obstacle avoider that will work for all locomotion mechanisms, using input from all possible sensors is a daunting task". It is not surprising then that a few years later middleware services are still far from solving this decoupling problem.

Consequently, robotics algorithms are still

- difficult to understand,
- difficult to adapt or combine,
- impacted by changes in lower-level details.

The second kind of variability multiplies the already large number of algorithmic variants: a general obstacle avoidance routine should take into account very different kinds of terrain with fixed or mobile obstacles of different shapes and sizes etc. Thus, even with the very simplified assumptions of a family of navigation algorithms like Bug [3] it is then far from obvious in which case such or such variant is best suited to go from one point to another while avoiding obstacles [4]. If we could use the same implementation of an algorithm, comparison between variants of this algorithm would become meaningful and easy to perform. In this paper we propose a top-down approach to complement the bottom-up middleware approach. The input of this approach is a robotic task and either a family of algorithms or at least enough knowledge to produce algorithms to solve it. The approach is similar to the partial evaluation of a planner except that it is performed by a human expert. Its output is twofold:

- a set of algorithmic, sensory or action abstractions,
- a configurable generic algorithm.

The produced algorithm can be configured by providing the values of a series of parameters to be adapted to different hypotheses, say on the environment. Furthermore it is generic in two other ways. First, it is decoupled from low-level details on sensors and actuators and second, the algorithmic variability which cannot be resolved statically by specifying configuration parameters is managed by dynamically linking the actions abstractions to executable routines.

This rest of this paper is organized as follows. Section II describes our approach and illustrates it on Bug algorithms. Section III deals with the problem of organizing reusable implementations of the abstractions. Section IV discusses the approach. Section V presents preliminary validation results while section VI compares our approach to related work.

II. Rational

In this section, we propose our approach to provide a generic algorithm for a family of robotics algorithms or a robotic task. Our approach is similar to the partial evaluation of a planner except that it is performed by a human expert. Starting from existing algorithms descriptions (textual

descriptions, pseudo-code) and from the requisite sensors to define them, we can identify appropriate abstractions and the invariant parts among all algorithms which are defined in terms of these abstractions (behaviors, data types). The usefulness of this approach is to extract a description of a generic hardware-independent algorithm. Figure 1 depicts the three main steps of our approach:

1) **Abstractions Identification**
   - Definition: Abstractions are a set of methods which allows to encapsulate specific data or to abstract specific methods by extracting a general signature from specific ones.
   - Description: If we consider a family of algorithms where variants propose different strategies to perform the same task, we should handle this variability. Therefore, abstractions must gather hardware and what we call algorithmic variability. One of the possible approaches to define abstractions is a goal directed approach. In our point of view, defining hardware abstractions does not require specific knowledge of specific robots sensors or actuators but a global description of the needed data types of the handled task. Regarding algorithmic abstractions they relie on a comprehensive study of algorithms variants whence the needed intervention of a robotic expert. Invariant parts among algorithms of the same family can then be written in terms of these abstractions.
2) **Generic Algorithm Definition**
   - Definition: A generic algorithm is a sequence of instructions written in terms of hardware and algorithmic abstractions.
   - Decription: At this level, the robotic expert combine the abstractions identified previsouly in order to write the algorithm or to design a state machine corresponding to a specific robotic task. In this way, there is no specific detail related to a particular algorithms variant and the defined algorithm is completely independent from low level details and from specific algorithms variants.
3) **Organising Implementation**
   - Definition: Organising implementation means implementing variants of a family of algorithms starting from the generic algorithm.
   - Description: In this step, we implement the invariant parts of the algorithm in terms of abstractions. To do so, the Template Method (TM) design pattern [5] allows the definition of an algorithm skeleton in an abstract class as a template method. This method is the generic algorithm identified previsouly. It defines the basic step of the algorithm as "placeholders" methods (or hook methods), that are different in each subclass. Invariants parts of the algorithm are represented once in the abstract class as concrete methods. Abstract methods will be used to represent needed operations that are different in each subclass. The main idea is to represent the variants of the algorithm as subclasses that implement these abstract methods. Thus, algorithmic abstractions will be implemented in these subclasses. Concerning hardware abstractions, they are not implemented in subclasses like algorithmic abstractions but delegated to adaptors which implement them. We define one adapter per physical sensor. Adaptors implement these operations to extract physical data and convert them to the needed abstractions. Then, the specification of the physical sensor used in the algorithm is done at the deployment level.

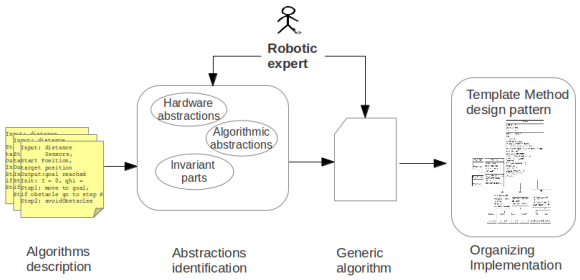


Fig. 1. Generic algorithm: from identification to implementation

In the next section, we illustrate our approach on the Bug family of navigation algorithms.

III. Case study: Bug algorithms

It is symptomatic that even with the very simplified assumptions underlying these algorithms it is still difficult:

- to understand the differences among the algorithmic variants,
- to choose one variant over another,
- to share development efforts among variants.

A. Bug algorithms overview

The Bug algorithms attempt to solve the navigation problem in an unknown two-dimensional environment with fixed obstacles and a known goal position. Over 19 versions of Bug algorithms have been defined in the literature. Among them, 7 variants are considered in this paper.

Bug algorithms share the following assumptions [3] : 1) The environment is unknown and a finite number of fixed obstacles are placed arbitrarily. 2) The robot is considered as a point (i.e. without body). It has perfect sensors (for obstacle detection) and a perfect localization ability (e.g. to compute its distance to its goal).

In Bug family, the robot has a local knowledge and a global goal. In other words, the inputs of Bug algorithms are the robot's start position and the target position. At the end of the algorithm's execution, the robot must indicate if its goal is reached or if the goal is unreachable. Most of the Bug algorithms can be programmed on any mobile robot using tactile or distance sensing and a localization method while some require distance sensing.

B. Abstractions identification

In the litterature, as far as we know, Bug algorithms are only published with an informal description. Bug algorithms are very similar fundamentally but differ in some points of interest. Their principle is the following: (1) The robot **motion to its goal** until an **obstacle is detected** on its way. (2) From the point where the obstacle were encountered (called hit point), the robot **looks for a point** (called leave point) **around the encountered obstacle** to be able to move to its goal again. (3) When a **leave point is identified**, the robot **moves to it** and leaves the obstacle. Steps (1), (2) and (3) are repeated until **the goal is reached** or until the robot indicates that the **goal is unreachable**. This description of Bug hides hardware and algorithmic variability. Hardware variability impacts on obstacle detection and localisation tasks because of their dependency of the robot actual sensors. Algorithmic variability is related to the leave point identification task. Following a goal directed approach and based on a comprehensive study of Bug algorithms, we have identified the following abstractions (described here as methods):

- Motion to goal: the robot needs to face its goal and to be able to go ahead it: FACEGOAL(POINT GOALPOSITION), GOAHEAD().
- Obstacle on the robot's way: BOOL OBSTACLEINFRONTOFTHEROBOT()
- Current position: POINT GETPOSITION()
- Get around the encountered obstacle consists in performing clockwise or anticlockwise circumnavigation while maintaining a safety distance from the obstacle: WALLFOLLOWING(DIRECTION)
  - DOUBLE GETSAFEDISTANCE()
  - BOOL OBSTACLEONTHERIGHT(), BOOL OBSTACLEONTHELEFT(): Depending if we follow the wall on the right or the left.
  - DOUBLE GETRIGHTDISTANCE(), DOUBLE GETLEFTDISTANCE(): To keep a safety distance from a wall, we need to know the distance to the wall from the right and the distance from the left.
- Look for a leave point: IDENTIFYLEAVEPOINT(BOOL DIRECTION, POINT ROBOTPOSITION, POINT GOALPOSITION) which consists in wall following while looking for a leave point. This could be a local decision (choose the first leave point which satisfies the algorithm condition) or a global decision (choose the best leave point after visiting all the points around the obstacle). The following conditions are examples of decisions that the algorithm defines to find a leave point:
  - **The closest point around obstacle boundary condition.** It consists on recording the closest point to the goal among all the points ever visited by the robot while performing boundary following[3][6].
  - **The m-line detection condition.** It is a straight line between the starting point and the target which aims at providing a set of prededefined leave points. The robot must leave only on these points[6][7].
  - **The local minimum condition.** Using its distance sensors, the robot can detect discontinuity points[8] on obstacles on its way with respect to the target.
  - **The disabling segment condition.** A disabling segment occurs when the robot cannot move to its goal from all points in a segment while perfoming boundary following.
  - **The step method.** The robot uses its distance sensors to detect a point which is *STEP*[9] closer to the target than any point already visited.

  Thus, we have identified the method FINDLEAVEPOINT(POINT ROBOTPOSITION, POINT HITPOINT, POINT GOALPOSITION) which will be applied to each point around the obstacle and which is specific to each algorithm variant to hide all low level details. The code of IDENTIFYLEAVEPOINT is given by the algorithm 1.
- Leave point identified: BOOL ISLEAVEPOINTFOUND()
- RESEARCHCOMPLETE: In case of a local decision of leave point identification, the research complete is equivalent to the condition leave point identified. In case of a global decision, the research complete decision is equivalent to a complete cycle around the obstacle: RESEARCHCOMPLETE(POINT ROBOTPOSITION, POINT HITPOINT, POINT GOALPOSITION).
- Move to the leave point: Once the leave point identified, the robot goes to it. This is a variability point because the robot can go to the leave point following the shorter distance or other strategies: GOTOLEAVEPOINT(POINT LEAVEPOINT).
- GOAL UNREACHABLE: This condition is checked if there is no leave point identified after performing an entire cycle around the obstacle: BOOL COMPLETECYCLEAROUNDOBSTACLE(POINT ROBOTPOSITION,POINT HITPOINT) AND NOT ISLEAVEPOINTFOUND()
- goal reached: Depending on the algorithm objective, we define an error margin which indicates if the robot must reach its goal or stops before arriving to it: BOOL GOALREACHED(POINT ROBOTPOSITION, POINT GOALPOSITION, DOUBLE ERR)

Algorithm 1: Identify leave method algorithm

function IDENTIFYLEAVEPOINT($Bool\ direction$, $Point\ robotPos$, $Point\ goalPos$)

computeData(robotPos);
wallFollowing(direction);
findLeavePoint(robotPos, hitPoint);

After identifying these abstractions, we classify them into hardware abstractions and algorithmic abstractions.

1) Hardware abstractions:
   GETPOSITION(), GETSAFEDISTANCE(), OBSTACLEONTHELEFT(), OBSTACLEONTHERIGHT(), GETRIGHTDISTANCE(), GETLEFTDISTANCE(), OBSTACLEINFRONTOFTHEROBOT(),

2) Algorithmic abstractions: FINDLEAVEPOINT(POINT ROBOTPOSITION, POINT HITPOINT, POINT GOALPOSITION), IDENTIFYLEAVEPOINT(BOOL DIRECTION, POINT ROBOTPOSITION, POINT GOALPOSITION), BOOL GOALREACHED(POINT ROBOTPOSITION, POINT GOALPOSITION, DOUBLE ERR), COMPLETECYCLEAROUNDOBSTACLE(POINT ROBOTPOSITION,POINT HITPOINT), BOOL ISLEAVEPOINTFOUND(), GOTOLEAVEPOINT(POINT LEAVEPOINT).

C. Generic Bug algorithm definition

The generic algorithm is a combination of the previously defined abstractions as a sequence of instructions. Our generic algorithm is given in 2.

Algorithm 2: Bug generic algorithm

Sensors : A perfect localization method.
An obstacle detection sensor
input : Position of Start (q_{start}), Position of Target (q_{goal})
Initialisation: robotPos ← `getPosition();`
direction ← `getDirection();`

```
if goalReached(robotPos) then
 | EXIT_SUCCESS;
end
else if obstacleInFrontOfTheRobot() == true
then
   identifyLeavePoint (direction, robotPos,
   goalPos);
   if leavePointFound() &&
   researchComplete(robotPos, getHitPoint(),
   goalPosition) then
      goToLeavePoint(getLeavePoint());
      faceGoal()();
   end
   else if
   completeCycleAroundObstacle(robotPos,
   getHitPoint()) && !leavePointFound() then
    | EXIT_FAILURE;
   end
end
else
 | motionToGoal();
end
```

D. Implementation: Template Method design pattern

As we said previously, the Template Method deals with variability problem by proposing to define the basic step of the algorithm as a template method written in terms of abstractions which will be implemented in subclasses. Define a subclass for each algorithm without handling sensors and actuators variability cause a combinatorial explosion. Consequently, we dealed with this problem by delegating hardware variability to what we call virtual sensors or adaptors. Virtual sensors convert specific data from the physical sensors to the needed data in the algorithm. They implement a set of interface abstract operations. For instance, the method OBSTACLEINFRONTOFTHEROBOT() is different in each sensor adaptor. We have defined an adaptor per sensor. In case of multiple physical sensors use, their adaptors are combined together to provide the needed data for the algorithm. The architecture of our implementation is presented in Figure 2.

To perform wall following, right hand and left hand algorithms are written in terms of sensors abstractions (i.e. OBSTACLEONTHELEFT() and OBSTACLEONTHERIGHT())

E. Discussion

The hierarchy of classes defined by our approach can lead to a combinatorial explosion if we add additional variants of Bug family. It is then much cheaper to build operations of the algorithm as components and to assemble desired family members from them. This is our main motivation for using an alternate solution based on Software Product Lines (SPL). SPL are easy to use, compact and generate only a configuration which interests the user. This constitutes an immediate topic for a future work.

IV. RESULTS AND VALIDATION

Several performance comparison studies[4] were realized on the Bug family. In this section, we do not intend to perform a comparison between Bug family but to prove that each algorithm of the Bug family fits with our generic algorithm.

We demonstrate the capabilities of our generic algorithm in the OROCOS-RTT framework through 7 variants of Bug: Bug1 [3], Bug2 [6], Alg1 [7], Alg2 [10], DistBug [9],TangentBug [8] and Rev1 [11].

Simulation was performed using Stage-ROS with 2 configurations.

The first tested configuration was done with a laser scanner with 180 degrees scanning angle and a detection range which varies from 0.02 meters to approximately 4 meters and a GPS for localization. The robot does not have any knowledge about its environment except its start position and the goal position.

The second configuration was done using 3 infrared range sensors placed on the front, on the left and on the right compared to the central axis of the robot with a field of view equals to 26 degrees and a range which detects until 2 meters.

To demonstrate that any algorithm of the ones studied here fits with our generic algorithm, we have tested both configurations in 3 different environments for all algorithms. We defined the first environment (see figure 3) to validate the target reachability condition. In all implemented algorithms, the robot returns failure because it can not achieve its goal. The second environment shown in figure 4 is a simple simulation environment with one obstacle. Some algorithms behave similarly in this environment despite their different obstacle avoidance strategies. For instance,

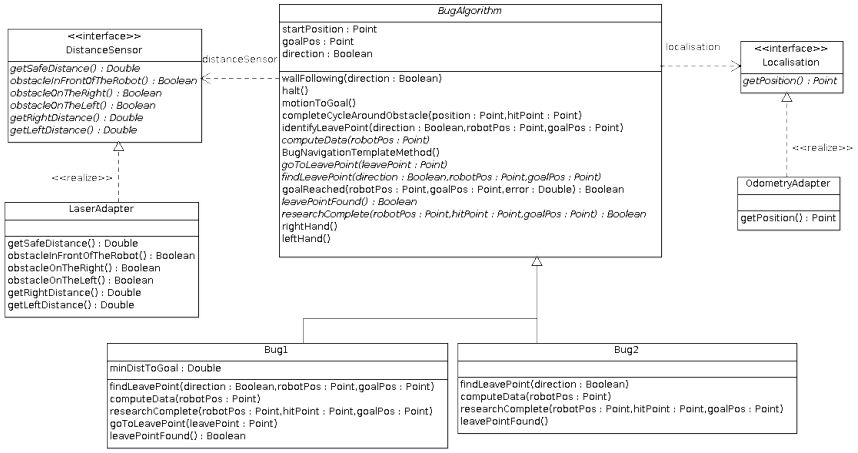


Fig. 2. Bug algorithms class diagram (extract)

Alg1 and Bug2 rely on the M-line detection condition but behave differently when they encounter an already traversed point. In other words, Alg1 was defined above Bug2 to overcome situations where the robot can find itself in an overlasting loop around obstacle. For this reason, we defined the third environment, presented in figure 5, to validate the encountered points condition (i.e. when the robot encounters an already traversed point), particularly used in the algorithms Alg1 and Alg2.

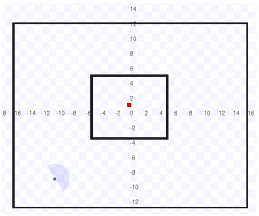


Fig. 3. Environment with unreachable target

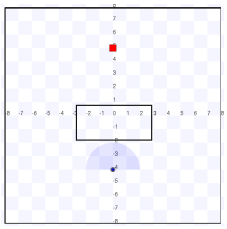


Fig. 4. Basic Simulation Environment

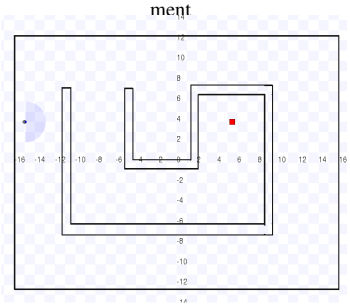


Fig. 5. Environment to validate the encountered points condition

The trajectory of the robot could depend on the computation time of sensors information. Since most of the algorithms treatments are reactive, we had to define the period of execution time which allow us to get as much real-time information as possible and to optimize the execution time and the robot's path. Consequently, we set the execution period of the algorithm to 0.5 seconds.

Alll the code of Bugs algorithm variants are available on github: `https://github.com/SelmaKchir/BugAlgorithms`.

V. Related work

Several software engineering technologies and methods aim at improving software design and reusability. In robotics, reusability is obtained after handling variability which could be related to the robot hardware or to the robot's capabilities (e.g. different strategies for obstacle avoidance, etc). Robotics software like Robot Operating System (ROS)[12], RISCWare framework [13], Player [14] propose a hardware abstraction layer to encapsulate the sensors that gather information about the environment and provide a set of predefined intefaces. These middlewares rely on a bottom up approach which consists on classifying the most used physical sensors (or actuators), analysing the potential data they are able to provide and then define interfaces.
Unlike these middlewares, we use a top-down approach which consists in analysing the needed data in a particular application, defining abstractions and then writing the application in terms of these abstractions (possibly provided by middlewares).

In our approach, we have applied the Template Method design pattern to implement variants of a generic algorithm. They are several proposals that try to apply design patterns in robotics to handle variabilities. CLARATy authors [15] say they leverage many well-know techniques developed by the software community including design patterns but without being more explicit. MARIE [16], a middleware framework for robotics, applied the Mediator Design Pattern [5] to create a mediator interoperability layer for distributed robotics applications.

VI. Conclusion and Perspectives

In this paper we have proposed an approach to organize families of algorithms so that the algorithmic decisions to make are clearly expressed and decoupled from implementation details. Our approach consists in relying on the Template Method design pattern to define a generic algorithm for Bug algorithms family.

Our approach consists of three steps. The first step takes as input a set of algorithmic variants; as described in the litterature, and manually extract hardware abstractions and what we have called algorithmic abstractions. The generic algorithm is then defined as a sequence of instructions in terms of these abstractions. The implementation of these variants relie then on the Template Method design pattern.

This approach has been illustrated on the Bug family of robot navigation algorithms. Seven implemented variants of Bug algorithms have been implemented using the OROCOS-RTT robotic framework. Simulation was performed in different unknown environments with a random positioning of obstacles.

Bug variants are about 20 versions of algorithms. Implementing all `BugAlgorithm` subclasses is very complicated to understand and too expensive to build all Bug family members. In addition, it is error prone to not define constraints on subclasses and the data they must specify and use.
It is then much cheaper to build operations of the algorithm as components and to assemble desired family members from them. This is our main motivation for using Software Product Lines (SPL) [17] as an immediate topic of future work.

References

[1] W. D. Smart, "Is a Common Middleware for Robotics Possible?" in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07) Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*, Nov. 2007. [Online]. Available: http://www.cse.wustl.edu/~wds/library/papers/2007/iros-ws2007.pdf

[2] A. Elkady and T. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, vol. 2012, 2012.

[3] V. J. Lumelsky and A. A. Stepanov, "Effect of Uncertainty on Continuous Path Planning for an Autonomous Vehicle," in *Decision and Control, 1984. The 23rd IEEE Conference on*, vol. 23, dec. 1984, pp. 1616–1621.

[4] J. Ng and T. Bräunl, "Performance Comparison of Bug Navigation Algorithms," *J. Intell. Robotics Syst.*, vol. 50, no. 1, pp. 73–84, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10846-007-9157-6

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Publishing, 1995.

[6] V. Lumelsky and A. Stepanov, "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment," *Automatic Control, IEEE Transactions on*, vol. 31, no. 11, pp. 1058 – 1063, nov 1986.

[7] H. Noborio, K. Fujimura, and Y. Horiuchi, "A Comparative Study of Sensor-based Path-planning Algorithms in an Unknown Maze," in *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, vol. 2, 2000, pp. 909 –916 vol.2.

[8] I. Kamon, E. Rimon, and E. Rivlin, "TangentBug: A Range-Sensor-Based Navigation Algorithm," *The International Journal of Robotics Research*, vol. 17, no. 9, pp. 934–953, September 1998.

[9] I. Kamon, "Sensory Based Motion Planning with Global Proof," in *In Proceedings of the IROS95*, 1995, pp. 435–440.

[10] A. Sankaranarayanar and M. Vidyasagar, "Path Planning for Moving a Point Object amidst Unknown Obstacles in a Plane: a New Algorithm and a General Theory for Algorithm Development," in *Decision and Control, 1990., Proceedings of the 29th IEEE Conference on*, dec 1990, pp. 1111 –1119 vol.2.

[11] Y. Horiuchi and H. Noborio, "Evaluation of Path Length Made in Sensor-Based Path-Planning with the Alternative Following," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2001)*, vol. 2. IEEE Xplore, 2001, pp. 1728–1735.

[12] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an Open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[13] A. Elkady, J. Joy, T. Sobh, and K. Valavanis, "A Structured Approach for Modular Design in Robotics and Automation Environments," *Journal of Intelligent & Robotic Systems*, pp. 1–15, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10846-012-9798-y

[14] T. H. Collett, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a Practical Robot Programming Framework," in *Proc. of the Australasian Conf. on Robotics and Automation (ACRA)*, Sydney, Australia, 2005.

[15] I. A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, and I.-H. Shu, "CLARAty: Challenges and Steps Toward Reusable Robotic Software," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 23–30, 2003.

[16] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud, "Robotic Software Integration Using MARIE," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 055–060, Mar. 2006. [Online]. Available: http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-3/055-060.pdf

[17] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

Engineering the Hardware/Software Interface for Robotic Platforms – A Comparison of Applied Model Checking with Prolog and Alloy

Md Abdullah Al Mamun[1], Christian Berger[1] and Jörgen Hansson[1]

Abstract— **Robotic platforms serve different use cases ranging from experiments for prototyping assistive applications up to embedded systems for realizing cyber-physical systems in various domains. We are using 1:10 scale miniature vehicles as a robotic platform to conduct research in the domain of self-driving cars and collaborative vehicle fleets. Thus, experiments with different sensors like e.g. ultra-sonic, infrared, and rotary encoders need to be prepared and realized using our vehicle platform. For each setup, we need to configure the hardware/software interface board to handle all sensors and actors. Therefore, we need to find a specific configuration setting for each pin of the interface board that can handle our current hardware setup but which is also flexible enough to support further sensors or actors for future use cases. In this paper, we show how to model the domain of the configuration space for a hardware/software interface board to enable model checking for solving the tasks of finding any, all, and the best possible pin configuration. We present results from a formal experiment applying the declarative languages Alloy and Prolog to guide the process of engineering the hardware/software interface for robotic platforms on the example of a configuration complexity up to ten pins resulting in a configuration space greater than 14.5 million possibilities. Our results show that our domain model in Alloy performs better compared to Prolog to find feasible solutions for larger configurations with an average time of 0.58s. To find the best solution, our model for Prolog performs better taking only 1.38s for the largest desired configuration; however, this important use case is currently not covered by the existing tools for the hardware used as an example in this article.**

I. Introduction and Motivation

Self-driving vehicles [1], as one popular example for intelligent robotics, highly depend on the usage of sensors of different kinds to automatically detect road and lane-markings, detect stationary and moving vehicles, and obstacles on the road to realize automated functionalities such as automatic driving and parking or to realize collision prevention functions. New functionalities, based on market demands for example, require the integration of new sensors and actors to the increasingly intelligent vehicle. These sensors and actors are interfaced by a hardware/software board, whose number of available physical connection pins is however limited.

When selecting such an interface board for a robotic platform, we do not necessarily limit our focus on a possible pin assignment for a set of sensors/actors which is fulfilling our current needs. Additionally, we consider also the possibility of extending the current hardware architecture with additional sensors and actors using the same interface board for future use cases. Furthermore, exchanging such an interface board might require the modification of existing low-level code or requires the development of new code for the embedded real-time OS to realize the data interchange with the given set of sensors/actors.

As a running example in this paper, we are using the STM32F4 Discovery Board [15] as shown in Fig. 1. This figure depicts our complete hardware/software interface setup for our self-driving miniature vehicle consisting of different distance sensors, actors for steering and accelerating the vehicle, an emergency stop over an RC-handset, as well as a connection to our inertial measurement unit (IMU) to measure accelerations and angular velocities for computing the vehicle's heading. The configuration space for that interface board from which an optimal solution shall be chosen is shown in Fig. 2.

The selection of an interface board of a certain type depends on different factors like computation power and energy consumption. Furthermore, it must support enough connection possibilities for the required sensors and actors. However, matching a given set of sensors and actors to the available pins of a considered hardware/software interface board is a non-trivial task because some pins might have a multiple usage; thus, using one pin for one connection use case would exclude the support of another connection use case. To derive the best decision how to connect the set of sensors and actors, we need to have a clear idea about all possible pin assignments up to a certain length l, where l describes the number of considered pins for one configuration (e.g., a configuration length using ten pins could describe the usage of 4 digital, 4 analog, and 2 serial pins).

From our experience, manually defining a feasible pin assignment for a desired configuration requires roughly an hour, which includes checking the manual and to evaluate, if future use cases for the HW/SW interface board can still be realized. This process needs to repeated, whenever the sensor layout is modified, e.g. by adding further sensors or replacing sensors with different types. Thus, this manual work is time-consuming and error-prone.

In this paper, we address this configuration problem common for robotic platforms by applying model checking to find a) at least one possible pin assignment, b) all possible pin assignments, and consequently c) the best possible pin assignment in terms of costs. In our case, costs are defined as the number of multiple configurations per pin; e.g., let us assume one pin from the hardware/software interface board can be used for analog input, I^2C bus, and serial communication; its price would be 3. Reducing the overall

[1]Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg {abdullah.mamun,christian.berger, jorgen.hansson}@chalmers.se

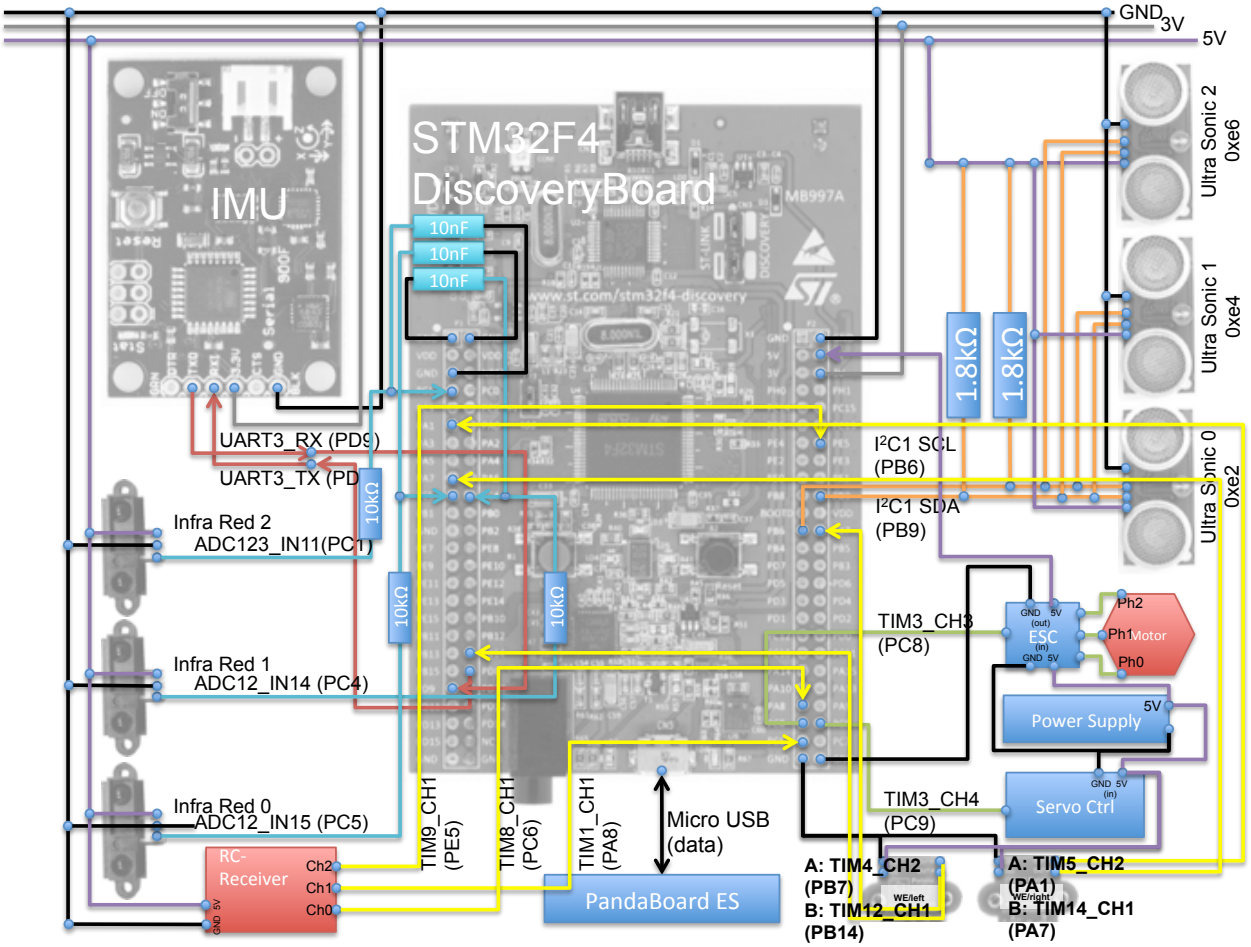


Fig. 1. Full setup of an STM32F4 Discovery Board with sensors and actors to realize the hardware/software interface for our self-driving miniature vehicle.

costs would result in a final pin assignment where pins with a low multiple usage are preferred to allow for further use cases of the board in the future.

We model the configuration problem as an instance of a domain-specific language (DSL) for the configuration space of a hardware/software interface board to serve the declarative languages Alloy [9] and Prolog [5]. Based on this model, we show how to realize the aforementioned three use cases in these languages while measuring the computation time to compare the model checking realized by these tools.

The rest of the paper is structured as follows: In Sec. II, the combinatorial optimization problem for engineering the hardware/software interface board is introduced, formalized, and constraints thereof are derived. Furthermore, the complexity of the configuration space is analyzed before the formal experiment of applying model checking with Alloy and Prolog and its results are described, analyzed, and discussed. The article closes with a discussion of related work and a conclusion.

II. Engineering The Robotic Hardware/Software Interface–A Combinatorial Optimization Problem

Fig. 1 shows the connection setup of the hardware/software interface board that we are using on our 1:10 scale self-driving miniature vehicle platform. In the given configuration, the board handles 14 different input sources and two output sinks:

- three Sharp GP2D120 infrared sensors which are generating a distance-dependent voltage level,
- an IMU Razor 9DoF board connected via a serial connection that provides acceleration and angular velocity data in all three dimensions as well as housing a magnetometer to provide information about the vehicle's heading,
- a three-channel receiver for the remote controller handset to stop and control the miniature vehicle in emergency cases connected as analog source to the input capturing unit (ICU),
- three (and up to 16) ultra-sonic devices attached via the I^2C digital bus,
- and a steering and acceleration motor connected via pulse-width-modulation (PWM) pins to access the actors

Pin	ADC1	I2C1	I2C2	I2C3	UART1	UART2	UART3	UART4	UART6	CAN	ICU1	ICU2	PWM2	PWM3	ICU4	PWM4	ICU5	PWM5	PWM6	ICU10
PA1	ADC1-IN1											TIM2_CH2					TIM5_CH2			
PA2	ADC1-IN2					UART2-TX							TIM2_CH3					TIM5_CH3		
PA3	ADC1-IN3					UART2-RX							TIM2_CH4					TIM5_CH4		
PA8				I2C3-SCL							TIM1_CH1									
PB0	ADC1-IN8													TIM3_CH3						
PB1	ADC1-IN9													TIM3_CH4						
PB6		I2C1-SCL			UART1-TX															
PB7					UART1-RX										TIM4_CH2					
PB8										CAN1-RX						TIM4_CH3				TIM10_CH1
PB9		I2C1-SDA								CAN1-TX										
PB10			I2C2-SCL				UART3-TX													
PB11			I2C2-SDA				UART3-RX						TIM2_CH4							
PC6									UART6-TX					TIM3_CH3					TIM8_CH3	
PC9				I2C3-SDA										TIM3_CH4					TIM8_CH4	
PC10							UART3-TX	UART4-TX												
PC11							UART3-RX	UART4-RX												

Fig. 2. Domain of possible pin assignment configurations for the STM32F4 Discovery Board: Analog input is marked with light blue, green highlights I^2C-bus usage, purple describes serial input/output, gray describes CAN bus connection, and light yellow ICU and PWM-timer-based input/output usage.

of the robotic platform.

To handle all aforementioned sensors and actors using ChibiOS [14] as our hardware abstraction layer (HAL) and real-time operating system, we need to engineer both the hardware connection mapping as well as the software configuration setup fulfilling the following constraints:

- Attaching the hardware data sources to those pins that are able to handle the required input source at hardware-level (e.g. the STM32F4 chip in our case),
- connecting the hardware data sinks to those pins that are able to handle the required output sources at hardware-level,
- configuring the CPU to handle the hardware data sources and sinks in the case of multiple usage per pin,
- and considering the appropriate software support in the low-level layer of the hardware abstraction layer (e.g. in our case considering that ICUs can only be handled if attached to a pin supporting timers on channel 1 or 2).

The aforementioned constraints need to be considered during the engineering process. In this section, we describe the general idea behind our modeling approach for these domain-specific constraints, considerations about the complexity in the model processing stage, as well as how instances of the DSL are transformed to enable model checking serving the following use cases during the engineering process for the hardware/software interface board:

1) Find a feasible and valid pin configuration fulfilling a requested configuration,
2) enumerate all possible pin configurations for a given configuration,
3) and in combination with the former use case, find the best possible pin configuration in terms of costs for pin usage.

A. The Domain of Pin Assignment Configurations

In Fig. 3, a visualization for the domain of possible pin assignment configurations is depicted. The basic model can be represented by a graph G consisting of nodes N representing all pins of a hardware/software interface board, a set E describing directed edges connecting the nodes, and a set A of edge annotations representing concrete pin configurations. One concrete pin assignment configuration is then represented by a path P from n_B to n_E.

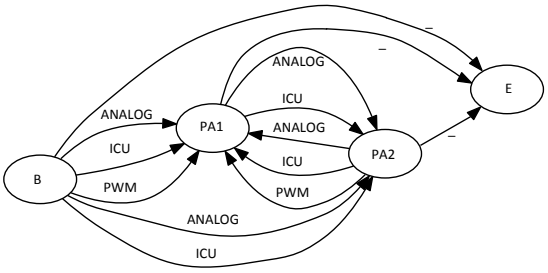


Fig. 3. Visualization of the graph $G = \{N, E, A\}$ for the domain of possible pin assignment configurations of a fictitious hardware/software interface board with two pins having multiple usage respectively. A concrete configuration is represented by a path P from n_B to n_E with $|P| < |N|$.

Furthermore, the following constraints must hold to restrict the set of possible paths through G to consider only those representing valid configurations:

- The graph must not contain self-reflexive edges at the nodes because one pin can only be used once for a pin configuration usage.
- The path P of a concrete pin assignment configuration must begin at n_B and must end in n_E.
- The length of P must be less than the size of set N.

This domain-specific model can also be represented as a table as shown in Fig. 2, which can be maintained with any spreadsheet tool for example. Thus, only all possible configuration settings need to be defined per pin because all aforementioned constraints must be considered only during the concrete assignment process, which in turn can be fully automated with model checking. An overview of the model checking workflow is shown in Fig. 4.

The concrete realizations for both paths in the workflow are described in Sec. III-B for Prolog and in Sec. III-C for Alloy.

B. Complexity Considerations

The combinatorial complexity of finding a solution for the pin assignment problem for a given configuration with a length l is determined by the following three dimensions: Set N of available pins, set M of different configurations per pin, and the maximum length L up to which the assignments shall be solved.

For $|M| = 1$, the combinatorial problem is reduced to determining how many possibilities $C_{|M|=1}$ are available

to pick k objects from N as calculated by the binomial coefficient shown in Eq. 1. Hereby, k describes the length of a considered configuration.

$$C^{|N|}_{|M|=1} = \sum_{k=1}^{L} \binom{|N|}{k} = \sum_{k=1}^{L} \frac{|N|!}{(|N|-k)!k!}. \quad (1)$$

However, the configuration space grows once the limitation for set M is relaxed as outlined in the following example:

$$\begin{aligned} C^{|N|=4}_{|M|=1} &= 4+6+4+1=15. \\ C^{|N|=4}_{|M|=2} &= 4*2+6*3+4*5+1*6=47. \\ C^{|N|=4}_{|M|=3} &= 4*3+6*6+4*10+1*15=103. \\ &\ldots \end{aligned}$$

Analyzing the factors, which are multiplied with the binomial coefficient summands, it can be seen that they are constructed by the rule depicted by Eq. 2.

$$K(n,m) = \begin{cases} 1+\sum_{p=1}^{n} *K(p,m-1) & \text{if } m>1, \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

Using Eq. 2, Eq. 1 can be adapted for the generic case as shown in Eq. 3.

$$C^{|N|}_{|M|} = \sum_{k=1}^{L} \binom{|N|}{k} * K(k,|M|). \quad (3)$$

With Eq. 3, a hardware/software interface board consisting of 6 pins each providing 4 different configuration possibilities would result in 1,519 different assignment options.

III. Evaluating Applied Model Checking for Pin Assignment Configurations

In the previous section, we have outlined the domain of possible pin assignment configurations alongside with complexity considerations. Now, we investigate the following research questions related to the challenges during the engineering process of the hardware/software interface for robotic platforms:

RQ-1: How can Prolog be used to apply model checking on instances of the domain of possible pin assignment configurations to determine a feasible, all possible, and the best configuration assignment?

RQ-2: How can Alloy be used to apply model checking on instances of the domain of possible pin assignment configurations to determine a feasible, all possible, and the best configuration assignment?

RQ-3: Which approach performs better compared to the other for the particular use cases?

Since we have full control over the involved parameters for the model checkers, we carried out a formal experiment according to [11] to answer the research questions.

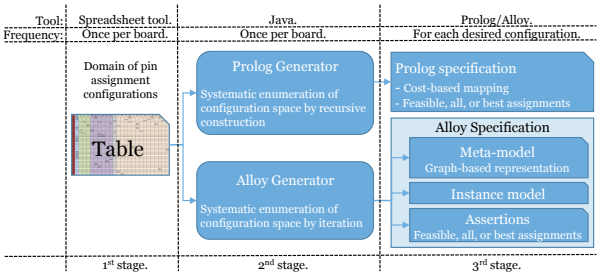


Fig. 4. Overview of the workflow with involved tools for transforming an instance of the domain model for possible pin assignment configurations into representations, which can be used for model checking and to compute a feasible, all possible, or the best pin assignment for a hardware/software interface board. The first two stages need to be maintained once per hardware/software interface board, while the last stage needs to be carried out for each desired configuration.

A. Designing the Formal Experiment

To compare the possibilities and performance of Prolog and Alloy, both tools were used to solve the following problems:

1) Basis for the formal experiment was the concrete instance of possible pin assignment configurations for the 46 pins of our hardware/software interface board STM32F4 Discovery board.
2) From this instance, 30 trivial assignments with costs of 1 were removed because any identified assignment for the pins with multiple usage can be simply extended by pins with costs 1 without modifying the assignment for the other pins.
3) For the remaining 16 pins as shown in Fig. 2 which can be used with multiple configurations up to costs of 4, a pin assignment for a given configuration of varying lengths ranging from 0 up to 10 is needed to be solved for the use cases *one feasible*, *all possible*, and *the best* pin assignment.
4) The given configuration, for which pin assignments are needed to be determined, consisted of {`analog`, `analog`, `analog`, `icu`, `analog`, `analog`, `serial-tx`, `serial-rx`, `can-tx`, `i2c-sda`}. This list was shortened from the end to provide shorter configurations as input.
5) To verify that both model checking approaches identify also impossible configurations, a given configuration containing too many elements from a given type of set A was constructed.
6) For every use case and for every configuration length, the required computation time was determined.

To answer RQ-1 and RQ-2, respectively, we decided to use action design research [13] as the method to identify and analyze a domain problem for designing and realizing an IT artifact to address the problem.

To answer RQ-3, we decided to measure the *required computation time* for each approach because in our opinion, it is the apparent influencing factor for the last stage in our workflow, where researchers and developers have to cope with during the development and usage of a robotic platform.

According to Eq. 3, the total configuration space for the running example with $|N| = 16$ pins and $|M| = 20$ configuration possibilities would contain 1,099,126,862,792 elements. However, due to the reduced number of multiple usages per pin in our concrete example of the STM32F4 Discovery Board, this space is reduced to 14,689,111 possibilities.

In the following, the formal experiment with Prolog and Alloy is described respectively.

B. Verification Approach Using Prolog

Target Model Design

This approach uses the logic programming language Prolog [5] to verify a given input configuration for the hardware/software interface. Prolog is a declarative language based on Horn clauses. Our target model which we derive from the tabular input specification consists of facts and an inference part. A fact in our model describes hereby a possible configuration as a mapping from the given configuration assignment to a pair consisting of a list of specific pins realizing this configuration and the associated costs like the following: `config([analog,analog],[[pa1,pa2],7]).` This fact describes that pins `pa1` and `pa2` can be used to serve two analog inputs with the associated costs of 7.

```
getConfig(RequiredConfiguration, Pair) :-
    msort(RequiredConfiguration, S),
    config(S, Pair).

allConfigs(RequiredConfiguration, Set) :-
    setof([Pins,Costs],
        getConfig(RequiredConfiguration,
        [Pins,Costs]), Set).

cheapestConfig(R, Pins, Costs) :-
    setof([Pins,Costs],
     getConfig(R, [Pins,Costs]), Set),
    Set = [_|_],
    minimal(Set, [Pins,Costs]).
```

Fig. 5. Excerpt from the inference rules from our Prolog model.

An excerpt of the inference rules is shown in Fig. 5 providing the interface to the the target model. Hereby, we have the methods to get one feasible (`getConfig/2`), all possible (`allConfigs/3`), and the best pin assignment (`cheapestConfig/3`). Due to optimization reasons, the facts and inference rules are instantiated for the particular lengths of given configurations.

Model Transformation & Constraint Mapping

To transform our input specification from the domain of possible pin assignment configurations to the target model in Prolog, we have realized the model transformation in Java. Hereby, the algorithm recursively traverses the tabular representation to create a hashmap with an ordered list of a configuration assignment as a key and a list of possible pins realizing this assignment as the associated value to the key. Due to the internal order of the used keys, the set of identified possible configurations was reduced. However, this design decision would require that the user would need to specify an ordered configuration request to find a suitable match from the facts; to relax this constraint, Prolog's function `msort/2` was incorporated to sort any request before it is actually evaluated while preserving duplicates. Furthermore, during the table traversal, the constraints as listed in Sec. II-A are obeyed to avoid self-reflexive pin assignments or resulting configurations using more pins than available.

The resulting hashmap is then iterated to create the single facts for Prolog by resolving the keys to the list of associated pins realizing this configuration. During this step, the specific costs for a concrete pin assignment are also determined. Generating the target model and applying the constraints during the traversal process took approximately 2,102.4s. These processing steps need to be done only once per hardware/software interface board since the actual model checking is realized in Prolog afterwards.

Results

In the following, the results from our experiment applying model checking with Prolog are presented. In Table I, the costs for one feasible pin assignment alongside with the Prolog computation time for different configuration lengths from 1 to 10 are shown. This table also shows the computation times for impossible configurations.

Length	Costs for feasible assignment	Computation time for feasible configuration	Computation time for impossible configuration
1	3	0s	0s
2	7	0s	0s
3	11	0s	0s
4	13	0s	0.01s
5	15	0.03s	0.02s
6	17	0.11s	0.10s
7	19	0.29s	0.30s
8	21	0.78s	0.64s
9	23	1.06s	1.06s
10	26	2.47s	1.36s
		∅ =0.474s ± 0.79s	∅ =0.349s ± 0.50s

TABLE I

PROLOG RESULTS TO CHECK BOTH POSSIBLE AND IMPOSSIBLE PIN CONFIGURATIONS FOR DIFFERENT CONFIGURATION LENGTHS.

Table II shows the results to find all possible pin assignment configurations and among them, also the best assignment in terms of costs for different configuration lengths from 1 to 10. If the identified pin assignment solution is cheaper compared to the previous table, the costs are highlighted.

C. Verification Approach Using Alloy

Target Model Design

This approach uses Alloy [9] to verify the input configuration space of the hardware/software interface. Alloy is a declarative language influenced by the Z specification language. Alloy expressions are based on first order logic and models in Alloy are amenable to fully automatic semantic analysis. However, Alloy does not perform fully exhaustive analysis of the models but rather makes reductions to gain performance.

Length	Number of all possible assignments	Costs for best assignment	Prolog computation time (all/best)
1	5	**2**	0s/0s
2	10	**4**	0s/0s
3	10	**7**	0s/0s
4	24	**9**	0.01s/0.01s
5	11	**13**	0.06s/0.03s
6	2	17	0.22s/0.11s
7	8	19	0.61s/0.30s
8	20	21	1.40s/0.64s
9	20	23	2.42s/1.08s
10	32	26	4.06s/1.38s
			$\varnothing_{all}$ = 0.878s $\pm$ 1.375s $\varnothing_{best}$ = 0.355s $\pm$ 0.51s

TABLE II

RESULTS TO CHECK FOR ALL POSSIBLE AS WELL AS THE BEST PIN ASSIGNMENT FOR DIFFERENT CONFIGURATION LENGTHS. IF A BETTER PIN ASSIGNMENT IN TERMS OF COSTS WAS FOUND COMPARED TO TABLE I, THE ENTRY IS HIGHLIGHTED.

We have used assertions in Alloy to verify whether a certain configuration is viable in the hardware/software interface board. Checking assertions results either true or false reflecting the unsatisfiability of the given predicate. If a predicate is not satisfiable, the Alloy analyzer reports counterexamples showing how the predicate is invalid.

To use Alloy for model checking, we transform the tabular input specification into an equivalent representation as described by a meta-model consisting of classes *Pin, ConnType, ConnDetail*, and *Cost* and references *conntype, conn_detail*, and *cost* originating from *Pin* with mapping cardinalities 0 - 1..*, 0 - 1..* and 1 - 1 respectively to the respective classes. Hereby, *Cost* is a derived construct originally not available in the input specification.

Model Transformation & Constraint Mapping

A given instance model conforming to the meta-model alongside with the domain constraints as listed in Sec. II-A is transformed to an Alloy specification. This instance model defines Alloy signatures for all connection types, connection details and pins available in the input specification. Two signatures from the specification are shown in Fig. 6.

```
one sig PA1 extends Pin {} {
  conntype = ANALOG + ICU + ICU
  conn_detail = ADC1_IN1 + TIM2_CH2 + TIM5_CH2
  cost = 3}

one sig PA2 extends Pin {} {
  conntype = ANALOG + SERIAL_TX + ICU + ICU
  conn_detail = ADC1_IN2 + UART2_TX +
  TIM2_CH3 + TIM5_CH3
  cost = 4}
```

Fig. 6. Alloy instance specification for two pins.

Checking Alloy assertions can find a feasible pin assignment for a given configuration. Assertions in Alloy may report counterexamples showing violations of the assertions with respect to the specification facts. Since, we want to find out a possible pin configuration, we generate assertions in Alloy assuming that the inverse statement of that request would be true. Then, we let Alloy find a counterexample, which in turn represents a possible realization of the desired configuration. An example for such a negated statement is depicted in Fig. 7.

```
assert ANALOG_ANALOG {
all disj p1, p2:Pin |
not (
   ANALOG in p1.conntype &&
   ANALOG in p2.conntype
)}

check ANALOG_ANALOG
```

Fig. 7. Generated negated assertion for the desired configuration "*ANALOG, ANALOG*".

If Alloy succeeds to find a counterexample, the variables `p1` and `p2` contain a feasible assignment to the pins of the hardware/software interface board. We have dealt with two ways of generating Alloy assertions. First, assertions for finding a feasible pin assignment for a desired configuration. This follows a trivial solution of reading and transforming the input string into Alloy expressions similar to the Fig. 7.

Second, assertions for finding the best possible solution. Alloy does not support higher order quantification to write predicates or assertions, which can automatically compute the cheapest possible pin assignment for a certain configuration of a specific length. Thus, we have generated a series of assertions where each of the assertions explores the possibility of a pin assignment for a specific total cost level. If we consider a domain of possible pin assignments with a minimum pin cost PC_{min} and maximum pin cost PC_{max}, then for a desired configuration of length l, we have generated in total $l \times PC_{max}$ - $l \times PC_{min}$ assertions. We have written a Java program to iteratively call these assertions within a cost-range starting from the cheapest possible cost for the desired configuration (i.e., $l \times PC_{min}$) to the maximum possible cost (i.e., $l \times PC_{max}$) and we stop the iteration as soon as we have found a solution.

Assertions for computing the best possible solution differ from the assertion in Fig. 7. To enable this use case, we added the expression "*p1.cost.add[p2.cost]<=X*" where *X* is taking a total cost value within the range mentioned above inside the *not()* expression of the assertion and by specifying integer bit-width in the corresponding *check* statement.

To generate the Alloy specification from the domain model, our Java program took approximately 0.3s. This step needs to be done only once per hardware/software interface board.

Results

The results of possible and impossible desired pin configuration are presented in Table III showing costs and computation time for possible and impossible configurations. Table IV shows results for all and best pin assignments for possible desired configurations. A cost in these tables is a sum of all the costs of the pins associated with the solution of the desired configuration. The sum of the costs is not automatically

processed by Alloy. However, it would be possible to post-process the output data to automatically compute the costs.

Length	Costs for the first feasible assignment	Computation time for feasible configuration	Computation time for impossible configuration
1	3	0.53s	-
2	7	0.52s	0.52s
3	11	0.56s	0.53s
4	13	0.54s	0.53s
5	15	0.56s	0.53s
6	17	0.57s	0.64s
7	19	0.62s	0.62s
8	22	0.63s	0.56s
9	23	0.65s	0.67s
10	26	0.67s	0.68s
		$\varnothing$ =0.58s $\pm$ 0.05s	$\varnothing$ =0.59s $\pm$ 0.06s

TABLE III

ALLOY RESULTS TO CHECK BOTH POSSIBLE AND IMPOSSIBLE CONFIGURATIONS FOR DIFFERENT LENGTHS.

Length	Number of all possible assignments	Costs for best assignment	Alloy computation time (all/best)
1	5	**2**	0.07s/0.53s
2	20	**4**	0.24s/0.63s
3	60	**7**	0.59s/0.67s
4	480	**9**	1.57s/1.63s
5	840	**13**	2.27s/1.20s
6	720	17	2.16s/1.17s
7	2760	19	4.68s/1.09s
8	7320	**21**	10.43s/3.25s
9	7320	23	9.27s/2.88s
10	9960	26	14.12s/3.38s
			$\varnothing_{all}$ = 4.58s $\pm$ 5.02s $\varnothing_{best}$ = 1.64s $\pm$ 1.11s

TABLE IV

RESULTS TO CHECK FOR ALL POSSIBLE AS WELL AS THE BEST PIN ASSIGNMENT FOR DIFFERENT CONFIGURATION LENGTHS. IF A BETTER PIN ASSIGNMENT IN TERMS OF COSTS WAS FOUND COMPARED TO TABLE III, THE ENTRY IS HIGHLIGHTED.

D. Analysis and Discussion

The results show that with Alloy, the growth of the computation time with respect to the increasing lengths of the desired configurations is moderate both for finding a feasible solution and for computing an impossible configuration. On the other hand, Prolog performs better on finding all and best pin assignments for a desired possible configuration. However, in the given scope of this experiment, both Prolog and Alloy not only are able to find solutions for all of the outlined use cases but also reporting the same solution with the same costs for finding the best pin assignment for a possible desired configuration.

The reason behind the surprisingly higher number of solutions reported by Alloy for all possible solutions is that the generated Alloy assertions report solutions that are not unique with respect to the pins.

From a practical point of view, finding a best pin assignment for a desired possible configuration is more valuable than feasible and all pin assignments. To find the best pin assignment, both Prolog and Alloy computation times increase by the length of the configuration. In this case, the growth of Prolog is smaller than the one from Alloy which ranks Prolog more scalable with the size of the configuration length compared to Alloy under the terms of settings for our experiment.

Furthermore, the Prolog solution provides a better user interaction in terms of taking input configuration requests and producing corresponding output. Moreover, the Prolog solution calculates the costs automatically, which is not inherently supported by Alloy but possible to achieve with work-around solutions.

Concerning the generation of the target model specification in the second stage of our workflow, Alloy takes considerably less time and space compared to Prolog. The size of the Alloy specification is less than 100KB compared to 1.7GB for Prolog. Loading the Alloy specification happens nearly instantly, while loading and compiling the target model specification for Prolog to start the model-checking process took 346.99s.

E. Threats to Validity

We discuss threats to validity to the results of our experiment according to the definition reported by Runeson and Höst [12]:

- *Construct validity.* With respect to RQ-1, the outlined approach with Prolog showed a possibility to apply model checking to verify a given configuration and to find a feasible, all possible, and the best pin assignment for a problem size, where researchers and engineers working with robotic platforms are faced with.
  As RQ-2 mentions, the solution with Alloy is also able to find a feasible, all, the and best pin assignment for a specific pin configuration. A check statement in Alloy does not guarantee that the associated assertion is invalid, if it does not report a counterexample unless the scope of the check is proper. We have taken necessary measures so that the scope always covers all possible solutions. For example, for finding the best possible pin assignment, we have introduced the bit-width of the integer in every check statement after assuring that the total costs of the resulting pin assignment would always be within the scope.
  For RQ-3, we consider the *required computation time* as the significantly influencing factor where researcher and engineers have to cope with when to find a possible pin configuration during experiments with robotic platforms. Other factors like memory consumption, experiment preparation time, reusability, or even model maintenance could have been also considered as influencing the performance. However, we have agreed on referring to the computation time only in our experiment.

- *Internal validity.* All experiments were executed on a 1.8GHz Intel Core i7 with 4GB RAM running Mac OS X 10.8.4. Furthermore, we have used the same sets of desired configurations for both RQ-1 and RQ-2. Among them, one set contains desired configurations of different lengths that are solvable and the other consists of configurations that are unsolvable.
  Concerning both RQ-1 and RQ-2, we outlined a possible solution how to utilize Prolog and Alloy for model checking. We do not claim having realized the best solution; yet, our results with respect to the required computation underline that both approaches are able to handle problem dimensions from real-world examples in an efficient way to assist researchers and developers. The results for RQ-3 might be influenced by the chosen execution platform as the varying computation times Table III suggests. However, the standard deviation for these results is rather small and thus, we consider the negative influence of other running processes on our measurements to be rather low.
- *External validity.* As the accompanying search for related work unveiled, the challenge of solving the pin assignment problem appears to be of relevance for researchers and developers dealing with robotic platforms, which interact as cyber-physical systems through sensors and actors with the surroundings. In this regard, both approaches for RQ-1 and RQ-2 outline useful ways how to address the practical problem of assigning input sources and output sinks to a hardware/software interface board. Furthermore, similar combinatorial problems, which can be expressed using either the graph-based or the tabular representation, can be solved in an analogous manner.
  The measurements and results to answer and discuss RQ-3 help researchers and developers to estimate the computational effort that must be spent to process and solve problems of a similar size and setup.
- *Reliability.* Since both outlined solutions for RQ-1 and RQ-2 depend on the design decisions met by the authors of this article, it is likely that there might be other designs to realize the model checking approaches in Alloy or Prolog, respectively. However, according to our results, our design and implementations are useful enough to be applicable to real-world sized problems. Since it was not our goal to focus on the utmost optimization for the outlined design and approaches, future work could be spent in this direction.
  With respect to RQ-3, we utilized standardized means to measure the required computation time. For Prolog, we used its standard profiling interface `profile/1` to gather data and for Alloy, *System.currentTimeMillis()* Java method to calculate the time.

IV. Related Work

This article extends our previous work on self-driving miniature vehicles [2]. Since we are focusing on the software engineering challenges [4] during the software development for this type of robotic platforms, this work is aligned with our model-based composable simulations [3] where we are trying to find the best suitable sensor setup for a specific application domain of a robotic platform before realizing it on the real platform.

The supplier of the STM32F4 Discovery Board provides a tool called MicroXplorer to assist the developer in verifying the selected pin assignment [16]. For that purpose, the user needs to select a desired pin configuration to let the tool subsequently check wether it is realizable by the microprocessor. In contrast to that with the verification approaches outlined in this article, we require the user only to specify the desired set of input sources and output sinks letting our model checkers finding a feasible, all possible, or the best pin assignment configuration. Furthermore, our verification approaches are flexible enough to also enable the merging, concatenation, and comparison of several existing configurations since both approaches depend only on the domain model, which can be accessed in a textual way.

Another tool which is freely available is called CoSmart [6] providing a similar support as the commercial one described before. However, at the time of writing, our desired hardware/software setup consisting of STM32F4 Discovery Board and Chibi/OS as real-time operating system is not supported yet. Moreover, the tool neither assists the user in finding a feasible nor the best possible pin assignment.

Other work in the domain of model checking using constraint logic programming was published e.g. by [8] and [7]. They focus on verifying that a given specification holds certain properties, while our approaches also aim for optimizing a given combinatorial problem with respect to predefined costs.

Another approach aiming for utilizing logic programming to find solutions for a pin assignment configuration problem is reported by the authors of [10]. However, their work does neither contain a description of a possible design how to realize this problem using a logical programming language nor any experimental results.

V. Conclusion and Outlook

In this article, we consider the problem of finding *a feasible*, *all possible*, or *the best* pin assignment configuration for a hardware/software interface board. This task needs to be addressed by researchers and developers dealing with embedded systems for robotic platforms to define how a set of sensors like ultra-sonic or infrared range finders and actors like steering and acceleration motors need to be connected in the most efficient way.

We have modeled the domain of possible pin configurations for such boards and analyzed its complexity. On the example of the hardware/software interface board STM32F4 Discovery Board which we are using on our self-driving miniature vehicles, we have modeled its pin configuration possibilities into a graph-based representation. To verify a desired configuration to be matched with a possible pin assignment, we traversed the graph and created an equivalent target model for the declarative languages Prolog and Alloy, respectively.

Using our example resulting in 14,689,111 configuration possibilities, we ran an experiment for the aforementioned three use cases and figured out that Alloy performs up to more than three times better finding feasible solutions for possible desired configurations and reporting insolvability of the impossible desired configurations. On the contrary, Prolog performs up to more than three times better finding all possible and best solutions for a given desired possible configuration. Moreover, the Prolog solution is more scalable with the increased configuration length which is reflected by the lower standard deviations for these use cases.

Using our Eq. 3, it can be seen that the number of possible configurations increases when either the number of pins or the number of functions per pin are increased. However, increasing the former let the size of the problem space grow significantly faster than increasing the latter. Furthermore, adding more physical pins is also a costly factor; thus, researchers and engineers will continuously have to deal with the problem of finding a feasible, all possible, or the best pin assignment configuration for their specific robotic platform.

Future work needs to be done to analyze this increasing complexity from the model checking point of view to estimate to which level of complexity instance models can still be handled properly by the model checking. Furthermore, semantic constraints like having assigned a pin for data transmission always requires another pin dealing with data receiving, need to be analyzed how they constrain the problem space and how they can be considered to optimize the target models in the particular declarative languages.

REFERENCES

[1] C. Basarke, C. Berger, K. Berger, K. Cornelsen, M. Doering, J. Effertz, T. Form, T. Gülke, F. Graefe, P. Hecker, K. Homeier, F. Klose, C. Lipski, M. Magnor, J. Morgenroth, T. Nothdurft, S. Ohl, F. W. Rauskolb, B. Rumpe, W. Schumacher, J. M. Wille, and L. Wolf. Team CarOLO - Technical Paper. Informatik-Bericht 2008-07, Technische Universität Braunschweig, Braunschweig, Germany, Oct. 2008.

[2] C. Berger, M. A. Al Mamun, and J. Hansson. COTS-Architecture with a Real-Time OS for a Self-Driving Miniature Vehicle. In E. Schiller and H. Lönn, editors, *Proceedings of the 2nd Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, Toulouse, France, Sept. 2013.

[3] C. Berger, M. Chaudron, R. Heldal, O. Landsiedel, and E. M. Schiller. Model-based, Composable Simulation for the Development of Autonomous Miniature Vehicles. In *Proceedings of the SCS/IEEE Symposium on Theory of Modeling and Simulation*, San Diego, CA, USA, Apr. 2013.

[4] C. Berger and B. Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer-Verlag, London, UK, 2012.

[5] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Berlin Heidelberg, 5th edition, 2003.

[6] CooCox. CoSmart. http://www.coocox.org/CoSmart.html, Aug. 2013.

[7] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO : A Logic Language for Executable Specifications of Real-Time Systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.

[8] G. Gupta and E. Pontelli. A Constraint-based Approach for Specification and Verification of Real-time Systems. In *Proceedings Real-Time Systems Symposium*, pages 230–239. IEEE Comput. Soc, Dec. 1997.

[9] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, Apr. 2002.

[10] B. Joshi, F. M. Rizwan, and R. Shettar. MICROCONTROLLER PIN CONFIGURATION TOOL. *International Journal on Computer Science and Engineering*, 4(05):886–891, 2012.

[11] S. L. Pfleeger. Design and Analysis in Software Engineering. *Software Engineering Notes*, 19(4):16–20, Oct. 1994.

[12] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, Dec. 2008.

[13] M. K. Sein, O. Henfridsson, S. Purao, M. Rossi, and R. Lindgren. Action Design Research. *MIS Quaterly*, 35(1):37–56, Mar. 2011.

[14] G. D. Sirio. ChibiOS/RT. http://www.chibios.org/, June 2013.

[15] STMicroelectronics. Discovery kit for STM32F407/417 line. http://goo.gl/hs7X28, Aug. 2013.

[16] STMicroelectronics. MicroXplorerMCU graphical configuration tool. http://goo.gl/3UUgdh, Aug. 2013.

Note: This is the original submission, not yet revised according to review comments.

Modeling Basic Aspects of Cyber-Physical Systems, Part II

Yingfu Zeng[1], Chad Rose[1], Paul Brauner[1], Walid Taha[1,2], Jawad Masood[2], Roland Philippsen[2], Marcia O'Malley[1], and Robert Cartwright[1,2]

[1]Rice University
[2]Halmstad University

***Abstract*— We consider the question of what language features are needed to effectively model cyber-physical systems (CPS). In previous work, we proposed a core language called Acumen as a way to study this question, and showed how several basic aspects of CPS can be modeled clearly in a language with a small set of constructs. This paper reports on the result of our analysis of two more complex case studies from the domain of rigid body dynamics. The first one, a quadcopter, illustrates that Acumen can support larger, more interesting systems than previously shown. The second one, a serial robot, provides a concrete example of why explicit support for static partial derivatives can significantly improve the expressivity of a CPS modeling language.**

I. Introduction

The increasing computational power embedded in everyday products promises to revolutionize the way we live. At the same time, the tight coupling between computational and physical mechanisms, often described as cyber-physical systems (CPSs), poses a challenge for the traditional product development cycles, particularly in physical testing. For example, car manufacturers are concerned about the amount of physical testing necessary to assure the safety of autonomous vehicles. Physical testing has been used to assess the qualities of new products for many years. One of its key ingredients is devising a collection of specific test scenarios. But the presence of even simple computational components can make it difficult to identify enough test scenarios to exercise more than a minute fraction of the possible behaviors of the system. In addition, physical testing is very expensive because it only detects flaws at the end of the product development process after physical prototypes have been constructed. These realities are spurring the CPS developers to rethink traditional methods and processes for developing and testing new products.

Computer simulations [2] performing virtual experiments [3] can be used to reduce the cost of physical tests. Virtual testing can be used to quickly eliminate obviously bad designs. It can also help build confidence that a new design can pass test scenarios developed by an independent party [4]. However, creating a framework for conducting virtual experiments requires a concerted, interdisciplinary effort to address a wide range of challenges, including: 1) educating designers in the cyber-physical aspects of the products they will develop, both in terms of how these aspects are modeled, and what types of system-level behaviors are generated; and 2) developing expressive, efficient, and robust modeling and simulation tools to support the innovation process. At each stage in the design process, the underlying models should be easy to understand and analyze. Moreover, it should be easy to deduce the mathematical relationship between the models used in successive stages.

Both challenges can be addressed by better language-based technologies for modeling and simulation. An effective model should have clear and unambiguous semantics that can readily be simulatedshowing how the model behaves in concrete scenarios.Engineering methods centered around a notion of executable or effective models can have profound positive impacts on the pace of advancement of knowledge and engineering practice.

A. The Accessibility Challenge

Designing a future smart vehicle or home requires expertise from a number of different disciplines. Even when we can assemble the necessary team of experts, they may lack a common language for discussing key issues – treated differently across disciplines – that arise in CPS design.

A critical step towards addressing this *accessibility challenge* is to agree on a *lingua franca* (or "common language") to break down artificial linguistic barriers between various scientific and engineering disciplines. Part of such a language will be a collection of technical terms from the domain of CPS that enables experts to efficiently express common model constructions; part will be effective modeling formalisms. Language research can be particularly helpful in developing tools that embody executable subsets of mathematical notations (*c.f.* [5], [6]). These are already used by engineers and scientists but are not available in mainstream tools. Because most physical phenomena are continuous and digital computation is discrete, support for hybrid (continuous/discrete) systems is important in such a language.

B. The Tool Chain Coherence Challenge

Based on our experience in several domains, it appears that scientists and engineers engaged in CPS design are often forced to transfer models through a chain of disparate,

This paper is a followup to a paper presented at the DSLRob 2012 Workshop with the same main title [1]. This second part focuses on modeling rigid body dynamics.

This work was supported by the US NSF CPS award 1136099, Swedish KK-Foundation CERES and CAISR Centres, and the Swedish SSF NG-Test Project.

specialized design tools. Each tool has a clear purpose. For example, drafting tools like AutoCAD and Solidworks support creating images of new products; MATLAB, R, and Biopython support the simple programming of dynamics and control; and finally, tools like Simulink [6] supports the simulation of device operation. Between successive tools in this chain, there is often little or no formal communication. It is the responsibility of the CPS designer to translate any data or valuable knowledge to the next tool. We believe that tool chain coherence should be approached from a linguistic point of view. Precise reasoning about models during each step of the design process can readily be supported by applying classical (programming) language design principles, including defining a formal semantics. Reasoning across design steps can be facilitated by applying two specific ideas from language design: (i) increasing the expressivity of a language to encompass multiple steps in the design process; and (ii) automatically compiling models from one step to the next. The latter idea reduces manual work and eliminates opportunities for mistakes in the translation.

C. Recapitulation of Part I

Part I of this work [1] identifies a set of prominent aspects that are common to CPS design, and shows the extent to which a small core language, which we call Acumen, supports the expression of these aspects. The earlier paper considers the following aspects:

1) Geometry and visual form
2) Mechanics and dynamics
3) Object composition
4) Control
5) Disturbances
6) Rigid body dynamics

and presents a series of examples illustrating the expressivity and convenience of the language for each of these aspects. In this narrative, the last aspect stands out as more open-ended and potentially challenging. The early paper considers a single case study involving a rigid body: a single link rod (two masses connected by a fixed-length bar). Thus, it does not address the issue of how well such a small language is suited to modeling larger rigid body systems.

D. Contributions

Modeling continuous dynamics is only one aspect of CPS designs that we may need to model, but it is important because it is particularly difficult. This paper extends previous work by considering the linguistic demands posed by two larger case studies drawn from the rigid body domain. After a brief review of Acumen (Section II), first case study we consider is a quadcopter, which is a complex, single rigid body system that is often used as a CPS example. The quadcopter case study shows that Acumen can simply and directly express Newtonian models. (Section III). The second case study is a research robot called the RiceWrist-S. In this case, developing a Newtonian model is difficult and inconvenient. It demonstrates that the more advanced technique of Lagrangian modeling can be advantageous for some problems. As a prelude to modeling the Rice Wrist-S robot, we consider two ostensibly simple dynamic systems, namely a single pendulum and a double pendulum. For the second system, we show that Lagrangian modeling leads to a much simpler mathematical formalization. (Section IV). To confirm this insight, we show how Lagrangian modeling enables us to construct a simple model of the dynamics for the RiceWrist-S(Section V). This analysis provides stronger evidence of the need to support partial derivatives and implicit equations in any hybrid-systems language that is expected to support the rigid body systems domain (Section VI).

II. Acumen

Modeling and simulation languages are an important class of domain-specific languages. For a variety of reasons, determining what are the desirable or even plausible features in a language intended modeling and simulation of hybrid systems [2] is challenging. For example, there is not only one notion of hybrid systems but numerous: hybrid systems, interval hybrid systems, impulsive differential equations (ordinary, partial), switching systems, and others. Yet we are not aware of even one standard example of such systems that has a simple, executable semantics. To overcome this practical difficulty in our analysis, we use a small language called Acumen [7], [8], which has a simple (discrete-time step) semantics. We are developing this language to apply the linguistic approach to the accessibility and tool chain coherence challenges identified above.

The language consists of a small number of core constructs, namely:

- Ground values (e.g., `True`, `5`, `1.3`, `"Hello"`)
- Vectors and matrices (e.g., `[1,2]`, `[[1,2],[3,4]]`)
- Expressions and operators on ground and composite types (+, −, ...)
- Object class (template) definitions (`class C (x,y,z) ... end`)
- Object instantiation and termination operations (`create`, `terminate`)
- Variable declarations (`private ... end`). For convenience, we included in the set of variables a special variable called `_3D` for generating 3D animations.
- Variable derivatives (`x'`, `x''`, ...) with respect to time
- Continuous assignments (=)
- Discrete assignments (`:=`)
- Conditional statements (`if`, and `switch`)

We are using this language for a term-long project in a course on CPS [9], and in the first two offerings of this course it has been received enthusiastically *c.f.* [10], [11]. A parsimonious core language can help students see the connections between different concepts and avoid the introduction of artificial distinctions between manifestations of the same concept in different contexts. This bodes well for the utility of such languages for addressing this challenge. However, to fully overcome this challenge, we must develop a clear understanding of how different features in such a language match up with the demands of different types of cyber-physical systems.

Remark: Since the writing of Part I, a minor change has been made to the syntax, where := now describes discrete assignments, and = now describes continuous assignments. Also, a syntax highlight feature has been introduced, in order to improve the user experience.

III. Quadcopter

A rigid body system consists of interconnected links with point masses, where the connections consist primarily of constraints on distances and/or angles between links. The dynamics of many physical systems can be modeled with reasonable accuracy as a rigid body system. It is widely used for describing road vehicles, gear systems, robotics, etc. In this section, we consider a system that is an example of how a complex system successfully modeled as simple rigid body, namely, the quadcopter.

A. Background

The quadcopter is a popular mechatronic system with four rotor blades to provide thrust. This robust design has seen use in many UAV applications, such as surveillance, inspection, and search and rescue. Modeling a quadcopter is technically challenging, because it consists of a close combination of different types of physics, including aerodynamics and mechanics. A mathematical model of a quadcopter may need to address a wide range of effects, including gravity, ground effects, aerodynamics, inertial counter torques, friction, and gyroscopic effects.

B. Reducing Model Complexity Through Control

Even if we limit ourselves to considering just six degrees of freedom (three for position and three for orientation), the system is underactuated (one actuation from each rotor vs. six degrees of freedom) and is therefore not trivial to control. Fortunately, controllers exist that can ensure that actuation is realized by getting the four rotors to work in pairs, to balance the forces and torques of the system. With this approach, the quadcopter can be usefully modeled as a single rigid body with mass and inertia, by taking account of abstract force, gravity and actuation control torques. This model is depicted in Fig. 1.

C. Mathematical Model

To generate the equations for the dynamics of our common quadcopter model [12], we first construct the rotational matrix to translate from an inertial (globally-fixed) reference frame to the body-fixed reference frame shown in Fig. 1. This matrix represents rotation about the y axis (θ), followed by rotation about the x axis (ϕ), and then rotation about the z axis (ψ).

$$R = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta C_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\theta & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \quad (1)$$

Here, C , S and T refer to cos, sin, and tan, respectively. Next, summing forces on the quadcopter results in:

$$\sum F = m\bar{a} = G + RT \quad (2)$$

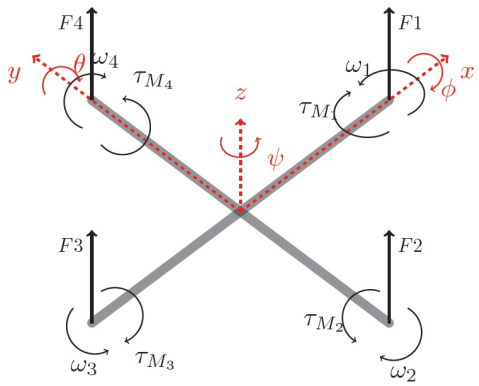


Fig. 1: Free body diagram of the quadcopter

where G is the force due to gravity, R is the rotational matrix, and T is the thrust from the motors. This expands to

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = -g \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \frac{T}{m} \begin{bmatrix} C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi S_\theta C_\phi - C_\psi S_\phi \\ C_\theta C_\phi \end{bmatrix} \quad (3)$$

Finally, by summing moments about the center of mass, the equations for the dynamics for each of the rotational degrees of freedom can be determined as follows:

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & \dot{\phi} C_\phi T_\theta + \dot{\theta} \frac{S_\phi}{C_\theta^2} & -\dot{\phi} S_\phi C_\theta + \dot{\theta} \frac{C_\phi}{C_\theta^2} \\ 0 & -\dot{\phi} S_\phi & -\dot{\phi} C_\phi \\ 0 & \dot{\phi} \frac{C_\phi}{C_\theta} + \dot{\phi} S_\phi \frac{T_\theta}{C_\theta} & -\dot{\phi} \frac{S_\phi}{C_\theta} + \dot{\theta} C_\phi \frac{T_\theta}{C_\theta} \end{bmatrix}$$

$$+ \nu + W_\eta^{-1} \dot{\nu} \quad (4)$$

Where

$$\nu = \begin{bmatrix} p \\ q \\ r \end{bmatrix} = W_\eta \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -S_\theta \\ 0 & C_\phi & C_\theta S_p hi \\ 0 & -S_\phi & C_\theta C_\phi \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad (5)$$

$$\dot{\nu} = \begin{bmatrix} (I_y y - I_z z) \frac{qr}{I_x x} \\ (I_z z - I_x x) \frac{qr}{I_y y} \\ (I_x x - I_y y) \frac{qr}{I_z z} \\ -I_r \end{bmatrix} \begin{bmatrix} \frac{q}{I_x x} \\ \frac{-p}{I_y y} \\ 0 \end{bmatrix} \omega_\Gamma + \begin{bmatrix} \frac{\tau_\phi}{I_x x} \\ \frac{\tau_\theta}{I_y y} \\ \frac{\tau_\psi}{I_z z} \end{bmatrix} \quad (6)$$

$$\begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} lk\left(-\omega_2^2 + \omega_4^2\right) \\ lk\left(-\omega_1^2 + \omega_3^2\right) \\ \sum_{i=1}^{4} \tau_{M_i} \end{bmatrix} \quad (7)$$

D. Acumen Model for Quadcopter

The equations derived earlier for the dynamics can be expressed in our core language as follows:

```
class QuadCopter(P,phi,theta,psi)
private
  g := 9.81;      m := 0.468;
  l := 0.225;     k := 2.98*10^(-6);
  b := 1.140*10^(-7);
```

```
  IM := 3.357*10^(-5);
  Ixx := 4.856*10^(-3);
  Iyy := 4.856*10^(-3);
  Izz := 8.801*10^(-3);
  Ax  := 0.25;  Ay  := 0.25;
  Az  := 0.25;
  w1 := 0;  w2:= 0;   w3 := 0;
  w4 := 0;  wT := 0;  f1 := 0;
  f2 := 0;  f3 := 0;  f4 := 0;
  TM1 := 0; TM2 := 0; T := 0;
  TM3 := 0; TM4 := 0;
  P' := [0,0,0];    P'' := [0,0,0];
  phi' := 0; theta' := 0; psi' := 0;
  phi'' := 0; theta'' := 0; psi'' := 0;
  p := 0; q := 0; r := 0; p' := 0;
  q' := 0; r' := 0; Ch:=0; Sh:=0;
  Sp:=0; Cp:=0; St:=0; Ct:=0; Tt:=0
end
  T = k* (w1^2 + w2^2 + w3^2 + w4^2);
  f1 = k * w1^2; TM1 = b * w1^2;
  f2 = k * w2^2; TM2 = b * w2^2;
  f3 = k * w3^2; TM3 = b * w3^2;
  f4 = k * w4^2; TM4 = b * w4^2;
  wT = w1 - w2 + w3 - w4;

  Ch = cos(phi); Sh = sin(phi);
  Sp = sin(psi); Cp = cos(psi);
  St = sin(theta); Ct = cos(theta);
  Tt = tan(theta);

  P'' = -g * [0,0,1] + T/m
   *[Cp*St*Ch+Sp*Sh,Sp*St*Ch-Cp*Sh,Ct*Ch]
   -1/m*[Ax*dot(P',[1,0,0]),
         Ay*dot(P',[0,1,0]),
         Az*dot(P',[0,0,1])];
  p' = (Iyy-Izz)*q*r/Ixx - IM*q/Ixx*wT
   + l*k*(w4^2 - w2^2)/Ixx;
  q' = (Izz-Ixx)*p*r/Iyy-IM*(-p)/Iyy*wT
   + l*k*(w3^2 -w1^2)/Iyy;
  r' = (Ixx - Iyy)*p*q/Izz  + b*(w1^2
   + w2^2 -w3^2 -w4^2)/Izz;
  phi'' = (phi'*Ch*Tt+ theta'*Sh/Ct^2)*q
   + (-phi'*Sh*Ct+theta'*Ch/Ct^2)*r
   + (p'+q'*Sh*Tt+r'*Ch*Tt);
  theta'' = (-phi'*Sh)*q + (-phi'*Ch)*r
   + (q'*Ch+r'*(-Sh));
  psi'' = (phi'*Ch/Ct+phi'*Sh*Tt/Ct)*q
   + (-phi'*Sh/Ct+theta'*Ch*Tt/Ct)*r
   + (q'*Sh/Ct+r'*Ch/Ct);
end
```

Fig. 2 presents snapshots of a 3D visualization of the quadcopter responding to a signal from a basic stabilizing controller [12]. This example shows that the Acumen core language can express models for complex mechatronic systems that are widely used in both research and education today.

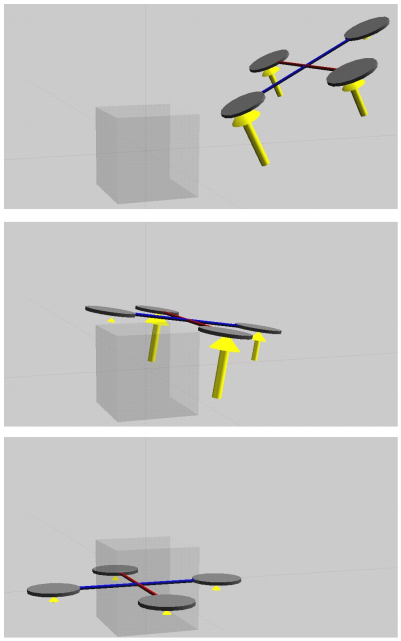


Fig. 2: The simulation results of the quadcopter model with PID control. Here the controller is bringing the quadcopter elevation, roll, pitch, and yaw to zero. Yellow arrows attached to each rotor indicate the thrust force generated by the rotors.

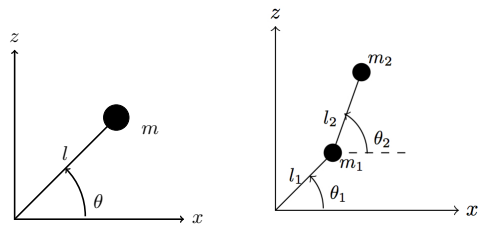
IV. Lagrangian Modeling

Mathematical modeling of rigid body systems draws heavily on the field of classical mechanics. This field started in the 17th century with the introductions of Newton's principles of motion and Newtonian modeling. Newton's foundational work was followed by Lagrangian modeling in the 18th century, and Hamiltonian modeling in the 19th. Today, mechanical engineers make extensive use of the Lagrangian method when analyzing rigid body systems. It is therefore worthwhile to understand the process that engineers follow when using this method, and to consider the extent to which a modeling language can support this process.

The Lagrangian method is based on the notion of *action* $L = T - V$, which is the difference between the kinetic T and potential energy V. The Euler-Lagrange equation is itself a condition for ensuring that the total action in the system is stationary (constant). In Lagrangian modeling of physical systems, this condition should be seen as the analogy of the combined conditions $\Sigma F = ma$ and $\Sigma\tau = I\omega''$ in Newtonian mechanics. The Euler-Lagrange equation is as follows:

$$\forall i \in \{1...|q|)\}, \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q \qquad (8)$$

Using just this equation, the modeling process is reduced to specifying the kinetic and potential energy in the system. Part of the power of the method comes from the fact that this can be done using cartesian, polar, spherical, or any other "generalized coordinate system". Compared to the classic



(a) Single (b) Double

Fig. 3: Free body diagram for single and double pendulums

Newtonian, force-vector based methods, the Euler-Lagrange equation is often a more direct specification of the dynamics.

The Lagrangian modeling process consists of four steps:

1) Start with a description of the components of the system, consisting of masses and joints. This description generally comes with a set of variable names which are collectively called called the generalized coordinates vector q. Intuitively, each variable represents a quantity corresponding to one of the degrees of freedom for the system. Usually, all of this information can be captured in an intuitive way in a *free body diagram*.
2) Define the total kinetic and potential energy T and V, respectively, of the system, in terms of the selected set of generalized coordinates q.
3) Identify and include any "non-conservative forces" Q such as friction.[1]
4) Substitute the values into the Euler-Lagrange equation (8) for the variables of second the derivative of q.

This process and its benefits can be illustrated with two small examples. Figure 3 presents a free body diagram marked up with generalized coordinates (θ in one, and θ_1, θ_2 in the other) for a single and a double pendulum system. First, we consider the single pendulum. A direct application of the angular part of Newton's law gives us the following equation:

$$\ddot{\theta} = \frac{g}{l}\cos\theta \tag{9}$$

which is easily expressed in Acumen as follows:

```
class pendulum (l)
private
  theta := 0;theta':=0;theta'':=0;
  g:=9.81;
end
  theta'' = g/l*cos(theta);
end
```

Lagrangian modeling can be applied to the single pendulum problem, but Newton's method works well enough here. However, Lagrangian modeling does pay off for a double pendulum. It is highly instructive for language designers to recognize that such a small change makes the model most of us learn about in high-school much more cumbersome than necessary. Whether or not this is due to weakness in Newtonian modeling or intrinsic complexity in this seemingly simple example is not obvious: The double pendulum is sophisticated enough to be widely used to model a human standing or walking [13], or a basic two-link robot such as the MIT-manus [14].

To derive a model for the double pendulum using Lagrangian modeling, we proceed as follows:

1) We take $q = (\theta_1, \theta_2)$. Here, because the Euler-Lagrange equation is parameterized a generalized coordinate vector, we could have chosen to use Cartesian coordinates (x, z) for each of the two points. Here we chose angles because they resemble what can be naturally measured and actuated at joints.
2) The kinetic and potential energies are defined as follows:

$$T = \frac{1}{2}m_1 v_1^2 + \frac{1}{2}m_2 v_2^2 \tag{10}$$

$$V = m_1 g z_1 + m_2 g z_2 \tag{11}$$

where we have introduced shorthands for speeds $v_1^2 = l_1^2 \dot{\theta_1}^2$ and $v_2^2 = v_1^2 + \frac{1}{2}m_2(l_1^2\dot{\theta_1}^2 + l_2^2\dot{\theta_2}^2 + 2l_1 l_2 \dot{\theta_1}\dot{\theta_2}\cos(\theta_2 - \theta_1))$, $z_1 = l_1 sin\theta_1$ and heights $z_2 = z_1 + l_2 sin\theta_2$. Substituting (or inlining) these terms we get:

$$T = \frac{1}{2}m_1(l_1\dot{\theta_1})^2 + \frac{1}{2}m_2(l_1^2\dot{\theta_1}^2 + l_2^2\dot{\theta_2}^2 + 2l_1 l_2\dot{\theta_1}\dot{\theta_2}\cos(\theta_2 - \theta_1)) \tag{12}$$

$$V = m_1 g l_1 \sin\theta_1 + m_2 g l_2 \sin\theta_2 + m_2 g l_1 \sin\theta_1; \tag{13}$$

3) We assume frictionless joints, and so there are no non-conservative forces ($Q = 0$).
4) By substitution and differentiation we get:

$$(m_1 + m_2)l_1^2\ddot{\theta_1} + m_2 l_1 l_2 \ddot{\theta_2}\cos(\theta_1 - \theta_2) + m_2 l_1 l_2 \dot{\theta_1}^2 \sin(\theta_1 - \theta_2) + m_1 m_2 g l_1 \cos\theta_1 = 0 \tag{14}$$

$$m_2 l_2^2 \ddot{\theta_2} + m_2 l_1 l_2 \ddot{\theta_1}\cos(\theta_1 - \theta_2) - m_2 l_1 l_2 \dot{\theta_1}^2 \sin(\theta_1 - \theta_2) + m_2 g l_2 \cos\theta_2 = 0 \tag{15}$$

It is important to note in this case that, while these are ordinary differential equations (ODEs), they are not in the explicit form $X' = E$. Rather, they are in implicit form, because the variable we are solving for (X') is not alone on one side of the. Using Gaussian elimination (under the assumption that the masses and length are strictly greater than zero), we get:

$$\ddot{\theta_1} = m_2 l_2(\ddot{\theta_2}\cos(\theta_1 - \theta_2) + \dot{\theta_2}^2 \sin(\theta_1 - \theta_2) + (m_1 + m_2)g\cos(\theta_1))/(-l_1(m_1 + m_2)) \tag{16}$$

$$\ddot{\theta_2} = m_2 l_1(\ddot{\theta_1}\cos(\theta_1 - \theta_2) - \dot{\theta_1}^2 \sin(\theta_1 - \theta_2) + m_2 g\cos(\theta_2))/ - l_1 m_2 \tag{17}$$

[1] For this paper, we do not consider any such forces.

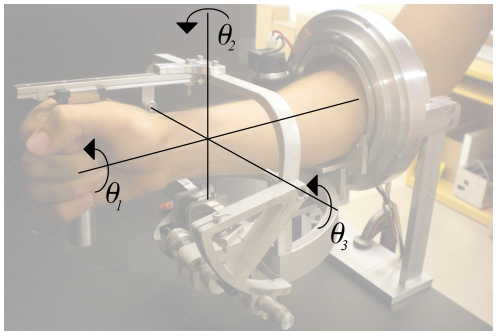


Fig. 4: The RiceWrist-S, with superimposed axes of rotation

The following code shows these equations expressed in Acumen:

```
class double_pendulum (m_1, m_2, L_1, L_2)
private
  t_1 := 0; t_2 := 0;
  t_1' := 0; t_2' := 0;
  t_1'' := 0; t_2'' := 0;
  g:=9.81;
end
  t_1'' =
   (m_2*L_2*(t_2''*cos(t_1-t_2)
             +t_2'^2*sin(t_1-t_2))
    + (m_1+m_2)*g*cos(t_1))
   *(-1)/((m_1+m_2)*L_1);
  t_2'' =
   (m_2*L_1*(t_1''*cos(t_1-t_2)
             -t_1'^2*sin(t_1-t_2))
    +m_2*g*cos(t_2))
   *(-1)/(m_2*L_2);
end
```

V. The RiceWrist-S Robot

Equipped with an understanding of Lagrangian modeling in the manner presented above, engineers model multi-link robots much more directly than with the Newtonian method. In this section we present one such case study using the RiceWrist-S research Robot.

A. Background

With an increasing number of individuals surviving once fatal injuries, the need for rehabilitation of damaged limbs is growing rapidly. Each year, approximately 795,000 people suffer a stroke in the United States, where stroke injuries are the leading cause of long-term disability. The RiceWrist-S [15] is an exoskeleton robot designed to assist in the rehabilitation of the wrist and forearm of stroke or spinal cord injury patients (Fig. 4). It consists of a revolute joint for each of the three degrees of freedom at the wrist. Because it has three rotational axes intersecting at one point, a good starting point to modeling it is the the gimbal, a commonly studied mechanical device that also features several rotational axes intersecting at one point.

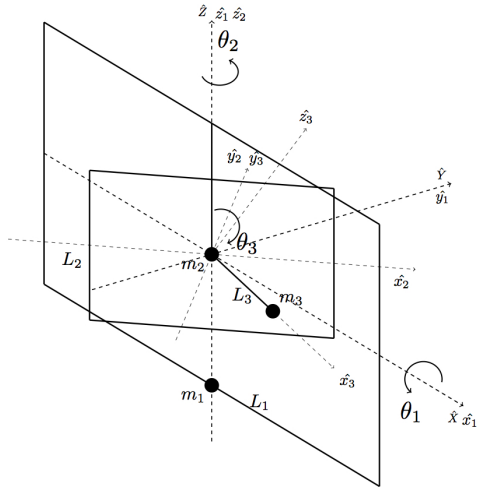


Fig. 5: Free body diagram of the RiceWrist-S as a gimbal

B. Analytical Model

We can apply the Lagrangian modeling process to determine the dynamics of a gimbal as follows:

1) We take $q = (\theta_1, \theta_2, \theta_3)$, where each of the angles corresponds to one of the three rotations possible in the RiceWrist-S Fig. 5. We chose to represent the mass of the system as centralized to three locations, one at the origin, one at the bottom of the outermost ring, and one at the end of the third link. The masses in this figure correspond to the motors and handle depicted in Fig. 4.
2) To define the energies concisely, it is convenient to use the following angular velocities of the gimbal frames definitions for the kinetic energy terms, and the subsequent heights for the potential energy terms:

$$\omega_1 = \dot{\theta_1} \cdot \hat{x_1} \tag{18}$$
$$\omega_2 = \dot{\theta_1} \cdot \hat{x_1} + \dot{\theta_2} \cdot \hat{z_2} \tag{19}$$
$$\omega_3 = \dot{\theta_1} \cdot \hat{x_1} + \dot{\theta_2} \cdot \hat{z_2} + \dot{\theta_3} \cdot \hat{y_3} \tag{20}$$

where $\hat{x_i}\hat{y_i}\hat{z_i}$ refers to the unit vector and coordinate frame about which these rotations occur, as shown in Fig. 5. Here, the ω_i terms correspond to the m_i masses, and describe the angular velocities of that mass. Since this is a complex rotational system, many of the rotations do not occur in the coordinate frames of the respective gimbal. Therefore, in order to express each ω_i in terms of the same coordinate frame, we applied the following coordinate transforms:

$$\omega_1 = \dot{\theta_1} \cdot \hat{x_1} \tag{21}$$
$$\omega_2 = \dot{\theta_1}(\cos(\theta_2) \cdot \hat{x_2} - \sin(\theta_2) \cdot \hat{y_2}) + \dot{\theta_2} \cdot \hat{z_2} \tag{22}$$
$$\omega_3 = (\dot{\theta_1}(\cos(\theta_2)\cos(\theta_3)) + \dot{\theta_2}(-\sin(\theta_3))) \cdot \hat{x_3} + (-\dot{\theta_1}\sin(\theta_2) + \dot{\theta_3}) \cdot \hat{y_3} + (-\dot{\theta_1}\sin(\theta_3)\cos(\theta_2) - \dot{\theta_2}\cos(\theta_3)) \cdot \hat{z_3} \tag{23}$$

Next, we define the heights above the predefined plane of zero potential energy (in Fig. 5, the XY plane) of each of the masses m_1, m_2, m_3, respectively, as the following:

$$h_1 = l_2 \cos(\theta_1) \quad (24)$$
$$h_2 = 0 \quad (25)$$
$$h_3 = l_3 \sin(\theta_1) \sin(\theta_3) \quad (26)$$

With this completed, the T and V terms can be quickly and easily defined. Since this is a rotational only system, T is defined as the sum of the rotational energy terms, shown below:

$$T = \frac{1}{2}(I_1\omega_1 \cdot \ \omega_1 + I_2\omega_2 \cdot \omega_2 + I_3\omega_3 \cdot \omega_3) \quad (27)$$

Where I_i is the rotational inertia corresponding to θ_i, and ω_i is defined as above. And since there are no potential energy storage elements other than those caused by gravity, V can be expressed with these heights:

$$V = m_1 g h_1 + m_2 g h_2 + m_3 g h_3 = -m_1 g l_2 \cos(\theta_1) + m_3 g l_3 \sin(\theta_1) \sin(\theta_3) \quad (28)$$

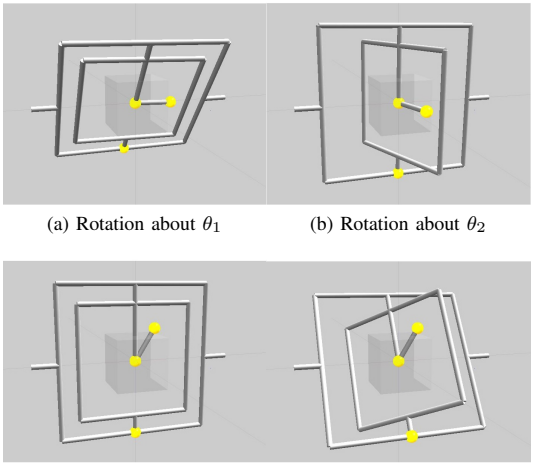
3) Again, we assumed frictionless joints, so $Q = 0$
4) After substitution and differentiation, we get an implicit set of equations. We solve these for q'' to get the equations of the systems dynamics.

$$\frac{1}{2}\frac{d}{dt}\left(I_1\frac{\partial\omega_1 \cdot \omega_1}{\partial\dot{\theta_1}} + I_2\frac{\partial\omega_2 \cdot \omega_2}{\partial\dot{\theta_1}} + I_3\frac{\partial\omega_3 \cdot \omega_3}{\partial\dot{\theta_1}}\right) + m_1 g l_2 sin(\theta_1) + m_3 g l_3 cos(\theta_1) sin(\theta_3) = 0 \quad (29)$$

$$\frac{1}{2}\frac{d}{dt}\left(I_2\frac{\partial\omega_2 \cdot \omega_2}{\partial\dot{\theta_2}} + I_3\frac{\partial\omega_3 \cdot \omega_3}{\partial\dot{\theta_2}}\right) - \frac{1}{2}\left(I_2\frac{\partial\omega_2 \cdot \omega_2}{\partial\theta_2} + I_3\frac{\partial\omega_3 \cdot \omega_3}{\partial\theta_2}\right) = 0 \quad (30)$$

$$\frac{1}{2}\frac{d}{dt}\left(I_3\frac{\partial\omega_3 \cdot \omega_3}{\partial\dot{\theta_3}}\right) - \frac{1}{2}\left(I_3\frac{\partial\omega_3 \cdot \omega_3}{\partial\theta_3}\right) - m_3 g l_3 sin(\theta_1) cos(\theta_3) = 0 \quad (31)$$

The resulting equations are, again, easy to express in Acumen, although they are too long to include here, as is the code [16]. What is really interesting about this situation is that these equations are linear in $\ddot{q}$ even though the system is also a non-linear differential equation (when we consider q, $\dot{q}$ and $\ddot{q}$). Because the system is linear in $\ddot{q}$, we can use standard Gaussian elimination to solve for $\ddot{q}$. This converts these implicit equations into an explicit form that is readily expressiable in Acumen. Fig. 6 shows the compound motion of the RW-S Gimbal model.



(a) Rotation about θ_1 (b) Rotation about θ_2

(c) Rotation about θ_3 (d) Compound motion

Fig. 6: RiceWrist-S modeled as Gimbal in Acumen.

VI. Discussion

At the highest level, Acumen can be seen as an untyped core formalism for subsets hybrid systems modeling and simulation languages such as Modelica [17], Scicos [18] and Simcape [19]. It is easy to see that the CPS aspects that Acumen can express should be expressible by "larger" languages, but it is less clear what finding a weakspot in the expressivity of Acumen means for larger languages. Here, it helps that Acumen is untyped, because static typing would clearly add restrictions that could distract from relating to other languages. But removing types does not remove all the difficulties.

With several annual workshops related to its design and applications, Modelica is the most actively studied of the hybrid languages. It is a large language with many constructs. Acumen is small, with only a few features that are specific to hybrid systems. For example, Open Modelica's parser [20] is roughly four times longer than the parser for Acumen. If Acumen can be seen as a subset of Modelica, it is reasonable to assume that the CPS aspects that Acumen can express can also be expressed in Modelica. At the same time, if features of CPS models are not easily expressed in Acumen, it is not immediately obvious wether or not they can be expressed in Modelica.

The experience with the RiceWrist-S Robot suggests the need for Lagrangian modeling, which in turn points out to the need for two language features. The first feature is *static* partial derivatives. It has been observed elsewhere that the type of partial derivatives used in the Euler-Lagrange equation for rigid body dynamics can be removed at compile time (or "statically") [6]. What is new here is that the pendulum examples pinpoint a concrete transition point for *when* Lagrangian modeling is needed in terms of a feature of the problem domain, namely, the number of links in the system.

Support for static partial derivatives in larger modeling

and simulation seems sparse. While the Modelica standard (at least until recently) did not support partial derivatives, one implementation of Modelica, Dymola [21], does provide this support. Static partial derivatives are clearly an important feature to include in any DSL aiming to support CPS design.

In response to this observation, we have already introduced support for static partial derivatives in Acumen. The second language feature required to support Lagrangian modeling, as illustrated by the double pendulum example,is support for implicit equations. This feature is present in Modelica and Simscape. Its utility is significant and goes beyond just supporting Lagrangian modeling, because it can increase the reusability and flexibility of models, making composite models more concise. At this point it is not entirely clear to us how to best introduce implicit equations into a core language. To address this issue, we are currently implementing and experimenting with several different approaches from the literature. While different approaches may be better suited for different purposes (such as numerical precision, runtime performance, or others), our purpose behind this activity is to better understand how to position implicit equations in the context of other CPS DSL design decisions. A particularly interesting question here is whether this feature must be a primitive in the language or whether it can be treated as a conservative extension [22] of a core that does not include it.

VII. Conclusions

This paper reports our most recent findings as we pursue an appropriate core language for CPS modeling and simulation. In particular, we present two examples. Our first example, a quadcopter, is significantly more sophisticated than any system considered in the first part. It shows that the core language proposed in Part I can express more sophisticated systems than was previously known. The second example, the RiceWrist-S, led us to understand the need for Lagrangian modeling. The most important lesson from this case study was that a DSL aimed at supporting CPS design should provide static partial derivatives and implicit equations. In order to ensure that we can push this line of investigation further, we have already developed a prototype of the former, and are working to realize the latter.

Acknowledgements

We would like to thank the reviewers of DSLRob 2012 for valuable feedback on an earlier draft of this paper, and Adam Duracz for comments and feedback on this draft.

References

[1] Taha Walid and Philippsen Roland. Modeling Basic Aspects of Cyber-Physical Systems. *arXiv preprint arXiv:1303.2792*, 2013.

[2] Luca Carloni and Roberto Passerone and Alessandro Pinto and Alberto Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Design Automation*, 1(1):1–204, 2006.

[3] Julien Bruneau, Charles Consel, Marcia O'Malley, Walid Taha, and Wail Masry Hannourah. Virtual Testing for Smart Buildings. In *Proceedings of the 8th International Conference on Intelligent Environments (IE'12)*, Guanajuato's, Mexico, 2012.

[4] Jeff Jensen, Danica Chang, and Edward Lee. A Model-Based Design Methodology for Cyber-Physical Systems. In *Proceedings of The First IEEE Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy'11)*, Istanbul, Turkey, July 2011.

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[6] Zhu Yun, Westbrook Edwin, Inoue Jun, Chapoutot Alexandre, Salama Cherif, Peralta Marisa, Martin Travis, Taha Walid, O'Malley Marcia, and Cartwright Robert. Mathematical equations as executable models of mechanical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 1–11. ACM, 2010.

[7] Walid Taha, Paul Brauner, Yingfu Zeng, Robert Cartrwright, Veronica Gaspes, Aaron Ames, and Alexandre Chapoutot. A Core Language for Executable Models of Cyber Physical Systems (Preliminary Report). In *Proceedings of The Second International Workshop on Cyber-Physical Networking Systems (CPNS'12)*, Macau, China, June 2012.

[8] Acumen website. www.acumen-language.org, 2010.

[9] Walid Taha. Lecture notes on cyber-physical systems. Available online from www.effective-modeling.org/p/teaching.html, September 2012.

[10] Taha Walid, Cartwright Robert, Philippsen Roland, and Zeng Yingfu. A First Course on Cyber Physical Systems. In *2013 Workshop on Embedded and Cyber-Physical Systems Education (WESE)*, Montreal, Canada, October 2013.

[11] Taha Walid, Cartwright Robert, Philippsen Roland, and Zeng Yingfu. A First Course on Cyber Physical Systems. In *Proceedings of the First Workshop on Cyber-Physical Systems Education (CPS-Ed 2013) at Cyber Physical Systems Week (CPSWeek 2013)*, Philadelphia, Pennsylvania, USA, April 2013. Available: http://cps-vo.org/group/edu/workshop/proceedings2013.

[12] Teppo Luukkonen. Modelling and control of quadcopter, 2011.

[13] Pekarek David, Ames D. Aaron, and Marsden E. Jerrold. Discrete mechanics and optimal control applied to the compass gait biped. In *Decision and Control, 2007 46th IEEE Conference on*, pages 5376–5382. IEEE, 2007.

[14] Neville Hogan, Hermano Igo Krebs, J Charnnarong, P Srikrishna, and Andre Sharon. MIT-MANUS: a workstation for manual therapy and training. i. In *Robot and Human Communication, 1992. Proceedings., IEEE International Workshop on*, pages 161–165. IEEE, 1992.

[15] A.U Pehlivan, Lee Sangyoon, and M.K. O'Malley. Mechanical design of RiceWrist-S: A forearm-wrist exoskeleton for stroke and spinal cord injury rehabilitation. In *Biomedical Robotics and Biomechatronics (BioRob), 2012 4th IEEE RAS EMBS International Conference on*, pages 1573–1578, 2012.

[16] Yingfu Zeng, Chad Rose, and Walid Taha. RiceWrist-S: Gimbal model. available online from. http://bit.ly/RiceWristGimbal, 2013.

[17] Fritzson Peter and Bunus Peter. Modelica A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation. In *SS '02: Proceedings of the 35th Annual Simulation Symposium*, page 365, Washington, D.C., USA, 2002. IEEE Computer Society.

[18] Ramine Nikoukhah. Modeling hybrid systems in Scicos: a case study. In *Proceedings of the 25th IASTED international conference on Modeling, indentification, and control*, MIC'06. ACTA Press, 2006.

[19] Inc. The Mathworks. Simscape documentation. http://www.mathworks.com/help/physmod/simscape/, 2013.

[20] Modelica. parser. https://www.modelica.org/tools/parser/ModelicaParser/parser.

[21] H. Olsson, H. Tummescheit, and H. Elmqvist. Using automatic differentiation for partial derivatives of functions in Modelica. In *Proceedings of Modelica2005*, pages 105–112, Hamburg, Germany , March.

[22] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

Author Index