RDIS: A DOMAIN MODEL FOR GENERALIZING THE MAPPINGS BETWEEN ROBOTIC SOFTWARE FRAMEWORKS AND ROBOTIC DEVICES

by

PAUL KILGO

MONICA ANDERSON, COMMITTEE CHAIR
JEFF GRAY
QI HAO
EUGENE SYRIANI

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2012

Copyright Paul Kilgo 2012
ALL RIGHTS RESERVED

ABSTRACT

Researchers, hobbyists, and industrial professionals alike have moved toward a framework-based approach to robotics application development. This allows robotic applications to be written for the domain of the framework and benefits the application developer as it offers domain-specific abstractions and reusability of applications. However, a framework must still know how to translate its domain-specific concepts to the device-specific concepts. One can do this in a general-purpose programming language, but this strategy is not sustainable because drivers must be hand-crafted for each possible permutation of framework and device. Therefore, exploring the appropriate level of abstraction for device drivers is interesting and may enable a description of a robot for it to be used with any robotics framework.

A domain model for defining these mappings has been developed. The domain model allows descriptions of robots to be reused between frameworks. The specific mechanism for this is to describe the domain model using a textual syntax and interpret the description at run-time. One may then define the transformation of framework domain concepts to concepts native to the domain model as an adapter. Thus, any appropriately enabled robotic framework and device can communicate with one another. As further evidence for the viability of the domain model, it was implemented formally using the model-driven tool AToM^3. One may then design the model at a high-level directly in the terms introduced by the model without the mental load posed by the textual syntax. As well, the tool presents a preliminary approach for generating adapters for frameworks. Future directions for the domain model include kinematic state modeling. A literature review and a preliminary approach for this has also been prepared.

DEDICATION

This thesis is dedicated to the network of support I have had during my graduate career, including friends, family, and colleagues. Their patience, mentoring, and encouragement is what has made this research possible.

LIST OF ABBREVIATIONS AND SYMBOLS

AToM^3 A Tool for Multi-formalism Meta-Modeling.

API Application Programming Interface.

ASG Abstract Syntax Graph.

AST Abstract Syntax Tree.

ATL ATL Transformation Language.

COMR Closeness Of Mapping Ratio.

DSL Domain-Specific Language.

DSML Domain-Specific Modeling Language.

EMF Eclipse Modeling Framework.

ER Entity-Relationship.

GUI Graphical User Interface.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

LHS Left-Hand Side.

MBSD Model-Based Software Development.

MDA Model-Driven Architecture.

MDE Model-Driven Engineering.

OCL Object Constraint Language.

ODE Open Dynamics Engine.

OGRE Open Graphics Rendering Engine.

OMG Object Management Group.

P2OS Pioneer 2 Operating System.

PCRE Perl-Compatible Regular Expression.

PEP Python Enhancement Proposal.

PIM Platform-Independent Model.

PREOP Providing Robotic Experiences Through Object-Based Programming.

PSM Platform-Specific Model.

QVT Query/View/Transformation.

RDIS Robot Device Interface Specification.

RDTF Robotics Domain Task Force.

RHS Right-Hand Side.

ROS Robot Operating System.

SDF Simulation Description Format.

SDK Software Development Kit.

SLOC Source Lines Of Code.

UML Unified Modeling Language.

URDF Uniform Robot Description Format.

VRML Virtual Reality Markup Language.

ACKNOWLEDGMENTS

Any appearance of expertise on this subject is no doubt due to someone else pouring hours of their time into helping me understand. That said, I am greatly indebted to the members of my thesis committee. Monica Anderson has spent a great deal of the fall semester helping me get a grasp on the project with a crash course in robotics and the finer points of writing drivers. Eugene Syriani has taught me everything I know about the theory and practice of modeling. Jeff Gray has offered many good ideas for evaluation and model metrics. Qi Hao has as well given me many good ideas from his expertise.

I would also like to acknowledge the Department of Computer Science at The University of Alabama for financial support for both research and teaching during my graduate career, and for the opportunity to travel and present this research to the robotics community.

CONTENTS

ABSTRACT ii

DEDICATION iii

LIST OF ABBREVIATIONS AND SYMBOLS iv

ACKNOWLEDGMENTS vi

LIST OF TABLES xii

LIST OF FIGURES xiii

1 INTRODUCTION 1

1.1 Robot Device Interface Specification 3

1.2 Contributions 5

1.2.1 Definition of the RDIS Model 6

1.2.2 A Model-driven Approach for Describing Robots 6

1.2.3 A Survey and Analysis of Kinematics Description 7

1.3 Limitations and Assumptions 7

1.4 Conclusion 8

2 MODELING AND ROBOTICS: A LITERATURE REVIEW 10

2.1 Application and Control Modeling 10

2.1.1 Pure Modeling and Robotics 10

2.1.2 Domain-Specific Languages for Robot Applications 11

2.2 Kinematic Modeling 12

2.2.1 Pure Modeling for Simulation Description 12

2.2.2 Domain-Specific Languages for Simulation Description 12

2.3 Model Metrics 13

3 TOWARD A MODEL-DRIVEN APPROACH 15
3.1 Model-driven Software Development and Domain-Specific Modeling . . . 15
3.1.1 Meta-modeling 16
3.1.2 Code Generation 18
3.2 Models Versus Text 18
3.2.1 Graph-like Versus Tree-like 18
3.2.2 Editing 19
3.2.3 Constraints 20
3.2.4 Maintenance 20
3.2.5 Drawbacks 21
3.3 Tools for Model-driven Development 21
3.3.1 AToM^3 22
3.3.2 Eclipse Modeling Framework 22
3.3.3 MATLAB/Simulink 23

4 THE RDIS DOMAIN MODEL 24
4.1 Introduction 24
4.1.1 The High-level Domain Model 24
4.1.2 The Approach for Implementation 25
4.1.3 The Framework-RDIS Interface 25
4.2 Revisions to RDIS 25
4.2.1 The Model 25
4.2.2 The Approach 27
4.3 State Variables 28
4.3.1 Type System 28
4.3.2 Scope 29
4.3.3 Initialization and Default Values 29
4.3.4 Future Deprecation 29
4.4 Connection 30
4.4.1 Startup, Keepalive, and Terminate Interfaces 30

4.5 Serial Connections 30
4.5.1 Baud 30
4.5.2 About Port 30
4.5.3 Unmodeled Attributes 30
4.6 Threading 31
4.7 Single Threading 31
4.7.1 Frequency 31
4.8 Other Threading Models 31
4.9 Primitives 31
4.9.1 Parameters 32
4.9.2 Format Arguments 32
4.9.3 Post-actions 32
4.9.4 On Variable-length Messages 32
4.10 Local Interface 33
4.10.1 Parameters 33
4.10.2 Types 33
4.10.3 Primitive Calls 33
4.11 Domain Adapter 34
4.11.1 Range 34
4.11.2 Differential Speed 34
4.12 Domain Interface 34
4.12.1 Incoming Domain Adapter 35
4.13 Domain Output 35
4.13.1 Outgoing Domain Adapter 36
4.14 Mini-language 37
4.15 Evaluation 38
4.15.1 Abstraction 38
4.15.2 Understandability 39
4.15.3 Accuracy 40

4.15.4 Predictiveness 42
4.15.5 Inexpensive 42
4.16 Conclusion 43

5 IMPLEMENTATION OF RDIS FOR AToM^3 45
5.1 Introduction 45
5.1.1 Modeling the Framework 46
5.2 Solution 46
5.2.1 The RDIS Formalism 47
5.2.2 Background for RDIS 48
5.2.3 The ROS Formalism 49
5.2.4 Code Generators 53
5.2.5 ROS Code Generator 54
5.2.6 RDIS Interpreter 55
5.2.7 Style 56
5.3 Evaluation 57
5.3.1 Modeling Speedup 57
5.3.2 Maintenance and Refactoring 58
5.3.3 Limitations and Assumptions 58
5.4 Conclusion 59

6 ON ROBOT KINEMATIC DESCRIPTION 60
6.1 Problem Description 61
6.2 Motivation 62
6.3 Description Languages of Simulators 62
6.3.1 Analysis 66
6.4 Approach 69
6.4.1 Link 71
6.4.2 Joint 72
6.4.3 Transmission 73

7 CONCLUSION 76
7.1 Summary of Contributions 76
7.1.1 Domain Model 76
7.1.2 An Example Model-Driven Implementation 77
7.1.3 A Design for Kinematic Modeling 77
7.2 Future Work 77
7.2.1 Framework Modeling 77
7.2.2 Threading 78
7.2.3 Sensor and Actuator Error Modeling 78
7.2.4 Sensor Modeling 78
7.2.5 Framework-RDIS Interface 78
7.2.6 Kinematic Modeling 79
7.3 Impact 79

REFERENCES 81

A THE RDIS INTERPRETER 87

B AN EXAMPLE DESCRIPTION 117

C RDIS BROKER 123

LIST OF TABLES

4.1 Contents of RDIS description and RDIS interpreter. 39

4.2 Calculation of COMR using AToM^3 implementation's language primitives. 41

4.3 Sample inputs and outputs with expected values. 41

4.4 Some common differential-drive robots and driver size in SLOC. 43

6.1 Robotic frameworks and their description format characteristics. 67

LIST OF FIGURES

1.1 An ER diagram of a common landscape in robotics software. 2

1.2 ER diagram illustrating the impact of RDIS. 5

3.1 Simple example of the relationship of a meta-model to a model. 16

3.2 An overview of the MDA layers with concrete example. 17

3.3 A simple graph of the references in RDIS. 19

3.4 Illustration of a maintenance problem. 21

4.1 Example mapping of Create bumper packet to a Range message. 36

4.2 A UML Class Diagram of the RDIS domain (meta-)model. 44

5.1 Flowchart outlining how the solution works. 46

5.2 An overview of the RDIS architecture. 48

5.3 RDIS modeling environment with concrete syntax for an iRobot Create. . 49

5.4 ROS modeling formalism with an example skeleton node. 52

5.5 Transformation rules for the ROS-RDIS model transformation. 53

5.6 Example of the left-to-right nature of the concrete syntax. 57

6.1 Excerpts of Stage and Webots simulation description files. 63

6.2 Excerpts from the SDF and URDF documentations. 65

6.3 An example transmission excerpt from the URDF documentation. 68

6.4 A feature diagram depicting the common features of a simulation description. 69

6.5 Package diagram showing the relationship between the two models. . . . 74

6.6 A Class Diagram showing a possible kinematic meta-model for RDIS. . . 75

CHAPTER 1
INTRODUCTION

Robotics is a growing field with both new platforms and devices being released regularly. It is an active field in academia; both large manufacturers and small home hobbyists work with robots. Thus, robotics has many stakeholders who are not likely to be able to write a control module for a robot or to model the kinematics and dynamics for simulation. When a robotics stakeholder plans for his or her project, the initial steps are to select a target device and target platform suited for the application. However, if the stakeholder is not an expert on either the device or the framework and no mappings already exist between the two, he or she will be unable to continue with the project without first resolving this gap, creating unneeded constraints.

One may argue all that is required is a universal framework for robots, but the reality is that no such framework currently exists. The argument is symmetric for a universal device interface. Therefore, the current generation of robotics stakeholders is faced with this problem: as the number of devices and number of frameworks increase, then the amount of total effort needed to bridge the gap between device and framework grows with the number of device-framework permutations. Often it is the stakeholders themselves who end up writing modules for their desired framework, which means that skilled research time is lost in acquiring expertise in implementation details which otherwise should have been hidden from the application programmer. Otherwise, if this scaling problem is left unchecked, someone is left with the task of repeatedly writing the framework modules for devices.

This problem is illustrated in Figure 1.1, where eventually someone will have to step outside of their domain in order to accomplish a task within their domain. Three domains are recognized. The Device domain is where much of the work in manufacturing the target device is accomplished. There, device domain experts will create device documentation

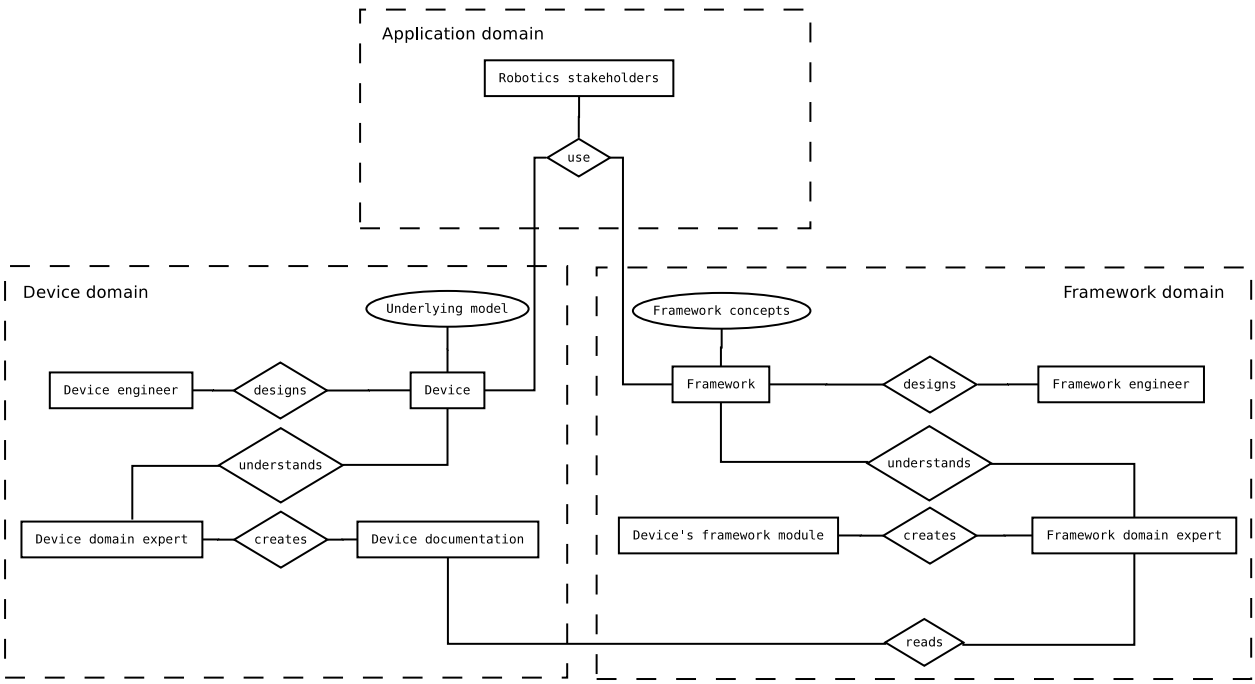

Figure 1.1: An ER diagram of a common landscape in robotics software.

which eventually must be utilized by someone from the framework domain. This is the primary interface between the two domains.

The Framework domain represents the developers and contributing community involved in the development of the robotics software development framework. The framework provides a set of framework-specific interfaces that are exposed to an application. In order to communicate with a target device, there must exist a device framework module that handles marshalling and unmarshaling of data exchanged between the device and framework, as well as managing the connection, threading, and other device-specific parameters.

The Application domain is where the robotics community at large resides. Researchers, engineers, and hobbyists develop applications for robots for their choice of framework and device. Application developers should not need to know the specifics for how a particular device works beyond the interfaces that the framework offers.

The problem with this development process is that a framework domain expert must be the one who creates device framework modules, which can be arbitrarily many. Alternatively, an application developer who wants to use a framework-device pair for which a module does not exist will be forced to develop one themselves. In this situation, the

work will clearly outweigh the manpower.

To scale this problem effectively, it seems the appropriate solution is to add an additional, constant amount of work to both the device side and application side, each of these somehow closing the gap between device and framework by providing abstractions to both the device and the framework. When this is done, an abstract layer is introduced between the device and the framework, as proposed in [4]. A tool that exploits these shared abstract concepts can be created to translate between the two.

1.1 Robot Device Interface Specification

The Robot Device Interface Specification (RDIS) is a formal model of the shared abstract layer that exists between the robotic device and the framework. The model is defined such that a model of the device can be transformed into control code for the target framework. In [4], this was demonstrated via a proof-of-concept Domain-Specific Language (DSL) in which a description for the K-Team Koala robot was translated into control code for HTML5 sockets and the Robot Operating System (ROS) [43].

However, the intention for RDIS is not solely to generate control code for robots. The paper [4] outlines some improvements that have still yet to be implemented for RDIS:

1. A kinematic, visual, and collision description consistent with existing frameworks
2. Error handling at various levels in the architecture
3. Additional threading models (i.e., dual threaded and multiple threaded)
4. Enhancement of local representation of robot state
5. Sensor and actuator error models
6. Enhanced mapping between exposed interfaces and framework-specific concepts

Each of the enumerated items is a necessary component of a complete robot model. However, each of these components is a significant task by itself. Each requires its own specific modifications to the underlying mechanisms which allow RDIS to function.

RDIS may be serialized to a JavaScript Object Notation (JSON) [10] compatible syntax. The JSON encoding was chosen for its light syntax, small data footprint, printability, wealth of existing tools and parsers for manipulating JSON text, and for its wide use on the Internet, for RDIS could foreseeably be retrieved over a network or from the device itself. The JSON data itself is the device description. However, a description by itself is not immediately useful without the tools which can transform it. Originally, the only existing RDIS tool is the compiler itself. It was able to take a description and transform it into an Abstract Syntax Tree (AST), which is how the robot model is currently represented when in computer memory. The AST can then be converted into robot control code. The compiler was implemented using ANTLR [37], a parser generator, and the code generation was implemented using its companion library StringTemplates [38]. The general idea is to write templates for each target framework which are general enough such that code can be generated for many valid model instances.

Since the proof-of-concept incarnation of RDIS, the control model has been more completely defined [3]. The parser has also been rewritten to better fit the JSON syntax. The long term goal of RDIS is to change robotics software development so that a description for a robot may be written once by the robot manufacturer, and RDIS adapters may be created once by the framework developers so that any RDIS-enabled device may be easily taken from framework to framework by a robotics researcher, application developer, or hobbyist so long as the RDIS adapters exist. The short-term goal for RDIS is to create a model for unmodeled aspects of the robot previously outlined, as well as to improve existing model components, for the long-term goal of reducing effort expended by application developers.

However, the described approach for RDIS is quite difficult to maintain. For one, much of the operational code is in templates. Part of this means that the operational portion of RDIS needs to be encoded into templates. This presents the possibility that operational code chunks of RDIS could be duplicated between frameworks, when in reality, much of the RDIS code can be reused. It seems that operational portion of RDIS needs to be implemented as component that exposes an Application Programming Interface (API)

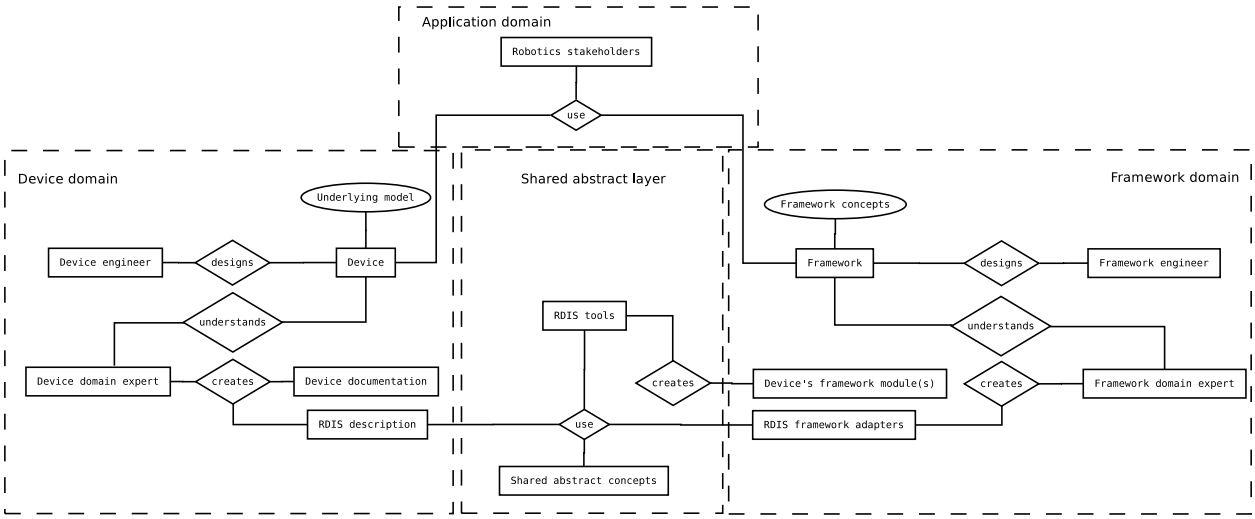

Figure 1.2: ER diagram illustrating the impact of RDIS.

for frameworks to use. This paradigm is more characteristic of an interpreter.

The choice of JSON as an encoding format has led to some difficulties in parsing. Many of the constraints imposed in the RDIS model have to be hand-written in a platform-specific language, meaning one has to implement the constraints in several different languages, depending on where RDIS will be translated. The philosophy that RDIS descriptions are an unverified artifact leads to some difficulties of having to validate the description each time it is loaded.

Figure 1.2 illustrates the long-term goal of how it is hoped that RDIS will change robotics software development. The burden of creating the device adapter for the framework has moved to the set of RDIS tools. The device domain expert only needs to create one description per device and maintain that description. The framework developer will only need to create RDIS adapters once and similarly maintain those.

1.2 Contributions

The overall focus of this thesis is to identify the work and improvements made to the initial RDIS proof-of-concept. The following sections introduce each of the contributions and offer a brief summary.

1.2.1 Definition of the RDIS Model

Initially, RDIS was implemented as a domain-specific language with its operational semantics contained in the templates used for code generation by the parser. A better goal is to move to an interpreted solution. However, there was the problem that it was difficult to extract the semantics from the existing code base. Therefore, as a part of the contributions of this thesis, RDIS has been defined formally.

An interpreter for RDIS is also submitted for part of this thesis. The interpreter implements the parts of RDIS previously described. It serves as an empirical means of evaluating the behavior of RDIS so that it may be taken to other platforms. As a side effect, the syntax was morphed to better fit the interpreter, and some constraints for the RDIS textual syntax imposed, in particular the syntax for mapping messages and executable code chunks.

As well, certain parts of the model were modified to better expose an interface to the framework. The changes do not add functionality, but provide a more logical mental model for Domain Concepts which has often been difficult to describe in the past.

1.2.2 A Model-driven Approach for Describing Robots

Previously, it was thought that the primary artifact for RDIS was a textual description of a robot, a domain-specific language. The DSL followed a JSON-compatible syntax. JSON was adopted since it is standardized and more attractive for the community to adopt. In doing so, RDIS failed to meet one of the goals of a DSL. A DSL is meant to raise the level of abstraction so that the programmer can work in terms of the problem domain. However, it seemed that more time was spent conforming to JSON syntax. JSON was still an attractive machine language to adopt, but it was very difficult for humans to work directly.

Therefore, this thesis shifts the philosophy of RDIS that the primary artifact is no longer the textual syntax, but the model itself, making RDIS a model-driven solution. To facilitate this, RDIS was implemented as a formalism for the modeling tool AToM^3 and an example problem is solved for a particular device and framework. The previous DSL was modified and is now generated from the model.

1.2.3 A Survey and Analysis of Kinematics Description

A future step for RDIS is the inclusion of a kinematic model with the existing control model. This allows for interesting problems to be explored such as maintaining the robot's physical state, simulation, and mapping the messages of the control model to the kinematic model.

Initial work has been done in this regard. A survey of literature in robot kinematic modeling was conducted as well as the analysis of the existing simulation descriptions for some popular robot simulation suites.

1.3 Limitations and Assumptions

There are several limitations and assumptions which are made to make RDIS a worthy pursuit. They are itemized:

- *The modeled robot is primarily controlled via message passing over a transport via an off-board application.* This research targets a class of robots which are common, but it is certainly not the case that all robots are controlled in this matter. Some robot chipsets may accept a robot program to hold in local storage for fully autonomous behavior. While the physical construction of this robot may be modeled, the benefit is lost because one cannot define a mapping from the interfaces to the kinematic model because the concept of an interface does not exist on this type of robot.

- *It will be less effort to model the robot in RDIS than directly for a target framework.* Currently, the textual syntax of RDIS is encumbered by the JSON conformance it adopts, so crafting it by hand may be no advantage. However, the proper meta-modeling of the domain can enable a domain-specific visual modelling language to be created to generate the RDIS file.

- *The presented analysis metrics are not standard.* Metrics at the model level is an area of active research. It should be noted that there currently is no "standard" way to assign a straightforward quality metric to the model. The authors presenting the metrics in [66] do warn that more work is needed to make any conclusions re-

garding the validity of the metrics they present. Any model-level metrics should be presented purely for discussion, and should not be used to directly draw conclusions about the outcome of the research. Success metrics should be based on empirical evidence, such as productivity measurements through user study or metrics based on code analysis.

1.4 Conclusion

The robotics community at large often use software frameworks to aid in the development of robot applications. This helps application developers as it provides them with domain-specific concepts to express their application and saves the developer time by providing a middleware. However, writing drivers for frameworks is not an easy task and application developers may be faced with this burden when working with new or experimental devices. As well, drivers for the same devices between different frameworks share some patterns of implementation. Therefore, it would seem that with the appropriate abstraction, software reuse could be facilitated.

The RDIS domain model has been developed to serve as this abstraction. It is designed specifically for modeling robots attached to a host computer. Therefore it uses concepts which are familiar to driver programmers who have worked in this context. The initial implementation was entirely text-based. This implementation enables a developer to work at the level of the domain model in a graphical environment.

The bulk of the contribution is mostly drawn from the domain model. The domain model has been defined, implemented, and documented. As a means to verify the domain model, it was implemented in a modeling tool, AToM^3 [25], to enable code generation of RDIS descriptions and as an example approach to generate adapters for frameworks. The model in its current state may be taken to other tools, and interpreters may be written for other languages.

Future directions for RDIS include creating a generalizable kinematic model for RDIS. This will allow for RDIS models to define simulation descriptions in a generalizable way to solve a similar problem seen in simulators. Some preliminary work including a survey

of the literature, analysis of some robot simulators, and design of a possible model has been accomplished in this effort. Future research may concentrate on the implementation of this model and the integration with the other parts of RDIS.

This research comprises one of very few projects to explore the use of abstractions specifically in the domain of robot middleware. At the most basic level, it can be used to describe how drivers are written for robot frameworks and defines a language to use in this context. More desirably, the model can be used to create platform-inspecific descriptions for robot syntax and semantics. The end goal is to have any RDIS-enabled framework to be able to communicate with any RDIS-enabled device. RDIS could be applied in its current implementation or serve as a design framework for creating standard interfaces for robotic devices, much like an operating system does for day-to-day hardware.

CHAPTER 2

MODELING AND ROBOTICS: A LITERATURE REVIEW

This chapter presents a literature review in the use of modeling techniques in the field of robotics, and characterizes the state of the art. The first section presents work which focuses on robot application and control modeling, which ranges from projects which are similar to RDIS, to DSLs for robot control. The second section presents projects which focus on robot kinematic modeling, which can include pure modeling approaches and DSLs for kinematics description.

2.1 Application and Control Modeling

This section presents a survey on approaches to controller modeling and general use of abstraction in the field of robotics. It can include anything from projects with similar goals to RDIS to projects which have invented their own DSL for writing robot applications.

2.1.1 Pure Modeling and Robotics

Pure modeling is not a standard term, but is meant to signify approaches which follow explicitly or approximately the approaches delineated in the Model-Driven Architecture (MDA) specification [30]. These approaches may use the UML family of languages as the primary tools of implementation, but more importantly they follow a clear meta-meta-model to model instance type approach.

The Object Management Group (OMG) has taken an interest in standardizing the domain of robotics in the establishment of the Robotics Domain Task Force (RDTF) [36]. They have a small set of Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs) [33–35] for robotics defined in terms of the Unified Modeling Language (UML). None, however, focus on the modeling of the kinematics and dynamics of robots.

The researchers in [45, 51] used the Eclipse Modeling Framework (EMF) to deploy application software to different device-framework pairs. One [45] used models for a P3DX robot and a QFIX-based robot to generate code for several different platforms, including Player, Microsoft Robotics Studio, and SmartSoft [46]. Statecharts [15] were used to define robotic behavior. The other [51] proposed SmartMARS, a meta-model derived from SmartSoft [46] to use for modeling and analysis of robots. A separate researcher [58] used a similar approach to build a subsumption-based robot architecture which was deployed to a NXT Mindstorms robot. These approaches represent an alternative to working directly in a domain-specific language, instead relying on well-understood standards like UML. There are some drawbacks to this approach as at least some textual representation is still needed for the model to be serialized and passed over the network. As well, textual representations tend to simplify visual models as they do not need to include user interface aspects of visual models such as the position of a model element on the computer screen. However, structured visual formalisms may be compiled into a domain-specific language which could turn out to be a preferred way to model particular robots.

2.1.2 Domain-Specific Languages for Robot Applications

URBI [5] is a DSL for robot applications. It is a runtime environment which controls robots. Its language, urbiscript, has native syntax for parallel and event-driven constructs. This gives it high expressivity for solving problems common in robots. As an example, urbiscript can easily natively express "when my left arm stops moving, start moving the right arm." However, the limitation in URBI is that it is another platform. It expects that a programmed description of robot syntax and semantics exists. Also, URBI focuses only on robotic control, however, it can tie into simulators to be used as a control module for the simulation.

The authors in [47] present a DSL for the control of a novel robot. The ATRON self-configurable robot is composed of several independent modules which can be assembled in different manners to create a robot whose function changes according to how the modules are connected. They use the example of a car configuration in their paper. Their language

is role-based, meaning a module can deduce its role by examining its own connectivity. The advantage to this type of description is that it allows the reuse of one behavior description for several configurations of ATRON modules. For example, a description file that represents a car may be used for four-wheel or six-wheel configurations. This is an interesting abstraction for application code, however it has the same problem as URBI, in that a platform-specific module must be created so that a robot may communicate with the framework.

XRobots [57] is a DSL based on hierarchical state machines, which are in turn based on Statecharts [15]. The language is highly behavior-oriented, and behaviors have native syntax and are first-class objects in the language. This leads to a very hierarchical way to build complex behaviors by building chains of calls to behaviors and facilitates code reuse by allowing behaviors to be parametrized.

2.2 Kinematic Modeling

As kinematic modeling is a future interest of RDIS, modeling approaches for robot kinematics are of interest. This section presents the efforts of current research to use abstractions to model robot kinematics. It can include DSLs for simulation description or approaches more akin to pure modeling.

2.2.1 Pure Modeling for Simulation Description

ModelRob [44] is a Simulink library for model-based design of robotic manipulators. Its particular merits are the hierarchical-style design which Simulink offers and a built-in C code generator. One can create a model for a manipulator and controller and view the output in a simulator. It makes use of Simulink's building-block language which enables automatic code generation.

2.2.2 Domain-Specific Languages for Simulation Description

In [26], the researchers created a DSL for the purposes of generating simulation code for the Newton-Euler inverse dynamics algorithm. Their model assumes the familiar link-joint model when dealing with manipulator kinematics [9]. This work is particularly

interesting because the syntax of their DSL tends to match up in several ways with existing simulation descriptions. Also of particular interest to this proposal is the metrics the researchers chose to assess their model. First the researchers tested accuracy by comparing their code's results against the output of a Matlab implementation of the algorithm. They chose to address performance by testing their simulation against the SL Simulator by varying the degrees of freedom in the model and observing how it affects time. However, the results in this paper are for legged robots. Some special consideration may need to be given to wheeled mobile robots whose operation is unlike manipulator type robots.

A second DSL stemming from the description of self-reconfigurable robots is described in [6]. The authors describe the Modular Mechatronics Modelling Language, or M3L. The language once again focuses on the ATRON robot modules, but it is general enough to describe other types of reconfigurable robots. The language is a kinematic description only, and works by declaratively specifying the available points of connections. Connections are then described indirectly by specifying a set of conditions which must be met to merit a connection. Links are a grouping of points which together define a mesh which composes the link. Axes can also be defined via an origin and a vector. Joints can be defined by specifying the joint type and the link and axis involved in the joint. It is important to note that a connection in this case represents a channel by which messages can pass between the modules. The authors are able to use their DSL to translate into a configuration for the Webots simulation framework [27].

2.3 Model Metrics

In the future, model metrics may be a useful tool for evaluation of this research. Therefore, a short survey of model metrics which may be helpful for the RDIS project is presented in this section.

Mohagheghi presents a survey of the state of model metrics as a part of an ongoing research effort in model quality metrics [28]. The paper presents six general goals for measurement at the model level which should serve as the inspiration for the metrics

used to evaluate a model. The ideas of Lange on model size [24] are also presented, in particular dealing with UML models. Lange proposes a specific metric for UML relative size, which could be used to measure the size of the proposed robot model against existing description models.

In [66], the authors present their perception of model metrics for Domain-Specific Modeling Languages (DSMLs), offer a classification of the types of effort they believe are encountered when working within a DSML, the two basic types being Development Effort and Runtime Effort. Within the context of this project, the metrics which measure Development Effort are most interesting, particularly those which measure Modeling Effort, the effort involved in developing a DSML model, and Cognitive Effort, the cognitive effort required to do so. They use a weighted sum to approximate the size of a model as a means to measure modeling effort. This metric should be fairly sufficient for calculating an inexact model size, and it should translate well between different models. Another interesting metric they use is the Closeness Of Mapping Ratio (COMR) which they used the ratio of the number of problem-level language primitives to the number of solution-level primitives which do not have a mapping to the problem domain. This metric is interesting because it provides insight to the disconnect between the problem domain and the solution domain. However, to actually implement this metric would require a classification of any existing language to be benchmarked and could be expensive time-wise. The authors also mentioned other metrics which are useful when evaluating imperative languages. However this project will focus on building a declarative language so they will not be so directly applicable. The author in [50] considers a new approach to estimating model complexity via the analysis of the meta-model which defines it. This could solve the problem of evaluating complexity in a purely declarative model like RDIS.

CHAPTER 3

TOWARD A MODEL-DRIVEN APPROACH

This chapter presents an argument to implement RDIS as a visual modeling language as opposed to a text-based DSL. It presents a review quick review of model-driven software design and domain-specific engineering, their merits and drawbacks, a comparison of the challenges when the primary artifact is a model or text, and a short selection of modeling tools. This chapter reviews the core concepts of modeling necessary for understanding the benefit of the contributions in this thesis. A more complete review on the matter may be found in related textbooks [19, 61].

3.1 Model-driven Software Development and Domain-Specific Modeling

Model-driven software development is a software development methodology where instead of source code, software models are the primary artifact. Software models are described by a formal modeling language such as UML [31]. The purpose of models is a matter of modeling theory, but Selic's characteristics of useful models [48] are a good guideline:

- **Abstraction** – the model is a reduction of the system under study.
- **Understandability** – the model is expressed in a form which appeals to intuition.
- **Accuracy** – the model represents the system under study sufficiently.
- **Predictiveness** – the model may be used to reveal interesting, non-obvious properties of the system under study.
- **Inexpensive** – the model is cheaper to construct than the system under study.

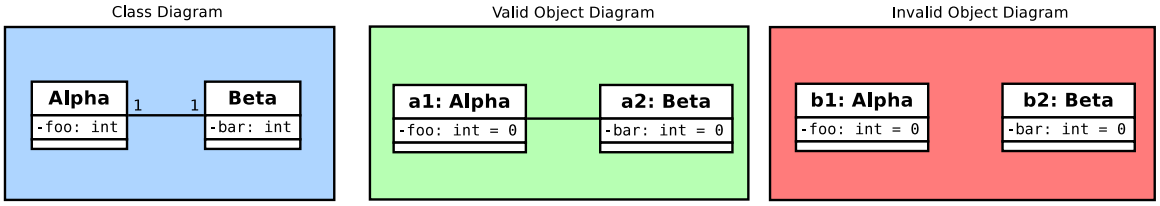

Figure 3.1: Simple example of the relationship of a meta-model to a model.

In [29] the interested reader can find an overview of the different modeling philosophies and definitions.

With these characteristics, one can recognize that the essential goal of software modeling is to provide an abstracted view of the desired implementation without the effort of creating a full implementation. If such a model can be expressed in software, then the added benefit is the model can be translated into a partial or full implementation.

As previously stated, models must be defined in terms of a modeling language, but as is the case with textual languages, general-purpose modeling languages can be cumbersome and difficult to use [18]. Such modeling languages can be extended to be domain-specific. This provides the domain modeler with access to language primitives which are based on problem-level abstractions. The aim of domain-specific engineering is to exploit existing understanding of a system and use these abstractions to accelerate the development process.

In order to understand how to make a modeling language domain-specific, one must first understand the hierarchy of modeling languages and the concept of meta-modeling.

3.1.1 Meta-modeling

Meta-modeling is a core concept to understand in order to grasp how RDIS is implemented. As the name might suggest, a meta-model is a model which describes a model. As normally this relationship is often hard to grasp, it merits further discussion. Meta-models are defined in terms of an existing modeling language. As UML class diagrams are well-understood, we shall use them for discussion. In a class diagram, one might define a class with some attributes. A class may be associated to another class by way of an association. The association can impose certain multiplicities between the two classes.

The relation of meta-model to instance model is not unlike the relation between a

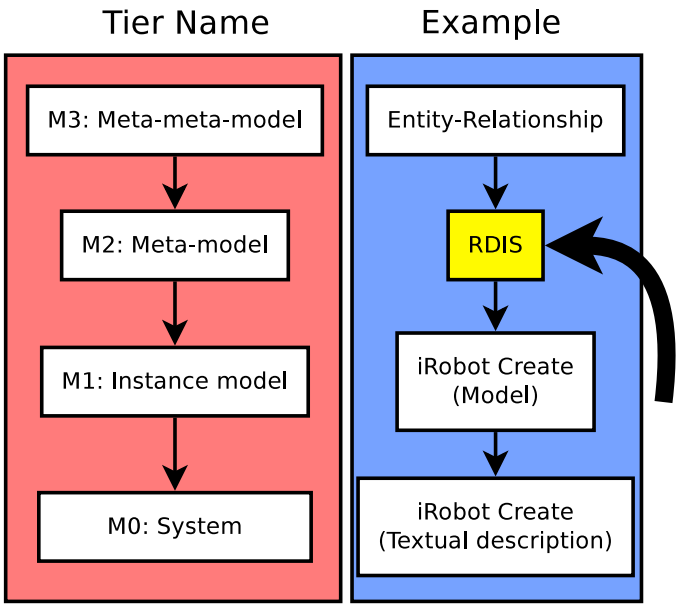

Figure 3.2: An overview of the MDA layers with concrete example.

class diagram and an object diagram in UML. A class diagram may define valid instances of objects in the way that meta-models define valid instances of models. For example, in Figure 3.1, objects 'a1' and 'a2' are in a valid state defined by the class diagram. However, 'b1' and 'b2' are not because there is not an association between them. By specifying one valid meta-model, one has defined the set of all systems it can classify. Interested readers are encouraged to read Kühne's paper [23] for a more theoretical definition of meta-models.

OMG's MDA specification [30] suggests a four-layer style approach to software modeling. The standard view is that meta-models are described in a choice meta-meta-model formalism, and meta-models are used to define valid model instances. These model instances can then be transformed in some form or another into an implementation, or intermediate implementation models. This thesis proposes that RDIS exists at the meta-model level, as shown in Fig. 3.2. Further, it suggests that the RDIS textual description should be considered an artifact generated by the model, not edited by the modeler directly.

The attractive feature of meta-modeling is that it allows one to get some useful artifacts as a side-effect of defining the meta-model. For example, in AToM^3 one can define a meta-model and provide a graphical concrete syntax and some constraints; then, a

model editor may be generated [12]. The editor can assist the modeler in designing valid model instances and alert the modeler of errors in the model. Additionally, model transformations can be defined at the level of the meta-model which can help in a variety of situations. For example, consider a model of a robot which uses features that are deprecated. If there is a mapping between the deprecated features and an accepted version of the same feature, a model transformation could be designed to migrate all deprecated models to the new version.

3.1.2 Code Generation

Code generation is another important benefit of modeling. Similarly to how a program's abstract syntax may be translated into another form, a model may be transformed into code. This is the most important end result for this project, as the desired result is to avoid writing code altogether.

When one creates a PIM for a system, it can be used to generate code for a variety of different platforms. The PIM must only contain the information which is universal to all platforms in an abstracted form.

3.2 Models Versus Text

One may wonder when using a visual model is more appropriate than using a text-based DSL. This section presents some of the advantages and drawbacks of using a graphical model over a textual model.

3.2.1 Graph-like Versus Tree-like

Models are not in a serialized form, so they can express and verify circular relationships more easily. Text languages must be serialized, so references are then handled as identifiers. References then must be verified by a validator. Relationship multiplicities can easily be specified in a meta-model as opposed to the alternative in a language validator. To further the point, the classification framework proposed by Costagliola [8], it can be inferred that many modeling languages have notions of entities and relationships; they simply choose different classes of syntax to represent it. If we consider references as

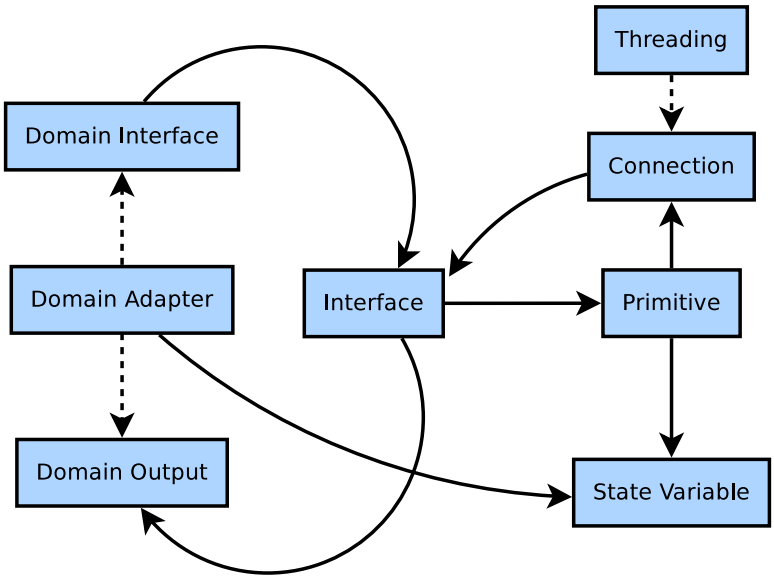

Figure 3.3: A simple graph of the references in RDIS.

relationships, it becomes easy to see that the RDIS domain model is simply a collection of model entities referencing one another.

Previous efforts considered the AST generated by ANTLR as the software model for RDIS. This shows that RDIS is suited for description by what Costagliola calls a connection-based class of visual syntax, or a graph. A tree is simply a connected graph without cycles. However, Fig. 3.3 shows a simple graph of RDIS where solid arrows represent references and dotted arrows represent containment. One can quickly see that the relationships between the model elements are complex and would be difficult to mentally reconstruct from a serialized form. As well, the cycles present in the graph can be more easily expressed in a graphical environment.

This thesis suggests that RDIS is not inherently tree-like, but graph-like based on its referencing scheme. Therefore, it will pose an added mental load to the modeler to work with RDIS in its serialized form. Instead, RDIS might benefit from a graphical syntax and an editor for the modeler to work in and icons representative of the domain.

3.2.2 Editing

Editing models can also provide benefit to the modeler. Models with a graphical syntax are typically edited in a Graphical User Interface (GUI), whereas a textual language might be edited in an enriched text editor. The benefits of editing in a graphical environ-

ment is that the modeler has immediate feedback as a visual representation of the model in a notation that appeals to intuition. Relationships and references can be expressed as arrows instead of identifiers.

3.2.3 Constraints

Another important aspect in any language is constraints. Most modeling languages have quick ways to specify simple constraints such as multiplicities. More sophisticated constraints can be expressed in a constraint language such as the Object Constraint Language (OCL) [32].

In fact, the majority of the constraints present in the RDIS model are simple ones such as multiplicities and reference verification. By using a modeling language to define RDIS, we gain the benefit of being able to quickly specify multiplicity constraints, whereas in the previous implementation this would need to be validated in a general-purpose programming language. Modeling RDIS presents a much easier way to satisfy multiplicity constraints.

RDIS also enforces unique names for elements of the same type. Some meta-modelling environments provide a built-in means of verifying that an attribute is "key" among entities of the same type. More domain-specific constraints have to be expressed in a constraint language. For example, a Serial Connection may not have its baud set to a negative value. While this must be solved in a similar manner as before, since the validation now occurs at compile time, then the constraints would only need to be specified once rather than for each generated parser.

3.2.4 Maintenance

A meta-model can be maintained easier than a parser. Consider the particular case in which a multiplicity must be changed. The necessary changes would first need to be implemented in the meta-model. The meta-model would be compiled, and the new formalism would be ready to use.

The difficulty in maintenance is mostly found in the usability of the modeling tools. As in our example case, consider a case where a meta-model allows three connections of *ElementA* to *ElementB*. Suppose this multiplicity drops to two. In the case of the

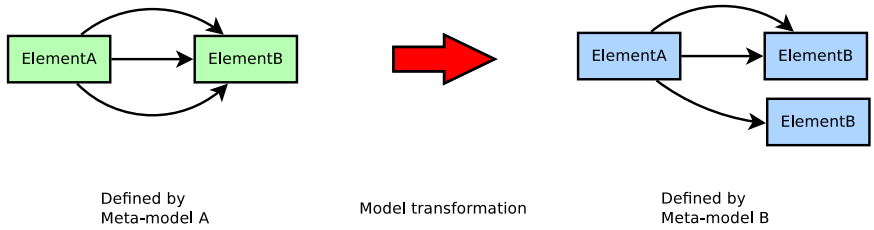

Figure 3.4: Illustration of a maintenance problem.

modeling tool AToM^3, one of the edges would simply be deleted the next time the model is opened. However, if the semantics of the migration was to create a new *ElementB* and create an edge to it, then one would need to maintain the two versions of the meta-model and design a model transformation which can perform the migration (Fig. 3.4). This is very possible in theory, but can be difficult in practice.

3.2.5 Drawbacks

The obvious drawbacks of an early adoption of model-driven practices is the maturity of the tools. Since model-driven tools are new, there are some difficulties regarding the usability and stability. As the discipline progresses, so will the tools.

Another drawback is that models are not readily usable by other tools – or the model is only useful for the tool that it is written in. Graphical models typically contain data relevant for their graphical presentation that is not relevant to the application for which they are developed. This means models must be transformed to a separate artifact the application can use, which is an extra step in the development process. Typically the gains are enough to justify this.

Finally, as pertains to robotics, is the possibility of the difficulty in community adoption. It is unlikely to find a roboticist who is educated in model-driven software development as it is such a new field. Therefore, the adoption of model-driven techniques as a community will be very difficult.

3.3 Tools for Model-driven Development

This section presents a look at a few different tools for model-driven development including a brief description, some of their features, and some applications which have

used the particular tool in the domain of robotics.

3.3.1 AToM^3

A Tool for Multi-formalism Meta-Modeling (AToM^3) is a modeling tool available from McGill University [11]. AToM^3 allows one to define meta-models using something akin to the Entity-Relationship (ER) formalism and provide a graphical concrete syntax via a built-in graphics editor. AToM^3 is able to generate a modeling environment for itself with just this amount of information.

Code generation is not wholly automatic for AToM^3 as the meta-modeller must provide the code generation scheme for custom formalisms. It has support for incremental constraints written in Python. Compile-time constraints can also be implemented by the meta-modeler externally using AToM^3's API.

AToM^3 also has support for rule-based model transformations. Rules can be defined via a graph grammar. Most notably, it can perform multi-formalism model transformations.

3.3.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is an Eclipse extension that allows for modeling and code generation within the Eclipse Integrated Development Environment (IDE). It is available from the Eclipse Foundation [13]. EMF is a very robust modeling solution gaining wider adoption and development activity, and is a popular application to teach in the university setting.

EMF contains all the standard fare for a modeling tool, such as a class diagram designer which can generate Java interfaces. However, it is a bit different as the developers the importance of the relationship between the model and the generated code. According to the EMF book [52], EMF considers modeling and programming the same thing. EMF allows one to design in Java, UML, or XML. Model transformations may defined in EMF via the textual transformation languages ATL and QVT. EMF has been used to create robot applications [45] and for analysis [51], as well for developing a robot behavior architecture [58].

3.3.3 MATLAB/Simulink

Simulink is a proprietary modeling and simulation tool tightly related to The MathWorks's MATLAB tool. Simulink has its own block-based language for defining models. Simulink does not offer the same sort of flexibility as AToM^3 or EMF in that it does not support meta-modelling. It is best categorized as a DSML for dynamic systems.

Though it does not have the same flexibility as EMF and AToM^3, Simulink does have its merits. It is a tool that has been adopted by the automotive industry [54] and has had some success in modeling the kinematics of robots [44]. It enables full code generation in C. The challenge in using Simulink is knowing whether or not it is general enough to generate code for the particular application at hand. For instance, Simulink is good for instrument control, so serial port and TCP/IP can be modeled in an application, but its ability to bind with any arbitrary robot framework is questionable.

CHAPTER 4
THE RDIS DOMAIN MODEL

The purpose of this chapter is to explain the RDIS domain model. First, we will identify previous work. Each of the major components will be explained in full, as well as each attribute of the component and its purpose. There will be some remarks as to major changes within the model, followed by an analysis of the model according to modern thoughts on domain modeling.

4.1 Introduction

Parts of the RDIS domain model had already been defined. This section will first define which parts of the model were already defined. The initial work regarding RDIS is shown in [4]. Much of the work discussed in this paper existed already. The following sections attempt to enumerate this prior work.

4.1.1 The High-level Domain Model

The high-level domain model is present in [4]. The main components were identified, such as Primitives, Interfaces, Connections, Threading, and Domain Concepts. However, the exact attributes of each of these components has changed dramatically between each iteration of RDIS. The relations between the entities are discussed in the paper and by following early renditions of the textual syntax one can see that most of the relationships were known.

RDIS had never been modeled in a formal modeling language, nor had it followed any formal design process. The sole existing de facto definition of RDIS was the grammar which defined it. Figures in another early paper [3] did reveal the overall architecture of RDIS but this is still a high-level, informal specification of RDIS.

4.1.2 The Approach for Implementation

Normally "RDIS" as a term referred to the textual syntax as there was not a formal model to support it. The textual syntax was the model. Therefore, the implementation approach was to use a parser generator and use translation templates to perform the code generation. The generated code was specific to a particular framework, and could then be compiled for that framework.

4.1.3 The Framework-RDIS Interface

One of the model components was referred to as Domain Concepts. These were intended to represent the standard messages that would need to be exchanged with the device. Domain Concepts worked similar to interfaces in that they had a name, some parameters, and small code chunk which would perform the transformation. These Domain Concepts served as the interface between RDIS and the robot framework.

4.2 Revisions to RDIS

As part of the contributions of this thesis, the RDIS domain model has been refined and implemented both in an interpreter and a modeling tool. The domain model itself has been substantially modified from its initial state. Hence, the changes affect both the model and the approach of the implementation. The following sections describe each.

4.2.1 The Model

While many of the relationships are the same as in the original model, the attributes and semantics of the components themselves have changed substantially. The components which have seen the most significant change are Primitives, Domain Concepts, and mini-language.

Primitives retain their original intent in that they have a name and several named arguments. However, Primitives and Message Syntax used to be thought of as separate model components. The current model does not model Message Syntax explicitly, but instead Message Syntax has been embedded as attributes of Primitive as the relationship between the two is one-to-one. As well, the formats for Message Syntax used to be

enumerated specifically. As an example, the Koala expects that drive commands are issued as the ASCII-encoded line "D,20,20", which tells the robot to drive forward at some arbitrary speed '20'. The old model would have modeled this explicitly as a delimited line with a comma as the delimiter.

There is a new approach for defining messages. Since messages are inherently string-like, it seems to make more sense to define them using well-known parsing/generating languages. This approach is taken because it is difficult to define byte sequences at a high level of abstraction and one has to take care to consider all the cases for message formats, such as fixed-width or delimited. In the cases of encoding and interpreting bytes, endian-ness also has to be considered. Taking all this into account can be quite verbose in a modeling language, whereas some text-based languages can represent this succinctly.

For primitives, we consider the cases where bytes are raw-encoded or ASCII-encoded as these two cases seem to be the most popular. For encoding ASCII data, we use a format string. The specific format string used is not important, but Python format strings as defined by PEP 3101 [17] are currently assumed by the interpreter. For reading ASCII-encoded data, we use a regular expression. The adoption of possibly platform-specific languages in part due to the fact that eventually the model must be serialized to a text form before it can be used, so it seems the best strategy is to adopt a specification which can be implemented on many platforms, rather than attempt to define a new specification which must be translated to the platform-specific solution. That being said, the programming community at large is familiar with the concepts behind format strings and regular expressions so it seemed to be an agreeable choice.

For raw-encoded bytes, one must deal with cases of endian-ness, bit width, and signed data. Since we are once again constricted to a textual description, it would be optimal to choose a similar approach as chosen for ASCII data. However, there are not many well-known formatting languages for packed binary data. Once again, it seems best to adopt something that is not necessarily constrained to a particular platform that solves the problem. We turn again to a Python standard for format strings for modeling packed binary data [42], the alternative being inventing such a format specifically for RDIS.

The drawbacks for both of these adoptions is that they are memoryless. For example, a regular expression could not handle a message type that specifies an arbitrary number of bytes which are to follow. Similarly, the packed binary data format string could not handle this as well. A more complex language would need to be developed should the need for it arise.

Further, the idea of Domain Concepts has been previously met with some confusion. Previously, Domain Concepts were modeled like keywords in the DSL. For example, there was a "controlDifferentialDrive" keyword which expected a value "transvel" and "rotvel" to be in scope to use in a transformation.

This iteration splits the framework-RDIS interface into roles of three different components. The first of which is called a **Domain Interface**, whose role is to accept input messages from the framework (a more fitting name may be *Domain Input*). Similarly, there is a **Domain Output**, whose role is to send output messages to the framework. Domain Interfaces and Outputs are named, though at some point it may be beneficial to limit the sets of Domain Inputs and Outputs in a similar manner to Domain Adapters.

Domain Adapters are specific data types which may flow in and out of the model. Domain Adapters are attached either to a Domain Interface or a Domain Output. In the case of a Domain Interface, the member names of a Domain Adapter serve as the named parameters (similar to a function call). In the case of a Domain Output, the Domain Adapter's attributes are defined as a function of the State Variables to provide messages back to the framework.

The mini-language adapted for arithmetic expressions that must be expressed in text is simply Python. The original implementation used a Python-like syntax that was custom-parsed by the RDIS parser. However, as Python environments exist for many target languages it is a viable way to continue as the interpreter can hide the complexity of embedded scripting languages.

4.2.2 The Approach

Previous iterations of RDIS considered the textual syntax to be the model. Thus, there was a grammar which defined RDIS and the language was validated by the parser.

However, there were many flaws with this approach. For example, the JSON specification requires that member names be quoted, however this renders member names to look identical to strings by the lexer. Given that JSON subsets are not normally defined, the unstructured characteristic of JSON is just a part of the nature of the language. In the end, it resulted in a parser which was very difficult to extend and maintain and a work flow that was unsustainable. It seems that developing RDIS restrictions in an unstructured environment (e.g., a text editor) and relying on the parser to validate is a poor approach. Instead, RDIS would do better to have a tool-assisted development work flow.

This thesis urges the adoption of model-driven tools to build and validate RDIS descriptions. This approach views RDIS textual descriptions as a compiled artifact of the model and are not to be edited manually. The benefit of this adoption would be a higher-level way to describe robots than the DSL originally offered, and would allow one to assume that RDIS textual descriptions are valid when loaded, and thus the standard tools for parsing JSON can be used.

4.3 State Variables

State Variables represent some aspect of robot state or named constant (though the semantics of a constant are not currently enforced). A specific example might be the last known value of a sensor. State Variables exist so Interfaces which do not call any Primitives may fetch the value of the last known value of the sensor as opposed to actually reading the sensor on the robot. State Variables may not be the most efficient way to handle robot state. Future iterations of RDIS might elect to replace State Variables with more abstract concepts. For example, robot pose could be maintained in the kinematic model or sensors could be explicitly modeled, cache their own values, and assign a time-to-live for the cached value.

4.3.1 Type System

State Variables are strongly typed. An interpreter should attempt to cast values to the corresponding types when setting State Variables, or raise an error on a failed attempt

to do so. RDIS identifies the types `int`, `float`, and `string`. Bijective mappings between RDIS types and host language types must be defined to detect errors in type assignment.

4.3.2 Scope

State Variables are **world-readable**, but are only writable by Primitives. This allows them to be used as parameters to Primitive calls (by Interfaces) or Domain Adapters may return values as a function of State Variables.

4.3.3 Initialization and Default Values

Initial values for State Variables are subject to evaluation by the mini-language. The empty string is understood to be the default value for that type of State Variable. For `int`, this is `0`, `float` `0.0f`, and for `string` this is the empty string.

4.3.4 Future Deprecation

As State Variables do not add to the abstraction of the model, future iterations of the RDIS domain model should seek to remove the necessity of their existence. There should be opportunity for robot state to be embedded into other elements of the model. To move toward this, one should consider the purpose of the usage of State Variables.

One usage of a State Variable is to store the last known value read from a sensor. A possible way to replace this usage is to add a Sensor model element which is a subclass of Interface. This model element could then cache values read from Primitives for some amount of time. This would result in less messages being passed to the robot and a higher level of abstraction, and removes the need for world-visibility and a use of Primitive post actions.

Another usage of a State Variable might be to track the position of a robot. This usage could be replaced by the addition of a kinematic model which aims to keep up with such aspects of state.

One final use of State Variables is as constants. As this usage is general, there is no general means to remove the need for it without considering the nature of the constant. For example, it may be a constant related to the geometry of the robot (such as the spacing between differential-drive wheels). This kind of constant may be taken care of in

the kinematic model.

4.4 Connection

Connections are abstract model elements which describe the transport which lies between the host application and the device. They cannot be instantiated in the formalism.

4.4.1 Startup, Keepalive, and Terminate Interfaces

All connection subtypes have startup, keepalive, and terminate interfaces. They are references to an Interface which are executed on connection startup, main thread tick, or on connection shutdown, respectively. Calls to these interfaces may have arguments for which State Variables are the only names in scope.

4.5 Serial Connections

A Serial Connection represents a connection over the serial port. This can include a variety of connections outside of just a standard 9-pin cable and can include anything that behaves like a serial port as far as the host operating system is concerned. This includes most commonly serial-over-Bluetooth, but any connection which has a serial interface on the host system is appropriate.

4.5.1 Baud

Baud may be set for a serial connection. Baud must be a positive integer. It may be omitted, but the default value of 9600 will be assumed.

4.5.2 About Port

Defining the port was formerly available in previous iterations of RDIS but this implementation has removed it. Instead, it is left to the framework or application developer to assign a port. This was due to the concerns regarding the platform-specific or even setup-specific nature of port naming schemes (e.g., `COM1` vs. `/dev/ttyS1` vs. `/dev/rfcomm1`).

4.5.3 Unmodeled Attributes

Serial connections generally have other parameters. Unmodeled attributes have undefined behaviors in implementation. The current implementation assumes a byte size of

eight bits, no parity, and one stop bit.

4.6 Threading

Threading is a corollary element of Connection. A Threading element has a one-to-one relation with a Connection. Currently, not much attention has been paid to threading, and only single threading has been modeled.

4.7 Single Threading

Single Threading assumes that all work happens on the main thread of execution. This means that all connections on the device are started in a serial fashion, and their individual ticks are serialized.

As the threading model develops with further research, the semantics of Single Threading may need to change. For example, it may be more advantageous to issue a single thread for each single-threaded connection in the model. The investigation of the semantics of threading is left to future work.

4.7.1 Frequency

Frequency, defined in milliseconds, defines the amount of time the thread should sleep between iterations.

4.8 Other Threading Models

Two other threading models have been proposed in the past for RDIS, but not implemented. They are Dual Threading and Multiple Threading. They are left undefined.

4.9 Primitives

A primitive is an invariant, atomic unit of functionality offered by the device firmware. They may be thought of as function calls with highly tight constraints on their parameters. However, Primitives also must deal with marshalling and unmarshalling of parameters to account for issues of encoding, endian-ness, and format. Primitive is one of the more

complex model elements making it the largest of the elements.

4.9.1 Parameters

Primitives take a set of positional, named parameters. Generally it is recommended they correspond to the manufacturer's documentation. The names are in scope in the mini-language during evaluation.

As explained previously, Primitives have absorbed the "Message Syntax" model element before, and messages have been divided into two categories:

- Fixed width, raw-encoded byte sequences, modeled as attributes `pack` and `unpack` taking the format of Python `struct` format strings [42].
- ASCII-encoded data, modeled in attributes `format`, a PEP 3101 format string [17], and `regex`, a Perl-Compatible Regular Expression (PCRE) [16]

The particular standards used for the textual representation should be a best effort to ensure the universality of each implementation. A PCRE implementation should be available on most systems, but a move away from PEP 3101 may be justified in the future, the reason being PEP 3101 implementation is less likely to exist in other languages, whereas a format string compatible with C's `printf` may be more universal.

Each Primitive must define the `pack` or `format` primitive, but not both. A Primitive may optionally define `unpack` or `regex` if data is returned upon execution of the Primitive.

4.9.2 Format Arguments

Format and pack strings each require arguments, which are given as format arguments. Format arguments are subject to evaluation by the mini-language.

4.9.3 Post-actions

Post-actions are executed by the mini-language after a Primitive has written to and read from its connection.

4.9.4 On Variable-length Messages

Currently variable-length messages are not supported. An example of a variable-length message is one where the first N bytes define the remaining length of the message

in bytes. In order to implement this, some custom modifications to the pack string format will be needed, as well as the regular expressions.

4.10 Local Interface

A local interface is an internal construct only used by the RDIS model. It represents the initial abstraction of Primitives.

4.10.1 Parameters

In general, the parameter count should reflect the primitives that it calls. The units of the parameters should be in world units, not in device-specific units. For example, if a valid SI unit is defined for the type of data it represents, it should be adopted. In the case of the iRobot create, its `drive` Primitive expects values in millimeters. The Interface which abstracts this Primitive accepts arguments in meters.

4.10.2 Types

Local interfaces may be ad-hoc or periodic. An ad-hoc interface must be triggered by a domain interface or a connection event. A periodic interface executes for every iteration of its associated connection's main thread.

As there is no association between connections and periodic local interfaces, there is currently difficulty in defining how and from which thread they are called. Local interfaces are not one-to-one with Primitives, so there is a possibility that one can be associated to Primitives belonging to different connections.

4.10.3 Primitive Calls

Local interfaces have one or more primitive calls. Primitive calls may need to occur in some order, so `priority` is a possible attribute of this call. As well, there may need to be some delay (in milliseconds) between calls so `delay` is also an attribute. A set of arguments may be given which must be a function of the state variables and the parameters given to the local interface.

4.11 Domain Adapter

A domain adapter maps most closely to domain concepts of previous iterations of the model. The change in name is to imply functionality. Previously, a domain concept was the abstraction of kinds of messages which may travel from framework to device and vice-versa. A domain adapter adapts messages to one of these abstractions. The full set of abstractions still remains undiscovered, but a few are used in this solution. They are presented in the following sections.

A convention for all Domain Adapters which contain attributes representing real-world units should be expressed in SI units. Where an SI unit does not exist, some other convention must be adopted.

4.11.1 Range

Range is the abstraction of detecting proximity to obstacles. A variety of robot sensors attempt to do this; for example bumpers and range-finding lasers do the same task with different fidelities. In reality, Range can be one-, two-, or three-dimensional. For this solution, it is assumed to be one-dimensional.

4.11.2 Differential Speed

Differential speed is the abstraction of velocity for differential-drive robots. Most, if not all, differential-drive robots can infer a path from simply a linear and angular velocity. Velocities should be represented in SI units, so meters per second for linear velocity and radians per second for angular velocity.

4.12 Domain Interface

A domain interface represents an entry point for messages originating from the framework. Domain interfaces are given names so that many may exist with the same type.

The implementation of a domain interface is not defined in this formalism. However, a calling framework must know two things in order to make a call to a domain interface:

1. The name of the domain interface it wants to call.

2. A key/value mapping of the message contents in the hosting language's native data structure.

As an example, the Python interpreter has a method `callDomainInterface`:

RDIS.**callDomainInterface**(*name*, *contents*)

where *name* is the name of the domain interface as a string and *contents* is a dictionary with keys corresponding to attributes of the associated domain adapter.

4.12.1 Incoming Domain Adapter

Domain interfaces must have a Domain Adapter on their input. The attributes on the Domain Adapter do not matter because the message is coming from the framework. The code generator only uses the names of the attributes of the Domain Adapter to validate the mapping from Domain Interface to Interface.

4.13 Domain Output

A Domain Output represents an exiting point for messages coming from the model. Domain Outputs are meant to be asynchronous, so the most appropriate implementation pattern is a paired callback approach. So there are a few possible ways this could be implemented:

- Register a callback with each Domain Output name which accepts the message.
- Register one callback which accepts the Domain Output name and the message.
- If synchronous return values for Domain Interfaces are needed, block until a message on the paired Domain Output is delivered. This should be considered bad form as robot architectures are generally asynchronous.

The Python interpreter implements the second choice. First, one registers a callback:

RDIS.**setCallback**(*callback*)

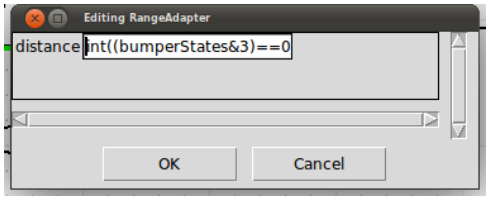

Figure 4.1: Example mapping of Create bumper packet to a Range message.

where *callback* is a one-parameter function name. The argument give to the callback is a Python dictionary with keys *name* and *contents*. The *contents* key contains the dictionary which represents the evaluated domain adapter. Listing 4.1 shows an example of the RDIS callback mapping a domain output message to a ROS publisher.

```
def rdisCallback(msg):
  (pub, callback) = getPublisher(msg["name"])
  callback(msg["contents"], pub)

def registerPublisher(name, pub, callback):
  global gPublishers
  gPublishers[name] = (pub, callback)

def publish_hitObject(rdisMsg, pub):
  env = dict()
  env['distance'] = rdisMsg['distance']
  env['data'] = rdis.safeEval('<distance == 0>', env)
  pub.publish(Bool( env['data'] ))

def main():
  registerPublisher('detect_bump', rospy.Publisher('hitObject', Bool),
    publish_hitObject)
```

Listing 4.1: Example of the RDIS callback mapping domain outputs to publishers.

4.13.1 Outgoing Domain Adapter

The associated domain adapter with a domain output defines a function of the state variables and the format of the message to be delivered to the framework. The message is once again a dictionary with the name/value pairs of the evaluated expressions in the domain adapter.

Figure 4.1 demonstrates a mapping of the Create bumper state variable to a Range message. The Create's bumper packet has bit 0 or 1 set if the left or right bumper is triggered, and the shown mapping in AToM^3 maps the bumper packet to 1 if no bumpers are triggered and 0 otherwise.

4.14 Mini-language

Mappings are defined using a mini-language. Currently, the mini-language is Python. However, since mapping statements have to be stored as strings, a modification is made. Strings of 3 or more characters encapsulated with "<" and ">" are evaluated as normal by `eval()` under the respective scope. For all model elements, State Variables are in scope for the mini-language. Otherwise, Domain Outputs have the attributes of their respective Domain Adapter in scope, and Primitives and Interfaces have the names of their formal parameters in scope.

There are some inherent problems with this approach. First of all, adoption of a textual language is risky because we lose the faculties that models provide. In particular, if the mini-language were to ever change, it would be difficult to migrate existing models using a model transformation. The expressions would need to be parsed and translated into the new language.

On the positive side, Python is an interpreted language and there are libraries for evaluating it in C++ [2] and Java [1]. So if a textual language is to be adopted, it is well worth considering, as well as its conventions because they are ported to two prominent target languages.

The other option is taking a fully modeled approach where the RDIS abstract syntax also contains the logic for evaluation. The disadvantage with this approach is the usability of the model editor and scaling for complex evaluations. The advantage is we regain the ability to use model transformations on these expressions. In the end a textual syntax will need to exist for RDIS's compiled textual form. However, if a modeling tool is adopted which allows varying-precision views of the model, it could be less troublesome to construct the simple arithmetic expressions which RDIS needs in the model editor,

which may then be compiled to whatever textual syntax is chosen.

4.15 Evaluation

This section offers an evaluation on the presented domain model with respect to Selic's characteristics of useful models [48]. Each section will provide a statement about that characteristic in Selic's own words and will provide an argument that the presented domain model has this characteristic, or how it could be improved to better fit that characteristic.

4.15.1 Abstraction

> "A model is always a reduced rendering of the system that it represents." [48]

When writing framework adapters one has to be concerned with two basic components of a driver. The first component is the declarative component, that is, the parts of the driver which are purely declarative. This is the part which RDIS seeks to model, and what is encoded into the RDIS description. It contains information pertaining to how many bytes a Primitive returns or which Primitive an Interface calls. The other component is the procedural component, which can execute a given declarative component.

The assumption is that the procedural component is already implemented (and can work for a substantial number of devices). Therefore, the modeler only must be concerned with the declarative component. For this particular situation, the declarative component is the robot model and the procedural component is the interpreter. The entire system under study is the union of the declarative and procedural components. Since modelers are working only with the declarative component, they are surely working on an abstraction of the entire system.

To quantify this, let us conduct a simple analysis of the iRobot Create model (Appendix B) and the Python interpreter (Appendix A). There is currently no generally agreed upon approach to comparing models to code. Instead, we shall compare the generated RDIS description and the interpreter, as the description is work the modeler must complete. To analyze the description, we shall categorize the lines of code as either

Item	Stuctural	Attribute	Blank	Comment	Code
Create Description	64	107	–	–	–
RDIS Interpreter	–	–	153	326	495

Table 4.1: Contents of RDIS description and RDIS interpreter.

"structural" or "attribute". Structural lines of code have no correspondence to the model and are present as a result of the encoding. Attribute lines of code are semantically rich and correspond to the modeler filling in an attribute on a model element. The interpreter is analyzed for blank, comment, and lines of code. The results are presented in Table 4.1.

If we make the assumption that an attribute constitutes approximately the same amount of work involved in writing a line of code in the interpreter, we can see that the work in creating a description is approximately 20% of writing the interpreter. This does not necessarily mean that creating a description is 20% the work of writing a driver as the interpreter contains code which would not be normally present in a driver, such as the parts which deal with loading the RDIS description.

4.15.2 Understandability

> "Understandability is a direct function of the expressiveness of the modeling form used... A good model provides a shortcut by reducing the amount of intellectual effort required for understanding." [48]

Though there may not be a specific metric suitable for measuring understandability of a model, a modeling language derived from the model may be able to be subjected to more formal analysis. To address understandability, we shall use the AToM^3 implementation of this model (Chapter 5) as the subject of the analysis, and the COMR [66] as the metric. The method for calculation is as follows:

1. The language primitives are enumerated from the meta-model.

2. The primitive is assigned as part of the solution or the problem domain.

3. The sum of solution-level and problem-level primitives is taken.

4. The ratio of the sum of the problem-level to solution-level primtives is the COMR.

The results of this is shown in Table 4.2. As the COMR is a fairly new and untested metric, it is difficult to draw conclusions directly from the result. However, in the trial run of this metric, Wu [66] speculates that WF-CML had evidence of productivity gains with a COMR of 7/4 (1.75). Generally higher values of COMR are desired as this implies the language is more grounded in the problem domain. With this analysis giving RDIS a COMR of 1.375, one may conclude that RDIS has more language primitives rooted in the problem domain, ideally making it easier to understand than having to write the driver in a general-purpose programming language.

As the COMR is not a generally accepted metric, the best way to measure understandability is empirically through a user study. To conduct such a user study, one would need to gather a population from the set of target users and introduce the model to the users briefly. Finally, the users should be asked to model a target device and generate a description. Interesting measurements to take might be the modeling time and the accuracy of the generated description as each of these may suggest the understandability of the model.

4.15.3 Accuracy

> "A model must provide a true-to-life representation of the modeled system's features of interest." [48]

Accuracy is a difficult term to define within the scope of RDIS. Technically, the purpose of RDIS is to translate framework concepts down to the correct device messages, or the same as the purpose of a framework adapter. This can be proved just by inspecting the bytes which are produced by the execution of the model on a computer.

Table 4.3 presents some example Differential Speed inputs and the constructed Primitive command as determined by the iRobot Create description (Appendix B). The columns present the interpreted bytes for readability purposes. By inspection, one can see that the model handles three of the cases perfectly. The stop command (input (0,0)) has a nonfunctional error (only sending 1 or -1 for the radius would cause the robot to behave incorrectly). However, we can see that the current model has some bugs due to the special case required by the Create when turning in place. The Create requires a

Primitive name	Problem	Solution	Equivalence
DifferentialSpeedAdapter	1	0	Twist (ROS)
SerialConnection	1	0	Serial Port
SingleThreading	1	0	Single Threading
Primitive	1	0	Firmware Interface
Interface	0	1	–
StateVariable	1	0	Variable
RangeAdapter	1	0	Range (ROS)
DomainInterface	1	0	Driver Interface
DomainOutput	1	0	Driver Callback
interface2primitive	0	1	–
primitive2connection	0	1	–
connection2threading	0	1	–
connectionStartup	1	0	Startup operations
connectionKeepalive	1	0	Keepalive operations
connectionTerminate	1	0	Terminate operations
domainInterface2Interface	0	1	–
domainAdapter2domainOutput	0	1	–
domainAdapter2domainInterface	0	1	–
interface2domainOutput	0	1	–
Total	11	8	–
COMR	1.375		–

Table 4.2: Calculation of COMR using AToM^3 implementation's language primitives.

1 or a -1 as the Radius for turning in place. However, all these errors may be fixed by modifying the model (or line 160 in the Create's description).

The other way to interpret accuracy is that each of the language primitives corresponds to something in the system under study. However, this has been previously addressed by Table 4.2. Over half of the language primitives correspond to a real entity in the system.

Input (Linear, Angular) (m, $\frac{rad}{s}$)	Actual (Opcode, Linear, Radius) (–, mm, mm)	Expected (Opcode, Linear, Radius) (–, mm, mm)
(-0.2, -0.4)	(137, -200, 500)	(137, -200, 500)
(0, 0)	(137, 0, 32767)	(137, 0, 0)
(0, 0.2)	(137, 0, 0)	(137, 0, -1)
(0, -0.2)	(137, 0, 0)	(137, 0, 1)
(0.2, 0)	(137, 200, 32767)	(137, 200, 32767)
(-0.2, 0)	(137, -200, 32767)	(137, -200, 32767)

Table 4.3: Sample inputs and outputs with expected values.

4.15.4 Predictiveness

> "You should be able to use a model to correctly predict the modeled system's interesting but nonobvious properties, either through experimentation (such as by executing a model on a computer) or through some type of formal analysis." [48]

A traditional analysis approach in the world of modeling is to construct a model transformation mapping the input model to a Petri Net [59] and perform an analysis on the resulting Petri Net (for more about Petri Nets and analysis, see [39]). This may be a possibility in the future. Petri Nets are especially useful in the analysis of asynchronous systems. Player [14] and ROS [43] are both asynchronous systems. As more complex threading systems for RDIS emerge, it is quite possible to define a model transformation of an RDIS model to a Petri Net which simulates the calling behavior of RDIS. The usefulness of this type of analysis is not yet known.

Otherwise, the RDIS model is declarative. There is not so much of a notion of state that can be used to predict any useful or interesting properties. As the model evolves, predictiveness may be a more interesting characteristic explore but as it stands only the stated method is possible.

4.15.5 Inexpensive

> "[A model] must be significantly cheaper to construct and analyze than the modeled system" [48]

Since creating robot drivers is not a widely shared experience, let us start by looking at a few differential-drive robots and their Player [14] drivers in Source Lines Of Code (SLOC) (Table 4.4). Though our metric may not allow for much formal analysis, one can get a feel for the possible size of robot drivers and the relative amount of time to create a testable first-iteration implementation.

Still, it is too early to tell the inexpensiveness of the RDIS model without an empirical analysis. To evaluate the inexpensiveness more study is necessary. Such a study might be a user study where a set of participants are asked to implement a framework driver for

Robot	Files	Blank	Comment	Code
ePuck	15	341	590	1315
Khepera	4	189	282	650
Garcia	2	119	210	357
P2OS	10	823	1312	11266
Nomad	3	749	2577	2965
iRobot Create	3	206	198	920

Table 4.4: Some common differential-drive robots and driver size in SLOC.

a given device and framework, whereas a separate set are asked to implement an RDIS model. Effort could then be analyzed between the two sets.

Alternatively, one may also make an argument for inexpensiveness by proving the portability of a model developed in RDIS. If the domain model can be shown to be compatible with a wealth of frameworks, then the cost of constructing the model is offset by the gains of implementations in many frameworks.

4.16 Conclusion

This section defines a domain model for the generalizable mapping of robotic software framework concepts to device-specific concepts. The motivation for this domain model is to allow for a one-time description of robotic devices for repeated use with many robotic software frameworks. As a part of the contributions of this thesis, the RDIS model has been defined from its general definition to an explicit description and plan for implementation.

The model elements, attributes, and relations between model elements are summarized as a UML Class Diagram in Fig. 4.2, but it is not a formal meta-model. A formal meta-model implementation is described in the following chapter.

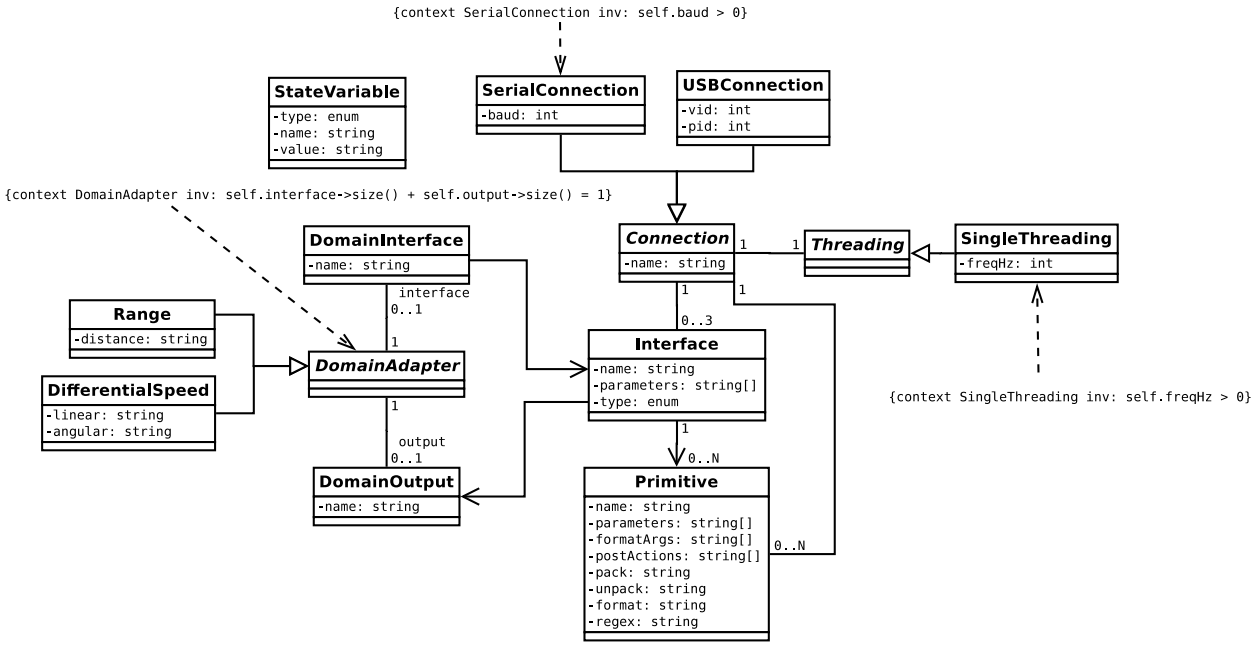

Figure 4.2: A UML Class Diagram of the RDIS domain (meta-)model.

CHAPTER 5

IMPLEMENTATION OF RDIS FOR ATOM^3

This chapter serves as an example implementation for the domain model presented in the previous chapter. It outlines the modeling of RDIS as a meta-model in AToM^3 [25]. In addition, RDIS is given a graphical concrete syntax and code generators are written so that RDIS description files may be generated from models. As well, an approach for modeling robotic software frameworks is shown and all the desirable features of model-driven engineering are presented, including a use of a model transformation [55]. The result is a sample application of this domain model implemented in model-driven tools.

5.1 Introduction

Originally, RDIS was implemented as a DSL using the parser generator ANTLR and its companion tool for code generation, StringTemplates [4]. This implementation generated code to support a few different robot-framework pairs. Later, the model was refined with the aim of supporting a wide variety of robots [3]. The syntax has taken a few different forms, initially taking form as a properties-like syntax and later converted into a JSON subset.

While achieving promising results, there are a couple of flaws associated with this approach. The first is the JSON syntax itself. The general goal in crafting a DSL is to accelerate the speed of development for the person who ultimately ends up using it. However, one can quickly find that a JSON-compatible syntax is not the most intuitive language to craft by hand. However, there is merit in using JSON because of the ease of use in data exchange, particularly via the network, and the wealth of parsers which exist for it in a variety of languages.

The operational semantics of the internal RDIS model were also directly written into

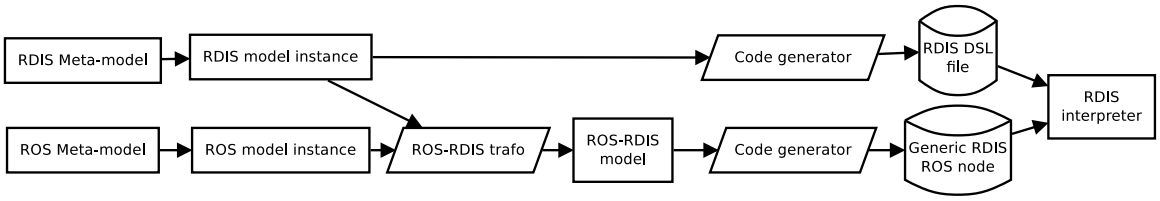

Figure 5.1: Flowchart outlining how the solution works.

the templates, parametrized by the declarative description. While functional, it was generally thought that loading a description into memory interpreting the description at run-time was a more desirable solution. This is a better style for modularity and results in more maintainable components.

Finally, there was a large amount of work involved in maintaining the validator itself. Extending the validator meant a modification to the grammar and possibly the underlying templates. JSON was also a format ill-suited for supporting keywords as to maintain JSON-compatible syntax means to make the keywords indistinguishable from string tokens.

5.1.1 Modeling the Framework

RDIS has no notion of how a framework works. Instead, it demands that the framework itself pass standardized message types into the model and returns messages of a particular type via its output channels. In order to achieve the goal of interpreting RDIS descriptions in the framework, at some point an RDIS adapter must be produced for each existing framework. If a generalized formalism for describing frameworks existed, RDIS could be empowered to generate its own framework adapters.

However, frameworks vary in their internal messaging systems and building a generalized formalism for modeling them is a large undertaking. Given this is not a well-understood task it is acceptable to limit the scope for the purpose of discovery.

5.2 Solution

A solution to the described problem has been implemented in AToM^3 for a course in Model-Based Design under Dr. Eugene Syriani of The University of Alabama. The described solution provides the following:

- A modeling formalism for RDIS
- A modeling formalism for the Robot Operating System (ROS) nodes [43].
- A model transformation for assigning the mappings between ROS and RDIS.
- An interpreter for RDIS descriptions.

Before delving into the workings of the solution, it should be noted that pieces of the original implementation of RDIS have been abandoned. For example, the de facto RDIS parser is not used because the textual syntax of RDIS was changed during the scope of this project. The parser may be abandoned altogether due to maintenance difficulty. The focus is less on the textual syntax and more on the model. The textual syntax is now viewed as a medium to communicate the model to the interpreter, not intended to be edited directly by programmers.

Figure 5.1 shows a flow chart for a general picture of how the solution works. There are two meta-models at work: a ROS meta-model and an RDIS meta-model. Valid instances of each of these meta-models may be fed as inputs into a model transformation which produces a hybrid ROS-RDIS model that contains the mappings between each of their message styles. The model produced from this transformation may then be compiled into a ROS node. Any Abstract Syntax Graph (ASG) containing a valid RDIS model may be serialized into an RDIS textual description.

The rest of this section is divided into subsections which explain how each component of the solution works in detail.

5.2.1 The RDIS Formalism

The RDIS formalism seeks to provide an easier way for developers to create robot descriptions conforming to the RDIS model. The formalism is implemented using AToM^3. Each of the components of the RDIS model is assigned a visual concrete syntax. Valid model instances can then be serialized to their equivalent JSON form.

Many of the original model elements outlined in [3] are present in the implementation. However, much of the model was changed during this implementation. This section gives a higher-level overview of generally how the RDIS model works and presents the modeling

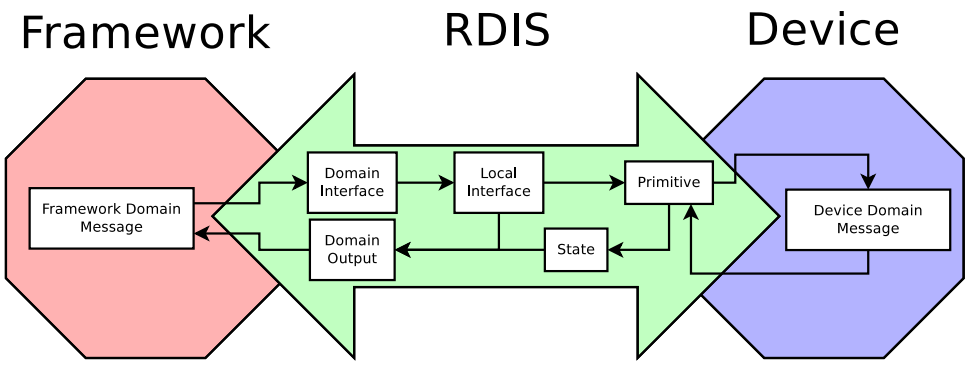

Figure 5.2: An overview of the RDIS architecture.

environment created for RDIS. Chapter 4 offers a more complete documentation of the RDIS model. The system to be modeled in AToM^3 is demonstrated as a UML Class Diagram in Fig. 4.2.

5.2.2 Background for RDIS

The general goal of RDIS is to bridge the gap between framework and device in a way that is standardized and general for a large number of devices and frameworks – in particular, the construction of a communication bridge between the framework and device. At a high level, the solution to this is simple: standardize the framework-RDIS interface and the RDIS-device interface, and connect the two using intra-model message passing. Figure 5.2 illustrates this concept.

Given the complex network that can form inside the RDIS layer, this can be a difficult model to visualize in a text-only environment. A visual modeling environment can alleviate the work of creating the robot description by allowing the modeller to visually organize the description, and can offer more helpful feedback in the event of an error.

Modeling Environment

A formalism for AToM^3 was created. An example of the resulting concrete syntax is demonstrated in Figure 5.3. The modeling environment offers a wealth of incremental and compile-time constraints guiding the modeler to create valid model instances.

The significant advantage of a modeling approach is the ability to visualize the web of relations between the model components. Previously, in a textual syntax all of these

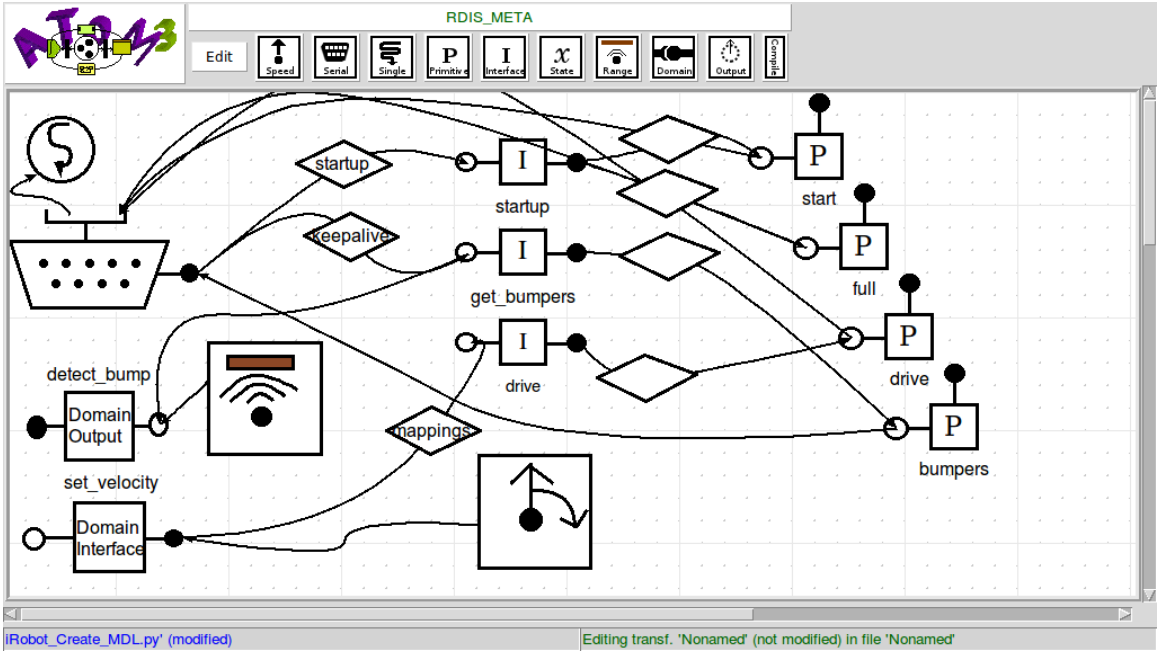

Figure 5.3: RDIS modeling environment with concrete syntax for an iRobot Create.

relations would need to be specified by name. On top of placing the strain of remembering the names of the model elements on the modeler, this also introduces the potential for misspellings. A validator would need to check for broken name references to check if the model is well-formed. However, as the model will almost always contain circular references, this sort of name validation must be completed in multiple passes over the input, which can be difficult to implement.

With a modeling approach, broken name references are an impossibility because the correct name of the model element can always be drawn by following the relation edge. If no edge exists, a constraint violation may be raised or that attribute may not be serialized during code generation. Small, incremental changes may be made quicker because a modeler may locate the component that needs to be changed based on human heuristics.

5.2.3 The ROS Formalism

With the robot description formalism created, the next step is to examine how one might create the adapters which become the framework-RDIS bridge. The problem becomes simpler when targeting just a specific framework. For the purpose of discovery, this paper focuses on ROS. The reason for the choice of ROS is because of its recent rise

to popularity, growing community, and a communication model which is easy to grasp. ROS is also multi-lingual.

ROS at its core is a publish/subscribe communication broker, but the project itself revolves around creating a package management system for robot software and creating bindings between existing robot applications and the ROS core. Robot applications, when written for ROS, are generally organized into packages. Packages are subsequently divided into nodes, the basic execution unit, and message specifications at the basic level. Many nodes may be a part of a package and nodes may communicate with other nodes outside their package.

This solution focuses on generating the node itself rather than the entire ROS package. The goal of modeling an entire package is left to future work. The next subsections explain each of the model elements, and finally the transformation which bridges the ROS formalism to the RDIS formalism.

Nodes

Nodes are the basic execution unit of ROS. For each instance of a node in the modeling environment, a node will be generated, sharing a filename with the model element name as well. A node should publish or subscribe to a few different topics. A subscription to a topic means the node will listen to that topic and react to messages which are passed via that topic. Publishing to a topic generally happens periodically (e.g., reading a sensor), or in response to a message which travels in via one of the subscribed topics.

Topics

Topics are nothing more than a named data channel. Topics are strongly typed, and a valid ROS type must be specified for each topic. Topic types may be one of the common types shipped with ROS, or it may be a nonstandard, custom type which only exists for one particular package.

Types

As mentioned previously, topics must have a type. ROS types are defined via a mini DSL so that the ROS types may be generated for many different languages. A good approach to modeling types should reverse engineer this model so that any general ROS type could be constructed. This might be counter-productive as the goal of RDIS is to use standard types on either side of the framework-RDIS interface. Instead of fully modeling the ROS type system two particular types were selected. This presents a very constrained modeling environment. While it is expected that the chosen set is not complete, it is hoped that a finite set of framework and RDIS types such that an adequate number of devices and applications may be supported.

We should take a moment to become familiar with the types used in this project. Pose is a 6-dimensional data type which describes both position and orientation in space. Twist is the time derivative of Pose, and it is how ROS represents a change in position and orientation over time. The second ROS type modeled in this project is a Boolean, which is a wrapper type for booleans in the host language.

The Modeling Environment

Each of these model elements was assigned a concrete syntax for use in AToM^3. The result is a modeling formalism for generating skeletons of ROS nodes in that enough information is present to define the subscriptions and publications for a given node, but the actions that must be taken in response cannot be generated. The modeling environment is shown in Figure 5.4.

The formalism is not helpful until its relationship to the RDIS model is defined. This may be done using a model transformation.

ROS-RDIS Transformation

A transformation which defines the bindings between RDIS domain interfaces and domain outputs has been created. It takes as input a ROS skeleton node model and a full, valid RDIS model and maps subscribed topics of the ROS type Twist and creates a mapping to

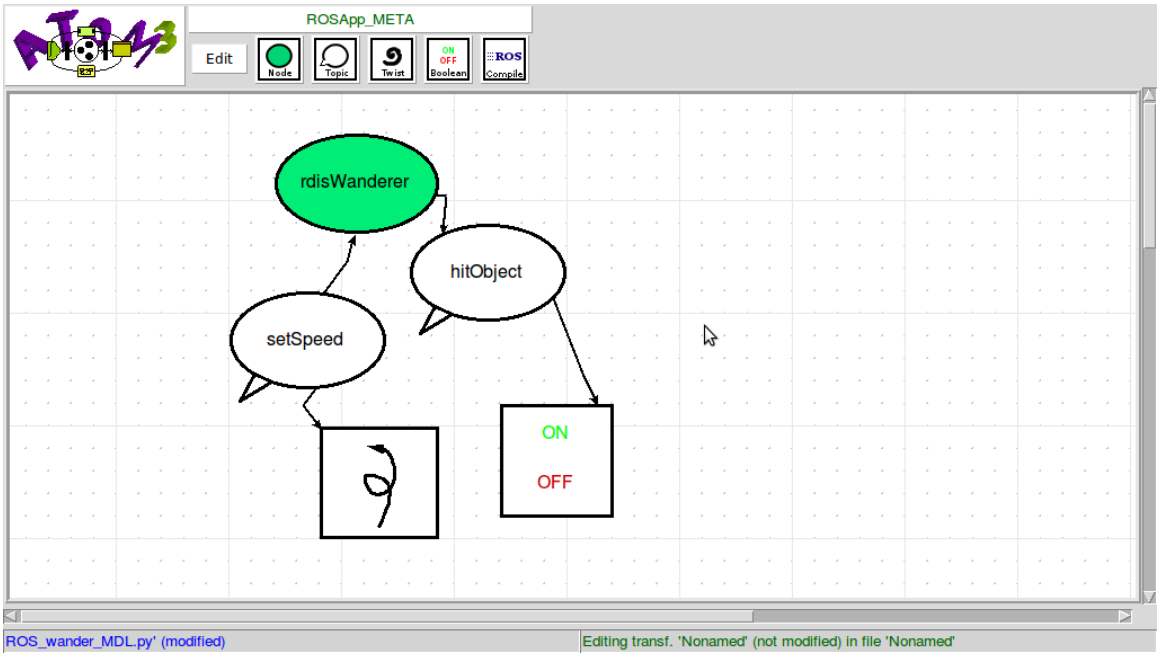

Figure 5.4: ROS modeling formalism with an example skeleton node.

domain interfaces which accept a Differential Speed. Differential Speed is just a special case of Twist where one linear component and one angular component are needed to describe motion.

Similarly, we can define a mapping between Range and Boolean. Range offers a one-dimensional view of how far ahead an obstacle is located. For bumper type sensors this is an extremely low fidelity view, but for more advanced laser systems one can measure an actual distance to the object. We define when an object is colliding with the robot, the range on the operative sensor should be zero. If an application only needs to know if the robot has bumped into something, and it is subscribing to a Boolean-type topic with the expectation that the message will be provided, we can map a Range on a domain output to that topic by a logical comparison of the Range value to zero.

The resulting model of this transformation then has enough information to fully generate a ROS node. Figure 5.5 shows the LHS/RHS transformation rules used by AToM^3 to perform the transformation.

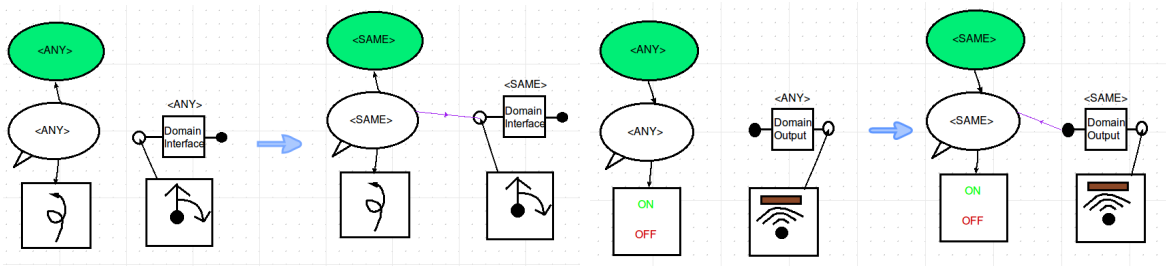

Figure 5.5: Transformation rules for the ROS-RDIS model transformation.

Alternate Approaches

Another (probably better) transformation would be to actually create the ROS model from the RDIS model. The above transformation made the assumption that it is important to actually have a tight coupling between the domain output name and the names of the topics used by the application node. For ROS, this is not true and it is bad style to strictly enforce topic names. Instead, it is better form to have the generic RDIS node publish or subscribe to topics which use the name of the domain interface or output, and simply rewrite topic names using the `rosrun` utility's command line parameters.

5.2.4 Code Generators

Code generation is the ultimate goal of the project. There are two artifacts to generate: a textual RDIS description and a ROS node. Two different strategies are adopted for each due to the nature of the artifact generated.

RDIS Code Generator

An RDIS description is a declarative description for a robot's communication model. Given that it is declarative, it can have a very natural correspondence to a data structure in computer memory. Because of this, the context object pattern [22] of code generation is adopted. For Python, this just means a dictionary is constructed which represents the model.

Python has a built in JSON serializer (the `json` module [41]) which natively knows how to serialize dictionaries and lists to their equivalent JSON types. The strategy used for code generation in the solution is to make a pass over the input AToM^3 abstract

syntax graph and build a context object using simple Python types. At the end of the pass, the context object is then serialized to an output file.

Listing 5.1 shows an example of the generated textual syntax for a Primitive from the iRobot Create robot. A couple of the differences between the abstract syntax and the textual syntax may be seen. The primitive-connection relation for instance is not represented by the containment of a connection by a primitive, but rather a by-name reference to the connection.

```
1  {
2    "postActions": [
3      "bumperStates = __out__[0]"
4    ],
5    "name": "bumpers",
6    "formatArgs": [
7      "<142>",
8      "<7>"
9    ],
10   "connection": "btserial",
11   "unpack": "B",
12   "pack": "BB"
13 }
```

Listing 5.1: An example of a generated primitive.

5.2.5 ROS Code Generator

A ROS node is also generated. A ROS node may currently be in C++ or Python; this solution generates ROS nodes in Python. As Python is a full-fledged programming language it is difficult to build a context object and serialize as done before. Fortunately, ROS nodes have a lot of boilerplate logic, so a template-based code generation scheme works quite well. The benefit for this is that the templates can be changed easily without modification to the code generation module itself.

The approach used is to write templates for each repeatable block of code we may see inside of a ROS node, and map them over incoming sequences from the AToM^3 abstract syntax.

Listing 5.2 demonstrates a callback generated for a subscribed topic which receives a Twist object. The goal is to map the Twist object into its equivalent Differential Speed message. The RDIS interpreter accepts dictionaries containing the name-value pairs it expects to see in a Differential Speed as its argument. The code generator will pull the expressions from the Differential Speed adapter in the abstract syntax, overwrite the key in the dictionary which shares the name of each attribute with the result of the expression that attribute is tied to, and call the relevant domain interface.

```
1  def setSpeed_callback(data):
2    global gModel
3
4    angular = (data.angular.x, data.angular.y, data.angular.z)
5    linear = (data.linear.x, data.linear.y, data.linear.z)
6
7    env = dict()
8    env["angular"] = angular
9    env["linear"] = linear
10
11   env["angular"] = rdis.safeEval("<angular[0]>", env)
12   env["linear"] = rdis.safeEval("<linear[0]>", env)
13
14   gModel.callDomainInterface("set_velocity", env)
```

Listing 5.2: Example of a generated callback for a subscribed topic.

5.2.6 RDIS Interpreter

For this solution, an interpreter was written in Python. The interpreter is derived heavily from the meta-model in the design of the modeling environment with each major entity having a Python class describing its operational semantics with respect to its instance variables.

The interpreter was written because previously there were no existing tools for RDIS besides a parser and several accompanying templates. However, the syntax and model had changed greatly over the scope of the project, so this solution opted not to use the original parser. Instead, a custom parser was written specifically for constructing the

Python model.

Parsing

Since a model has been serialized to JSON, stock parsers may be used to load the model from its abstract syntax. Validation is not implemented in the interpreter. This is because the primary artifact is the model. If the model was invalid, a constraint would have been raised and the description would not be generated. Therefore, the descriptions can be assumed to be valid. As a sanity check, the interpreter can execute the model and the result may be observed for validity.

Interpreting

Interpretation is simply a series of intra-model message passing. On the side of the framework one has the option of which domain interface to call. The model will decide which primitives to call as a result. Similarly, after executing a local interface, the model will decide if it needs to invoke a domain output which eventually invokes a framework-side callback with an RDIS message on its payload.

Eventually, the model will need its own internal threads, but this is not implemented at present. The reason for this need is calling periodic interfaces and the keepalive interface.

5.2.7 Style

The RDIS concrete syntax is designed with the presumption that the framework lies on the left side of the string and the device on the right. Therefore, as the level of abstraction gets closer to the framework, model elements should be placed on the left. As it gets closer to the device, they should be placed on the right. There are no particular style guidelines for placing connection, threading, and state variable model elements. There are no topological constraints on the language so the semantics remain the same regardless of placement, but this style aids in creating readable models.

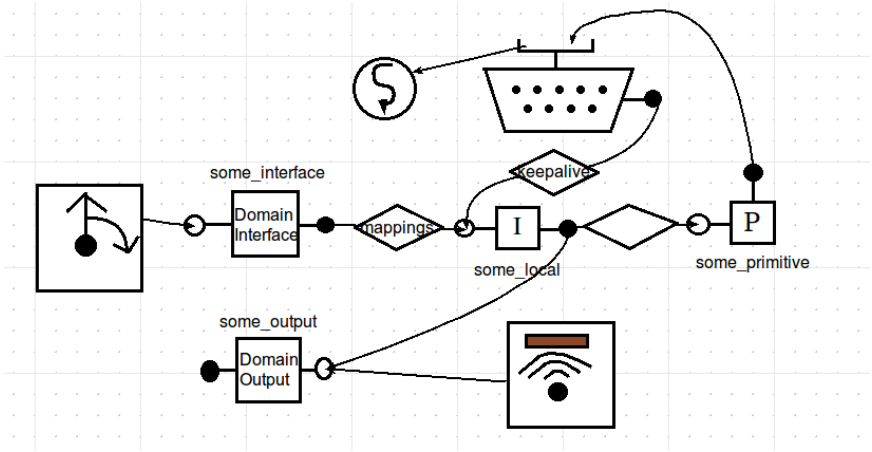

Figure 5.6: Example of the left-to-right nature of the concrete syntax.

5.3 Evaluation

This section presents an evaluation of the performance of the model-driven means of creating robot drivers as opposed to the traditional method. We will address the modeling speedup, analyze maintainability, and remark upon some limitations and assumptions.

5.3.1 Modeling Speedup

The speed of modeling is the most significant gain in this method. A developer who is familiar with the model, modeling tool, and the device can generate an initial description in less than an hour. If the generation target filename is set up correctly, incremental changes can be made quickly, redeployed, and tested at the modeler's pace. The developer will spend most of his or her time refining the mappings between framework messages and RDIS messages, which is the desired outcome. Where the alternative is manually creating an RDIS description in a text editor with no feedback, the model editor is the clear winner.

Some drawbacks to this method is that the model used is not one which is universally understood by robot software developers. As well, most robot software developers will be more accustomed to hand-crafted code rather than code generation. Finally, communication models are not normally described declaratively, but rather in a general-purpose programming language. Thus, any model-based approach would seem very foreign to a robot software developer, and would take some time to be adopted by the community as a whole.

For generating ROS nodes, the speedup may not be so noticeable. This may be due to the way that the transformation is defined. There is a very clear constructive mapping from an RDIS description to an equivalent ROS node. The current solution requires an explicit design of a ROS node. A better solution might generate the subscriptions by discovery at runtime of domain interfaces or domain outputs, and a static mapping of ROS types to RDIS types and vice-versa. Another possible solution is designing a transformation which creates ROS instance models from an RDIS description.

5.3.2 Maintenance and Refactoring

Assuming a static meta-model, long term maintenance should be achievable. If changes within the device specifications occur, then those changes should not often propagate beyond just one model component. As an example, if the Create's "read bumpers" primitive changed from returning one byte to two, then only that primitive's unpack string and post-actions would need to be modified.

Trouble might arise from changes in the meta-model. The addition of types of model elements could be supported without much trouble. A simple deletion, however, would invalidate all current model instances and a transformation would need to be defined to recover them. The main problem would be with the interpreter. The operational semantics of RDIS are currently not modeled, so keeping the interpreter synchronized would be an issue. The code generators also are not modeled, so they would have to be changed by hand.

5.3.3 Limitations and Assumptions

RDIS is currently limited to describing disconnected devices which are controlled via some transport. Only robots which use fixed width messages can be modeled. Robot systems which are in part heterogeneous (e.g., an attachment) may work in its current state but is untested. Threading is still an unsolved issue.

Some assumptions are made which make RDIS possible in concept. The first of which is that a finite set of framework-RDIS messages exists such that a good portion of robot applications can be supported. There is still research to be done in the categorization of messages which may travel between the robot and the framework. This categorization of

messages will be based on existing defined interfaces within each framework.

5.4 Conclusion

The problem of creating drivers for pairs of robots and frameworks is significant and if left unchecked will leave the robotics community littered with incompatible combinations of devices and frameworks. RDIS models the communication between framework and device and vice-versa. When formalized it can generate code for a variety of platform-device pairs. Previously, RDIS was thought of as a DSL and adopted a JSON syntax. However, the tools were difficult to maintain, there was no editor for the DSL, and the complex syntax made it difficult to write descriptions. This solution puts more emphasis on the model and treats the textual syntax as a compiled form of the model. The result is improved creation of the RDIS textual description. The solution also demonstrates the possibility of compiling ROS nodes from an RDIS model.

There are still many flaws in the current implementation. Threading is largely ignored. Fixed-width messages can only be modeled. ASCII-encoded messages have not been tested. Periodic interfaces are not considered in the interpreters, and as well scheduling is not handled in the interpreter. Also, more attention should be given to the compile-time constraints so that no invalid RDIS textual descriptions may be generated.

Future work lies in defining the set of domain interfaces and domain outputs. This is an essential assumption of RDIS so it is important these are defined and standardized. More work will be done in the coming months in the specification of the physical construction and kinematics of robots. Even then RDIS is not complete. More work needs to be done in the enhancement of robot state, exception handling, sensor and actuator error modeling, and improved modeling of threading in addition to that outlined above.

CHAPTER 6

ON ROBOT KINEMATIC DESCRIPTION

When modeling robotic devices, not only is it important to model how to control the robot, but a kinematic and dynamic description of the robot can be instrumental when prototyping application code. With a kinematic and dynamic description, the robot can be simulated in a digital space making prototyping and debugging a robot application much easier. As well, a proper visual model of the robot can aid the developer in visualizing the robot in simulation, and a functional collision model is necessary for proper simulation.

Within the context of a kinematic and dynamic description, we encounter a problem not unlike the one previously described. There exists a large selection of robotic frameworks to work with and each of them adopts its own preferred format for a kinematic, visual, and collision description for the robot in question. In the previous case for robot control, this meant that a control module would need to be created – usually meaning some developer would need to write some code which utilized that framework's SDK. However, in the case of simulation description, the problem may not be able to be approached in the same way as before.

Not only is the problem of uniformity across simulation descriptions fairly undiscussed, but the various frameworks that do exist are disjoint in how they choose to model robots. Within robotic frameworks, there are two main categories of frameworks: those where the description format is fitted to robotics and those where it is fitted to the framework. These two cases will result in different descriptions for the same robot. A simulator designed to support specifically robots, such as Gazebo [21], might choose to focus on modeling the links and joints of the kinematic tree. An extension to a more general-purpose application will forced to adapt its description format to what is expected by the general-purpose framework, as is the case with PREOP [63] and USARSim [7].

6.1 Problem Description

Backtracking to the original problem of modeling robot control, a robotics stakeholder who ports his or her device between frameworks might run into problems understanding the messages and architecture that framework adopts. In this current problem, the stakeholder will likely find trouble in understanding the elementary concepts of the framework's simulation description when moving between framework categories. The solution to the previous problem was to create templates for the control module, however in this case we may not be able to create templates for simulation description using the same tools.

Simulation description files tend to be text-based [40, 56, 64, 65], and as a result they have the option of being hand-crafted or produced using tool software. As the simulation descriptions are themselves models, certain simplifications are utilized. For example, the body of a robot might be box shaped. The box could be modeled in high fidelity specifying many points along the box's surface, or the length, width, height and position may be given. It is fairly common to do the latter, but both are valid representations. This type of representation may be done for any number of reasons, including human readability, the conciseness of the representation, ease of collision detections, and lower computational complexity.

The major problem is that some frameworks use different representation strategies. While one may choose a sphere to model one part of a robot, the other may allow only mesh coordinates. If the new model represents the said part as a sphere, it may easily translate to one framework, but it would be difficult to translate. If the model were to be equipped for translation, more sophisticated code generators may need to be written.

The goal of this research will be to investigate the problems encountered when modeling robots for the purpose of being cross-framework compatible. In particular, the following research questions should be addressed:

- What are the major components for a viable model of a robot's simulation description?

- Which parameters are necessary in order for the said model to be taken to other frameworks?

6.2 Motivation

The model of software development introduced in Figure 1.1 has the potential to waste much time. With the proposed changes to the software development model, the motivation is to investigate what the parameters are that can universally model a robot so that a single description can be used in multiple frameworks. This fits with the overarching goal of the RDIS project to save development effort by creating a domain model which facilitates code reuse. Additionally, the viability of the created model can then be established through actual simulation and a conclusion can be made about a fitting standard for a simulation description.

There is also a unique benefit to having the simulation description and the communication model described in the same model. When the two are colocated, it is possible to define a mapping between the interfaces and their parameters to the effect on the robot's pose. Therefore, if these mappings are to be investigated further it would necessitate the existence of a single generalized simulation description model.

6.3 Description Languages of Simulators

When examining the frameworks themselves, the platform-specific simulation description format that they accept can be viewed as a kinematic model of the robot itself, so it is important that each of the individual frameworks be analyzed for their approach to modeling a robot's kinematics and dynamics. The syntax can be very different even when describing the same robot. The appearance of the description file is a function of the features of that robot, the parameters required by the simulation framework, and how the framework designers have decided to format the description file.

Figure 6.1 demonstrates this fact. It shows excerpts from the Stage kinematic simulator [40] and Webots dynamics simulator [56]. Stage uses a custom tree-like syntax, while Webots opts to use its own custom flavor of VRML. The result is a variety of description

```
define roomba position
(
  size [0.33 0.33 0.1]

  block
  (
    points 16
    point[0] [ 0.225 0.000 ]
#     . . .
    point[15] [ 0.208 -0.086 ]
  )

  color "gray50"
)
```

```
DifferentialWheels {
  children [
    Shape {
      geometry IndexedFaceSet {
        coord Coordinate {
          point [ . . . ]
        }
        texCoord TextureCoordinate {
          point [ . . . ]
        }
        coordIndex [ . . . ]
        texCoordIndex [ . . . ]
      }
    }
# ...
```

Figure 6.1: Excerpts of Stage and Webots simulation description files.

formats whose syntax and semantics will vary based on the choices of the framework developers.

The following sections will examine in depth how existing simulation frameworks choose to describe robots. This will provide insight as to how, in abstract terms, frameworks choose to describe robots and suggest the best model for describing them. As well, the technical choices made will help to address the ease of translation to that framework.

Kinematics Simulators

The Player/Stage project [14] is a staple of the robotics community in part due to it support for several devices and its being free and open-source. While Player is the component that controls the robot, Stage [60] is the simulation component to be examined. It is a 2D kinematic simulator whose focus is to support multiple robots in a single simulation with ease. Stage uses a simple tree-like configuration file [40] to model specific components which are attached to the robot body. As the simulation is wholly kinematic, it does not accept a wealth of physics parameters. The visual components of the simulation are primarily represented with simple boxes or other arbitrary polygons extended into a 3D space. The boxes which represent the robot may be colored. The collision bounding boxes and the visual description are shared. The attraction to using Player/Stage in re-

search is that application code can be written once to be used in simulation or in physical experiments. In this sense, the control model is aware of the kinematic model and knows how to manipulate it in simulation.

Dynamics Simulators

Gazebo [21] is a simulation framework closely tied to the Player/Stage project. It is similar to Stage in that it is Player-compatible, but it is a fully dynamic simulator. Gazebo's physics simulation is handled by the Open Dynamics Engine (ODE) [49], and the graphics rendering is handled by the Object-Oriented Graphics Rendering Engine (OGRE) [53]. Gazebo represents robot models by means of its own custom file format, the Simulation Description Format (SDF) [65], an XML subset. SDF adopts a familiar link-joint model for its descriptions of the robot model. Its links are defined in a flat manner (links do not contain other links in the syntax), and joints are declared afterward, pairing parent link ID to child link ID. Joint range of motion is defined by a simple lower and upper bound. The result is a fairly human readable file format where a robot may be partially visualized just by inspection.

Robot Operating System (ROS) [43] is a framework currently maintained by Willow Garage. The entirety of ROS provides an operating system-like manner of software development for robots. A part of the ROS project gives specifications for the Uniform Robot Description Format (URDF) [64] which, like SDF, is an XML subset. URDF bears a strong similarity to SDF, as demonstrated in Figure 6.2. This comes as no surprise, as Willow Garage is a major player in the Gazebo project and Gazebo is even included as a part of ROS. The main difference between the two seem to be that URDF is generalized so as to separate it from the simulation framework. In the case of Figure 6.2, the behavior of a named material is different. In SDF, name would refer to an installed script outside of the description file, whereas in URDF it refers to the name of a material defined just beneath the root element of the XML tree. Whereas an SDF description may depend on how the emulator is set up, URDF is more simulator-independent as it is more self-contained in this case.

```
<link name="body">
  <inertial mass="1.0">
    <origin pose="0 0 0.5 0 0 0"/>
    <inertia ixx="100"  ixy="0"  ixz="0"
      iyy="100" iyz="0" izz="100" />
  </inertial>

  <collision>
    <geometry>
      <box size="1 1 1"/>
    </geometry>
  </collision>

  <visual cast_shadows="true">
    <geometry>
      <box size="1 1 1"/>
    </geometry>
    <material name="Gazebo/Grey"/>
  </visual>
</link>
```

```
<link name="body">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <inertia ixx="100"  ixy="0"  ixz="0"
      iyy="100" iyz="0" izz="100" />
    <mass value="1"/>
  </inertial>

  <collision>
    <geometry>
      <box size="1 1 1"/>
    </geometry>
  </collision>

   <visual>
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Grey">
      <color rgba="128 128 128 1.0"/>
    </material>
   </visual>
</link>
```

Figure 6.2: Excerpts from the SDF and URDF documentations.

Webots [27] is a commercial dynamics simulator for robots. This framework marks the first real departure from the shared syntax of SDF and URDF. The Webots User Guide [56] describes a node-based model for its simulations, including the robots. The description file itself is a tree-like format with each node in the tree. The nodes can represent anything from a transformation, shape, or coordinate set. Webots appears to enumerate categories of robot it can support. For example, the iRobot Create Webots scene description mentions a specific "DifferentialWheels" node. Interestingly, the scene description does not explicitly include the joints of the robot. It is also of note that Webots depends heavily on exhaustive specifications of points for high fidelity geometries (used by the visual model) and specification of texture coordinates. This is probably necessary to achieve a realistic visual model which Webots strives to do, but it results in a description file overburdened with non-human-readable information. Webots also uses ODE as the driving force for its physics simulations.

Other Frameworks

Other frameworks use more general-purpose modeling formats in order to model robots. These are examples of frameworks which have been extended from an existing framework which was not created for the express purpose of modeling robotics. As an example, USARSim [7] is a robotics simulation framework with a focus on search and rescue. It is built on top of the Unreal Engine and uses that engine's binary-encoded modeling formats to build the physics descriptions of the models with support from UnrealScript, Unreal Engine's scripting engine. As another example, PREOP [63] is a simulation framework built with computer science education in mind, meant to provide robotics programming experience to those with low exposure to computer science. Its model descriptions are zip-encoded with underlying XML properties files and binary-encoded vertex lists.

Herein lies the difficulty in translating to other frameworks. The aforementioned frameworks make difficult translation targets mostly due to their binary encoding. The realization of certain parts of the model may be lost in the translation as the underlying model for them is not based on robotics. However, they are simultaneously very attractive targets as they may have the highest potential for reduced work.

6.3.1 Analysis

Between all the mentioned robot simulators there appears to be a disagreement over the proper format for robot description. From conducting this survey, the following parameters seem to be most important when analyzing for compatibilities between any two simulators:

- The **file type** is an indicator of how easily a particular format may be read or created. Raw file types will tend to require more effort to create an encoder or decoder simply because it is not as human-readable. Text-based file formats are more readable and represent an opportunity for translation.

- **Simple shapes** are simple geometric shapes that are available to use in the description language, such as a sphere. An example of a nonsimple shape would be a triangle mesh. Simple shapes are a compact means to specify a part of the collision

Table 6.1: Robotic frameworks and their description format characteristics.

Description format	File type	Simple shapes	Link-joint?
Stage	Text-based	Yes	No
SDF	Text-based	Yes	Yes
URDF	Text-based	Yes	Yes
Webots	Text-based	Yes	No
M3L	Text-based	No	Yes
Alice	Text-based	No	No
USARSim	Raw	No	Yes [1]

[1] The Unreal Engine, from which USARSim is built, uses a proprietary skeletal animation format. Collision and animation skeletons are created for the robots in USARSim.

model or the visual model itself. A description format utilizing simple shapes might be translated into a framework which exclusively uses meshes; however the reverse, would be more difficult.

- Some frameworks follow the **link-joint model** while others do not. This marks a stark departure between frameworks in the underlying model that they assume. The frameworks which do not assume a link-joint model will often enumerate a robot category, like a differential-drive robot, in order to offload the complete description of differential drive kinematics to the simulation application itself. A more model-driven approach would be to let the model itself describe the kinematics of the robot, not to rely on maintaining the platform-specific implementation of those kinematics.

Table 6.1 shows some of these main characteristics of robotic frameworks for the reviewed simulation frameworks. Previous experience suggests that a robot model should be text-based, support some sort of simple geometric shapes, and should follow a link-joint model. From the table it would appear that SDF and URDF are the most fitting description types. This is because a text-based format is easy to parse and read, simple geometry gives us the flexibility to target other frameworks using simple geometry or exclusively mesh geometry, and the link-joint model is general enough to describe many kinematic categories of robots, not just particular ones.

```
<transmission type="SimpleTransmission" name="caster_front_left_trans">
  <actuator name="caster_front_left_motor" />
  <joint name="caster_front_left_joint" />
  <mechanicalReduction>1</mechanicalReduction>
</transmission>
```

Figure 6.3: An example transmission excerpt from the URDF documentation.

However, SDF and URDF are not without their problems. Both are designed with a specific application in mind. By examining their deficiencies we can extract what is missing from the current state of robot kinematic modeling.

Recall that SDF is the format used by the Gazebo simulator. SDF has some small problems where it may reference description scripts which are shipped with the simulator itself, but this is trivial to fix. While it may be possible to define the links and joints which compose the robot, it is currently not possible to declaratively define the forces which drive those joints. These definitions are offloaded to plugins or the simulator itself.

The ROS project's URDF is interestingly the only format to remark upon a transmission. A transmission within URDF defines a module which knows how to apply an effort to a joint. However, the shortcoming in this approach is one has to reference a piece of a platform-specific implementation in order to map transmissions to joints. Figure 6.3 demonstrates an example transmission. The excerpt defines a transmission `caster_front_left_trans` which actuates the joint `caster_front_left_joint`. The `SimpleTransmission` is the name of a C++ class which actually does the work of applying the effort to the joint. The actuator `caster_front_left_motor` references an external symbol in a lower-level specification of the robot actuators.

The URDF documentation itself [64] also mentions that the transmission extension is not meant to be used by anything but the PR2 robot. While being the only description format from the review to model the transmission, it depends on pieces of a platform-specific implementation. This indicates that there is further investigation to be done in order to discover how to appropriately model transmission. Additionally, there are no mappings defined between any sort of control interfaces to the transmission in the model itself which would be a problem to explore as well.

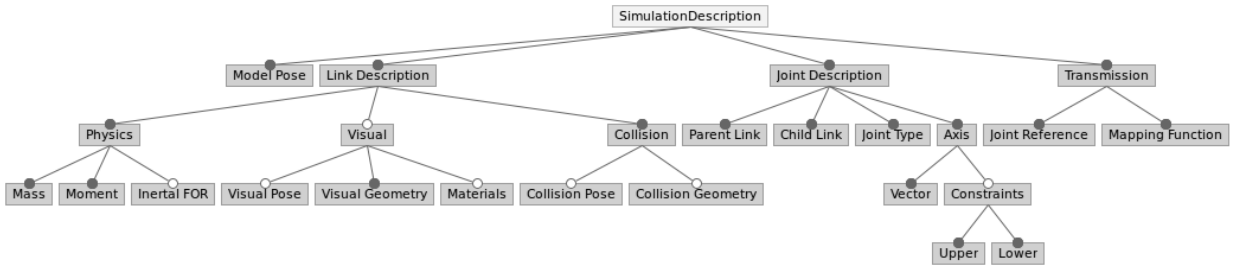

Figure 6.4: A feature diagram depicting the common features of a simulation description.

When exploring the range of description features offered by simulation frameworks, there are certain equivalent features that appear in several of them, at least in the cases where those features exist. Figure 6.4 shows a feature diagram of kinematic description. Links are composed of three essential components: inertial, collision, and visual. The inertial component describes the mass, rotational inertia, linear and angular damping, and the inertial frame of reference. The collision component defines a geometry to be used by the collision simulation, along with the frame of reference for the collision model in reference to link space. The visual component also defines a geometry to be used by the rendering engine, a frame of reference also with respect to link space, and relevant lighting parameters for the link's material description. A joint description consists of a parent and child link, along with a joint type which is supported by the specific platform. The joint axis is defined with a normal vector. Constraints may be applied to define the joint range of motion, or they may be ignored for continuous joints. The transmission is something not currently well-defined, but it is apparent that its goal is to drive some joint so it will at least require a joint name. The entire model itself has some frame of reference with respect to the world space.

6.4 Approach

From the previous sections it is apparent that the robot kinematic modeling languages are missing necessary components for a generalized description for use in simulation. Some description formats do well in defining the geometries of the robot, but make categorical assumptions about the kinematics of the robot, offloading the actual kinematic

description to the framework itself. As an example, Stage has a `drive "diff"` statement to label a position element as having differential-drive steering, and Webots has a `DifferentialWheels` node which acts as the base node of a robot description having differential steering. URDF solves this by defining a relationship between transmissions and joints, but its approach is specific to the PR2 robot.

Current models of kinematics are adept at modeling the construction of the robot in a way that is almost a standard. However, what is missing between all of them is how to model controllable joints. If a solid mapping can be defined between programming interfaces and resultant efforts at joints, this would benefit the robotics community greatly as there is not currently a description format which does so without the aid of a companion plugin or hardware description. RDIS would benefit much from this component as well, as it would define the bridge between the control model introduced in earlier work [3, 4] and the kinematic model which it currently lacks. Thus, the work that needs to be done is to define RDIS' kinematic model according to the trends seen from other frameworks, and then provide a mapping from interfaces in the control model to transmission efforts at joints in the kinematic model.

It is clear that a viable robot kinematic model will somehow take heed of some of the existing description formats to ensure there is sufficient compatibility with the model. The problem lies in deciding which parameters are essential to a proper kinematic modeling and which are parameters which lose meaning when carried to other frameworks. An additional goal is to find a concise set of these parameters such that no one parameter may be defined in terms of other parameters.

It is necessary for this robot model to follow the link-joint model because it is a well-understood and well-established concept within the robotics community from which much of its theory is based [9]. With the adoption of this model it is clear that the major components in the model are Link and Joint. These two main components will be defined and the necessary problems associated with each will be reviewed. As well, the possibility of modeling a Transmission component will be discussed.

Because of the strength of the model, Link and Joint have already been examined

extensively from many different viewpoints, resulting in a mature description of each of these components in just URDF or SDF. It is the author's opinion that these components are described well by these formats and there is not much to discover save for minute details. For these components a description is offered of how SDF and URDF accomplish their description.

6.4.1 Link

The LINK is an atomic rigid body in the robot's structure which constitutes the majority of the robot's composure by mass. A link is a primary parameter to the physics simulation as it contains the parameters necessary to simulate the physics of a robot. Additionally, a simulator typically has a visual representation of the robot to present to the user and a LINK will represent the majority of the visualization, so there will be parameters which are to be used by the simulation renderer. Finally, as the LINK is the only material component in the robot model, it will serve as the basis for the collision model, so the parameters to the collision engine must also be defined.

Physics

The PHYSICS component of the description of LINK defines the parameters needed for physics simulation. This will include the robot mass, rotational inertia matrix, and the inertial frame of reference. This portion of the model is consistent in frameworks which simulate the dynamics of robots. SDF introduces a linear and angular velocity damping in this component, however it was alone in this specification. As the implementation proceeds, it will be apparent if this component is missing any parameters.

Collision

The COLLISION component describes the parameters needed by the collision engine. Generally in most simulation description formats, the main components were the pose of the collision model and the simple geometry which defines it. This is to be differentiated from the visual geometry, as it is not necessarily true that a collision geometry may be used as a visual geometry or vice-versa. For instance, Webots uses an `IndexedFaceSet`

type to represent complex or high-fidelity geometries for the visual component of a robot. However this complex type of geometry would be a strain for the collision engine so it is not available for use as a collision geometry. A challenge in defining this component is deciding which collision geometries will effectively represent most robots.

Visual

The VISUAL component describes the parameters provided to the scene rendering engine. Visual representations are most easily characterized by some sort of geometry, which tended to be the observed pattern within most simulation frameworks. It should be noted that not all frameworks separate the collision model and the visual model. In this case, it usually marks that the simulator is simplistic and does not provide a high definition visual representation of the simulation. In these cases, one geometry is defined to serve as both the collision and visual geometry. In the context of our model, the collision geometry should serve as a possible dual model, as using the visual geometry for collision engines would be inefficient. Solving the collision geometry problem should also determine at least a subset of visual geometries. An additional challenge to defining the visual component will be handling mesh types. Between Webots and Gazebo, there are mesh types ranging from the standard triangle mesh to polygon meshes to height maps. As well, if meshes are supported they are best referenced externally to enable the readability of the description file.

6.4.2 Joint

The JOINT component of the model characterizes a set of constraints imposed on two joints. In this relationship, there is strictly one related parent LINK and one child LINK. There are six degrees of freedom between two unconstrained links, and a joint restrains one or many of them. The standard degrees of freedom are X, Y, Z, roll, pitch, and yaw. From the examined frameworks, three main types of joints reappear between frameworks, all of which JOINT generalizes:

- A **prismatic joint** constrains all degrees of freedom except for Z. This allows child link to translate freely in one dimension with respect to the parent link's frame. By

convention, this is normally the Z-axis in joint space. This type of joint might be used to model a gripper for a robot arm.

- A **hinge joint** constrains all degrees of freedom except for pitch. This allows a link to rotate in one dimension about a single point of rotation with respect to its parent link's frame. This type of joint might be used to model a fixed wheel on a differential drive robot if no bounds were posed on the joint rotation.

- A **ball-and-socket joint** constrains X, Y, and Z. This allows for rotation of a link about any of its axes with respect to its parent link. This type of joint might be used to model a spherical caster wheel for a differential drive robot.

6.4.3 Transmission

A Transmission component is not yet well-defined in the current state-of-the-art in robot modeling. Therefore it may contain some parameters which are to be discovered. From the previously discussed frameworks, URDF was the only description format which included this component, which the documentation notes was not intended to be used for anything but the PR2 robot.

Generally speaking, it is known that a transmission is supposed to define how a motor applies effort to a joint or several joints. Therefore, it can be concluded that a transmission must have an association with at least one joint.

Additionally, transmissions mark a point of control for the robot application. If an application wishes to command a robot to move in some manner, then the application must utilize the appropriate interfaces, which in turn triggers one or more robot control primitives which alter the state of the joints. Therefore, a primitive must somehow relate to some transmission and subsequently the parameters provided to the primitive. For example, it could be possible to translate a motor speed to the effort applied at that joint declaratively, and this might require a modification to the previously mentioned control model at the Primitive component.

With the major components of the model identified via a survey of existing frameworks, what remains is to discover which components are required to produce a full

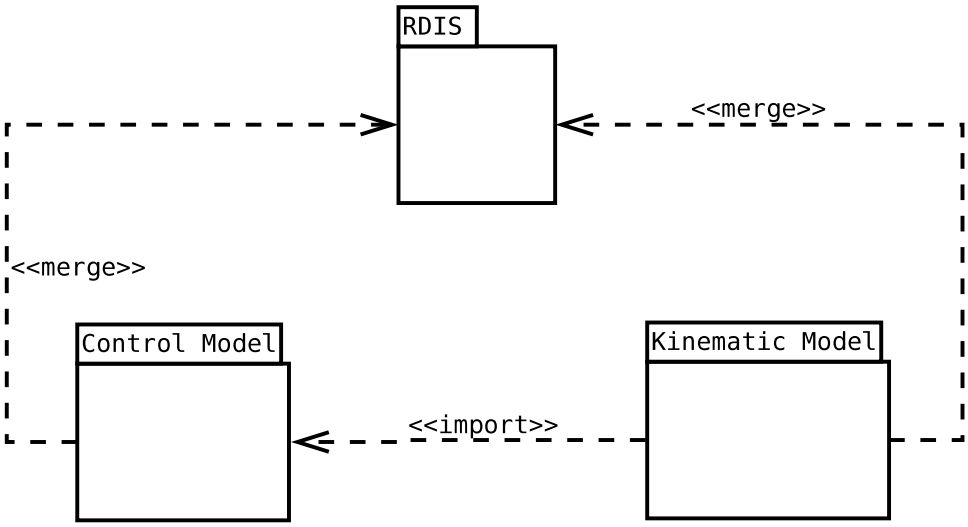

Figure 6.5: Package diagram showing the relationship between the two models.

implementation of the RDIS model. To that end, further research is required. At this point, the relationship between the kinematic model and the control model have been established as shown in Figure 6.5. The kinematic model will have no references to the control model, but the control model will need to be able to manipulate the kinematic model in order for simulation to be possible.

From the survey of literature and the above analysis a model can be inferred. The relationship of the above components is shown as a Class Diagram in Fig. 6.6. We see the same sort of link/joint model present in other solutions to this problem, whereas RDIS will need a component which maps its message models to a change in robot state in the kinematic model via use of the Transmission model element.

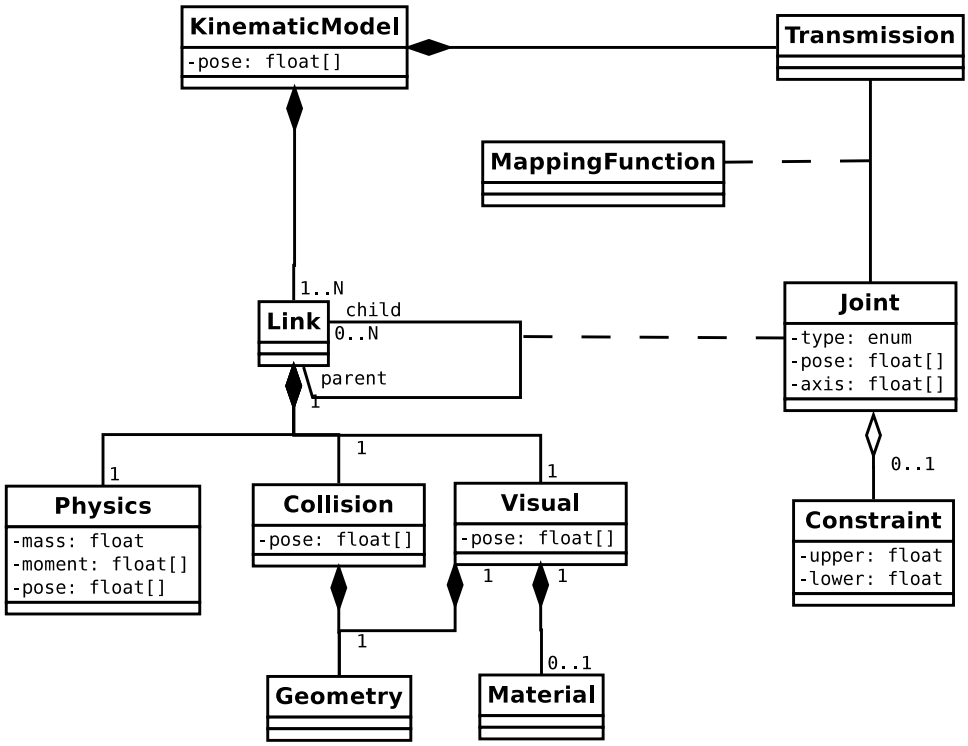

Figure 6.6: A Class Diagram showing a possible kinematic meta-model for RDIS.

CHAPTER 7
CONCLUSION

This chapter presents a summary of the contributions contained in this thesis. It presents the impact this model has on robotics software development and future work for RDIS.

7.1 Summary of Contributions

The previous chapters have detailed explicitly the contributions of this thesis. The principal contribution is the refinement of the RDIS domain model. A new textual syntax was developed and an interpreter was written to interpret it. To solve some of the development issues of the previous implementation, a tool which can generate these textual RDIS descriptions was developed, utilizing the presented domain model. A future goal of RDIS is the support of kinematic modeling, so a literature review and an analysis of frameworks was conducted as a part of the contribution.

7.1.1 Domain Model

Initially, RDIS was a proof-of-concept implementation. It was a set of grammars for use with the ANTLR parser generator and would use templates to generate drivers for various frameworks. The core components of the RDIS model were defined as well as some of the relationships, but there was much need for some of the specifics of the model to be defined.

As a part of the contribution of this thesis, the RDIS model was refined at all levels from general component architecture down to the attributes. Documentation has been written [20]. The textual syntax has been revised and an interpreter has been written so that it may be verified.

7.1.2 An Example Model-Driven Implementation

RDIS is defined precisely enough that it may be modeled in a host of different modeling tools. To demonstrate this, an example implementation in the AToM^3 tool has been implemented. The tool shows how RDIS textual descriptions may be generated automatically by models rather than crafted by hand. It demonstrates the benefit of using modeling tools to create RDIS models by enabling use of model transformations to define the mappings between RDIS and ROS. We gain the added usability of working in a graphical rather than textual environment, and the ability to represent more complex non-serialized relationships between entities. The model editor also adds a degree of organization since model elements may be spatially organized.

7.1.3 A Design for Kinematic Modeling

A future goal of RDIS is to extend into simulation frameworks as well, so a model for robot kinematics, visuals, and collision logic must also be defined. Some initial work has been done in this regard with a survey of the current landscape of robot hardware description and an analysis of the current approaches used by various frameworks. From this, a model is derived and presented. This work may be used as a design framework for kinematics modeling in RDIS.

7.2 Future Work

RDIS currently defines enough to drive a robot and read sensors. However, more functionality is usually available for an application developer to use. Each of the following sections summarizes future significant research projects for RDIS.

7.2.1 Framework Modeling

As a part of the example application given in this thesis, parts of the ROS software framework was modeled. However, the solution was tailored specifically for ROS and without consideration for other frameworks. Future research might explore the generalized modeling of software frameworks perhaps in the construction of framework adapters.

7.2.2 Threading

Threading is a rather unexplored topic in RDIS. Though Single, Dual, and Multiple threading models are noted, a survey of the threading models used in software frameworks has not been conducted. Future research might analyze the use of threading in robot frameworks and propose an abstraction to these threading models. A starting point for implementation might be to implement per-Connection threads for Single threading.

7.2.3 Sensor and Actuator Error Modeling

Sensor and actuator error modeling is a large component of the robotics application. RDIS currently does not have a sophisticated notion of state, so it is unable to support this in its current form. It is likely that sensor modeling and kinematic modeling should be completed first. An example project might examine how one could attach a Kalman filter [62] to a Sensor model element, how one appropriately parametrizes a Kalman filter, and the subsequent implementation. A thorough analysis of how modern frameworks handle error modeling should be conducted.

7.2.4 Sensor Modeling

Sensor modeling is one step the project could take to improve the RDIS representation of state. For example, in the example application, the State Variable is only used to hold the last known value of a sensor. As well, Interfaces are highly abstract – there is no real world concretion that necessarily has to correspond to an Interface. Therefore, some Interfaces might be replaced with a Sensor model element. This abstraction (like Interfaces) might call a Primitive and cache the last read value for some fixed amount of time. This could eliminate one use of a State Variable, and cut down on the amount of data requested from the robot. The approach used by modern frameworks should be taken into account as well.

7.2.5 Framework-RDIS Interface

One of the primary challenges of RDIS is to define the standard interface between the device and framework. In the context of the RDIS model, this is the Framework-RDIS interface, and includes the Domain Interfaces, Domain Outputs, and Domain Adapters.

A first challenge would be to develop standardized names and types for Domain Outputs and Domain Interfaces to solve some naming issues. This would require more exploration of the internal messages used by robot software frameworks.

7.2.6 Kinematic Modeling

Lastly, kinematic modeling would serve two purposes. The first is the robot's own knowledge of its state. This might involve the invention of an Actuator model element (similar to the Sensor of the previous section) which has some sort of mapping function that affects Links and Joints in the kinematic model. This would replace a second possible use for State Variables. A second purpose would be as a simulation description for simulation frameworks. Each simulator requires its own simulation description format, so RDIS also might focus on how to successfully generalize these descriptions. An approach for this has been proposed in this thesis.

7.3 Impact

The impact of this research is that it is one of very few projects that seeks to simplify software development for robotics by the use of software models. It is the only work in the author's knowledge which focuses specifically on defining the mappings between software frameworks and device messages. In this application, there is no formal process for developing robot drivers, so at the most basic level it advances knowledge of this by defining the components involved and the relationships between them.

If the RDIS model becomes very viable and standardized, then it may become possible to embed RDIS descriptions within the robots themselves so that a robot may describe its own communication protocol. This will allow for a true plug-and-play robotic system. Currently there is not such a system which is generalizable to a large category of robots.

Part of the RDIS project is defining the standard messages which can pass from the robotics software framework an into the device. The pathways for these messages are one subject on their own, but the existence of such standard messages and interfaces would greatly benefit the robotics community as no such standard exists. Since no standard exists, this greatly diminishes the possibility of code reuse. By standardizing these inter-

faces, a framework may be able to communicate with any arbitrary device so long as the appropriate interfaces exist.

REFERENCES

[1] The Jython Project. *http://www.jython.org/*, retrieved 6/3/2012, May 2012.

[2] Dave Abrahams. Boost.python. *http://www.boost.org/doc/libs/1_49_0/libs/python/doc*, retrieved 6/3/2012, August 2003.

[3] M. Anderson, P. Kilgo, and J. Bowman. RDIS: Generalizing domain concepts to specify device to framework mappings. In *International Conference on Robotics and Automation*, pages 1836 – 1841, St. Paul, Minnesota, USA, May 2012.

[4] M. Anderson, P. Kilgo, C. Crawford, and M. Stanforth. Work in progress: Enabling robot device discovery through robot device descriptions. In *2nd International Workshop on Domain-Specific Languages and models for ROBotic systems*, San Francisco, California, USA, September 2011.

[5] J.-C. Baillie. URBI: towards a universal robotic low-level programming language. In *IEEE International Conference on Robotics and Automation*, pages 820–825, Edmonton, Alberta, Canada, August 2007.

[6] Mirko Bordignon, Ulrik Pagh Schultz, and Kasper Stoy. Model-based kinematics generation for modular mechatronic toolkits. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pages 157–166, Eindhoven, The Netherlands, 2010.

[7] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. USARSim: a robot simulator for research and education. In *IEEE International Conference on Robotics and Automation*, pages 1400–1405, Roma, Italy, April 2007.

[8] G. Costagliola, A. Delucia, S. Orefice, and G. Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, 13(6):573 – 600, 2002.

[9] John J. Craig. Manipulator kinematics. In *Introduction To Robotics: Mechanics and Control, 2nd Ed.*, pages 69–112. Addison-Wesley Longman, 1955.

[10] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[11] Juan de Lara. AToM^3: A tool for multi-formalism meta-modeling. http://atom3.cs.mcgill.ca/, January 2008.

[12] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca Moreno. Using metamodelling and graph grammars to create modelling environments. In *Electronic Notes In Theoretical Computer Science*, volume 72 (3), Barcelona, Spain, 2003. Elsevier.

[13] Eclipse Foundation. Eclipse modeling - EMF - home. http://www.eclipse.org/emf/, April 2009.

[14] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, Taipei, Taiwan, September 2003.

[15] David Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, May 1988.

[16] Philip Hazel. Pcre - perl-compatible regular expressions. Technical report, University of Cambridge, January 2012. *http://www.pcre.org/pcre.txt*, retrieved 6/3/2012.

[17] David Joiner. PEP 3101 – advanced string formatting. Technical report, Python Software Foundation, April 2006. *http://www.python.org/dev/peps/pep-3101/*, retrieved 6/2/2012.

[18] S. Kelly and J.-P. Tolvanen. Visual domain-specific modelling: benefits and experiences of using metaCASE tools. In *Proceedings of the International Workshop on Model Engineering*, pages 1–9, 2000.

[19] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, New Jersey, USA, 2008.

[20] Paul Kilgo. Robot device interface specification. *http://ua-robotics.net/index.php?title=RDIS*, retrieved, 6/11/2012, June 2012.

[21] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, September 2004.

[22] A. Krishna, D.C Schmidt, and M. Stal. Context object: A design pattern for efficient middleware request processing. In *13th Pattern Language of Programs Conference*, Monticello, Illinois, USA, 2005.

[23] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5:369–385, 2006.

[24] Christian Lange. Model size matters. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 211–216. Springer Berlin / Heidelberg, 2007.

[25] Juan Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin / Heidelberg, 2002.

[26] Jonas Buchli Marco Frigerio and Darwin G. Caldwell. A domain specific language for kinematic models and fast implementations of robot dynamics algorithms. In *2nd International Workshop on Domain-Specific Languages and models for ROBotic systems*, pages 1–6, San Francisco, California, USA, September 2011.

[27] O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

[28] P. Mohagheghi and V. Dehlen. Existing model metrics and relations to model quality. In *ICSE Workshop on Software Quality, 2009. WOSQ '09*, pages 39 –45, Vancouver, Canada, May 2009.

[29] Pierre-Alain Muller, Frédéric Fondement, Benoît Baudry, and Benoît Combemale. Modeling modeling modeling. *Software and Systems Modeling*, pages 1–13.

[30] Model-driven architecture (MDA) guide version 1.0.1. Technical report, Object Management Group, June 2003. *http://www.omg.org/cgi-bin/doc?omg/03-06-01*, retrieved 6/27/2012.

[31] Unified modeling language (UML) 2.0 core specification. Technical report, Object Management Group, July 2005. *http://www.omg.org/spec/UML/2.0/*, retrieved 6/27/2012.

[32] Object constraint language (OCL), version 2.0. Technical report, Object Management Group, May 2006. *http://www.omg.org/spec/OCL/2.0/*, retrieved 6/27/2012.

[33] Robotic Technology Component (RTC) 1.0. Technical report, Object Management Group, April 2008. *http://www.omg.org/spec/RTC/1.0/*, retrieved 6/27/2012.

[34] Super Distributed Object (SDO) 1.1. Technical report, Object Management Group, October 2008. *http://www.omg.org/spec/SDO/1.1/*, retrieved 6/27/2012.

[35] Robot Localization Service (RLS) 1.0. Technical report, Object Management Group, February 2010. *http://www.omg.org/spec/RLS/1.0/*, retrieved 6/27/2012.

[36] Robotics DTF, December 2011. *http://robotics.omg.org/*, retrieved 2/2/2012.

[37] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[38] Terence John Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA, 2004.

[39] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, New Jersey, USA, 1981.

[40] Stage model, December 2011. *http://rtv.github.com/Stage/group__model.html*, retrieved 2/2/2012.

[41] Python Software Foundation. `json` – JSON encoder and decoder. Technical report, Python Software Foundation, June 2012. *http://docs.python.org/library/json.html*, retrieved 6/5/2012.

[42] Python Software Foundation. `struct` – interpret strings as packed binary data. Technical report, Python Software Foundation, June 2012. *http://docs.python.org/library/struct.html*, retrieved 6/2/2012.

[43] 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 *Proc. Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 2009.

[44] Indranil Saha and Natarajan Shankar. ModelRob: A simulink library for model-based development of robot manipulators. In *2012 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2654–2659, St. Paul, Minnesota, United States, May 2012.

[45] C. Schlegel, T. Hassler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. In *International Conference on Advanced Robotics*, pages 1–8, Munich, Germany, June 2009.

[46] Christian Schlegel. Smartsoft, June 2011. *http://smart-robotics.sourceforge.net/*, retrieved 2/2/2012.

[47] U. Schultz, D. Christensen, and K. Stoy. A domain-specific language for programming self-reconfigurable robots. In *Automatic Program Generation for Embedded Systems – Workshop Proceedings*, October 2007. *http://www-fp.cs.st-andrews.ac.uk/APGES/OnlineProceedings/21-schultz-al.pdf*, retrieved 6/27/2012.

[48] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19 – 25, September 2003.

[49] R. Smith. Open dynamics engine v0.5 user guide. Technical report, Open Dynamics Engine Project, February 2006. *http://ode.org/ode-latest-userguide.pdf*, retrieved 2/3/2012.

[50] Jonathan Sprinkle. Analysis of a metamodel to estimate complexity of using a domain-specific language. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 13:1–13:6, Reno, Nevada, 2010.

[51] Andreas Steck and Christian Schlegel. Towards quality of service and resource aware robotic systems through model-driven software development. In *1st International Workshop on Domain-Specific Languages and models for ROBotic systems*, pages 1–6, Taipei, Taiwan, October 2010.

[52] D. Steinberg, F. Budinsky, M. Pternostro, and E. Merks. Introducing EMF. In *EMF: Eclipse Modeling Framework*, pages 9–38. Addison Wesley Professional, 2008.

[53] S. Streeting. OGRE manual v1.7 ('cthugha'). Technical report, The OGRE Team, October 2010.

[54] Peter Struss and Chris Price. Model-based systems in the automotive industry. *AI Magazine*, 24:17–34, December 2003.

[55] Eugene Syriani. What is model transformation? In *A Multi-Paradigm Foundation for Model Transformation Language Engineering*, chapter 1. McGill University, 2012. *http://cs.ua.edu/ syriani/teaching/docs/MTSurvey.Syriani.pdf*, retrieved 6/27/2012.

[56] Webots Team. Webots user guide, release 6.4.3. Technical report, Cyberbotics Ltd., January 2012. *http://www.cyberbotics.com/guide.pdf*, retrieved 2/4/2012.

[57] Steve Tousignant, Eric Van Wyk, and Maria Gini. Xrobots: A flexible language for programming mobile robots based on hierarchical state machines. In *2012 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1773–1778, St. Paul, Minnesota, United States, May 2012.

[58] Piotr Trojanek. Model-driven engineering approach to design and implementation of robot control system. In *2nd International Workshop on Domain-Specific Languages and models for ROBotic systems*, San Francisco, California, USA, 2011.

[59] H. Vangheluwe and J. de Lara. Computer automated multi-paradigm modelling for analysis and design of traffic networks. In *Simulation Conference, 2004. Proceedings of the 2004 Winter*, volume 1, pages 2 vol. (xliv+2164), Washington DC, USA, December 2004.

[60] Richard Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2:189–208, 2008.

[61] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley and Sons, Ltd., New Jersey, USA, 2006.

[62] Greg Welch and Gary Bishop. An introduction to the kalman filter, 1997. *http://clubs.ens-cachan.fr/krobot/old/data/positionnement/kalman.pdf*, retrieved 6/27/2012.

[63] B. L. Wellman, M. Anderson, and S. Vrbsky. PREOP as a tool to increase student retention in CS. *Journal of Computing Sciences in Colleges*, 25(2):167–175, December 2009.

[64] Xml robot description format (urdf). *http://www.ros.org/wiki/urdf/XML*, retrieved, 2/3/2012, June 2010.

[65] Simulation description format. *http://gazebosim.org/wiki/sdf_format*, retrieved, 2/3/2012, October 2011.

[66] Yali Wu, Frank Hernandez, Francisco Ortega, Peter J. Clarke, and Robert France. Measuring the effort for creating and using domain-specific models. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 14:1–14:6, Reno, Nevada, 2010.

APPENDIX A

THE RDIS INTERPRETER

```
#!/usr/bin/env python

"""
An interpreter for RDIS models and quick-n-dirty parser for the JSON concrete
syntax.

This contains several methods which are useful to other applications which are
associated somehow with RDIS and need to emulate some of its behavior.

Usage:
  import rdis
  model = rdis.load( "/path/to/my/model.rdis.json" )

  model.startup()

  while <stuff-to-do>:
    model.tick()

    if <some-event>:
      model.callDomainInterface("interface-name", ["arg1", "arg2"])

  model.terminate()
"""

import struct     ## Used to pack bytes to send over the connection
import re         ## Used for models which reply with ASCII-encoded data.
```

```
27 import sys        ## Used for printing error messages.
28 import json       ## Used to parse RDIS textual syntax.
29 import time       ## Used for scheduling.
30 import random     ## Used for dummy methods.
31 import string     ## Used for checking printable characters.
32 import serial     ## Used for serial port connections.
33
34 ## Dictionary which maps RDIS types to Python types.
35 _gTypeMap = {
36   'int': 'int',
37   'float': 'float',
38   'string': 'str'
39 }
40
41 ## Dictionary which maps RDIS types to their equivalent
42 ## default values in Python.
43 _gDefaultValueMap = {
44   'int': 0,
45   'float': 0.0,
46   'string': ''
47 }
48
49 def mapDefaultValue(typeName):
50   """
51   Maps an RDIS type name to Python default value.
52   Raises TypeError if typeName is not a valid RDIS type.
53   """
54   global _gDefaultValueMap
55
56   mapType(typeName)
57   return _gDefaultValueMap[typeName]
58
59 def mapType(typeName):
60   """
61   Maps an RDIS type to a Python type.
```

```
62    Raises TypeError if typeName is not a valid RDIS type.
63    """
64    global _gTypeMap
65    if typeName not in _gTypeMap:
66      raise TypeError("Unknown RDIS type: " + str(typeName))
67    return _gTypeMap[typeName]
68
69  def createEnvironment(parameterNames, values):
70    """
71    Given two parallel lists, one of names and one of values, create a
72    dictionary with the corresponding name/value pairs.
73    """
74    env = dict()
75    i = 0
76    for parameter in parameterNames:
77      env[parameter] = values[i]
78      i += 1
79    return env
80
81  def deleteAdditions(before, after):
82    """
83    Deletes any key which is in after, but is not in before.
84    """
85    for key in after.viewkeys() - before.viewkeys():
86      del after[key]
87
88  def safeEval(expression, env):
89    """
90    If expression is not a string, it returns expression.
91    Otherwise, if the first and last characters of the string
92    are '<' and '>', it chops them off and evaluates the resulting
93    expression under the provided environment.
94    """
95    if not isinstance(expression, basestring) or len(expression) == 0:
96      return expression
```

```
  returnValue = expression

  oldEnv = dict(env)
  if expression[0] == '<' and expression[-1] == '>':
    returnValue = eval( expression[1:-1], env )

  ## Subtract off any keys added by eval
  deleteAdditions(oldEnv, env)

  return returnValue

def safeEvalAll(expressions, env):
  """
  Evaluates each expression contained in expression and returns the
     list.
  """
  return [safeEval(k,env) for k in expressions]

def safeExecs(stmts, globalEnv=None):
  """
  Calls safeExec on a sequence of statements.
  """
  oldGlobals = dict(globalEnv)

  for stmt in stmts:
    exec(stmt, globalEnv)

  deleteAdditions(oldGlobals, globalEnv)

def safeType( value ):
  """
  Gets a string representation of value's type.
  Safe for objects and primitive types.
  """
```

```
131   varType = type(value).__name__
132   if varType == "instance":
133     return value.__class__.__name__
134
135   return varType
136
137 def printBytes(byteSequence):
138   """
139   Prints a byte sequence to stdout.
140
141   If a character is printable, it will print the ASCII representation
        of that
142   byte as well. Otherwise, it only prints the hexadecimal represenation
        .
143   """
144   for b in byteSequence:
145     if b in string.printable: print hex(ord(b)), b
146     else: print hex(ord(b))
147
148 def load(rdisFile):
149   """
150   Builds RDIS model from its textual description.
151
152   @raises RDISExcption if the model description is invalid.
153   """
154   fp = open(rdisFile)
155   ctx = json.load(fp)
156   fp.close()
157
158   try:
159     return RDIS(contextObject=ctx)
160   except KeyError as e:
161     raise RDISException("Invalid RDIS description given: " + str(e))
162
163 class RDISException(Exception):
```

```
  """
  Class for RDIS-related exceptions, generally problems with the model itself.
  This will be implemented more fully one day.
  """
  pass

class RDIS:
  """
  The public class for RDIS models.

  Generally this should be constructed by calling rdis.load(), though public
  methods exist for incrementally building a model by hand. 4 in 5 computer
  scientists agree that this is generally a bad thing to do though.
  """

  def __init__(self, contextObject=None):
    """
    Makes an empty RDIS object.
    """
    self._primitives = dict()
    self._connections = dict()
    self._interfaces = dict()
    self._stateVariables = dict()
    self._domainInterfaces = dict()
    self._domainOutputs = dict()
    self._callback = None
    self._name = None
    self._author = None

    if contextObject != None:
      self._fromContext(contextObject)

```

```
196   def setCallback(self, callback):
197     """
198     Sets the callback to use when a domain output is triggered.
199     """
200     self._callback = callback
201
202   def callDomainInterface(self, name, env):
203     """
204     Calls a domain interface by its name.
205
206     The dictionary "env" corresponds to the parameters that the domain
207     interface takes.
208     """
209     domainInterface = self._getDomainInterface(name)
210     if domainInterface == None:
211       sys.stderr.write("'{}': called nonexistant domain interface\n".
          format(
212          name
213        )
214       )
215       return
216     domainInterface.call(env)
217
218   def startup(self, **kwargs):
219     """
220     Performs initialization tasks for the model.
221
222     This should be called after the model is constructed, but before
          any
223     real work is done as it initializes the connection to the robot and
          things
224     like that.
225     """
226     for conn in self._connections.values():
227       apply(conn.startup, [], kwargs)
```

```
228
229   def terminate(self):
230     """
231     Performs termination tasks for the model.
232
233     This should be called when all work is done for the model and it is
          not
234     needed anymore. This will do things such as putting the robot into
          a safe
235     state and releasing any resource locks.
236     """
237     for conn in self._connections.values():
238       conn.terminate()
239
240   def tick(self):
241     """
242     Notifies the model that a single cycle for the main thread of
          execution
243     should be performed.
244
245     This does not necessarily correspond to the tick rate of the
          connection,
246     but this should be called at a frequency faster than that of any
          component
247     in the model.
248
249     TODO: Actually there is no scheduling in the model right now. This
          should
250     be done real soon.
251     """
252     for conn in self._connections.values():
253       conn.tick()
254
255
256   def setName(self, name):
```

```
257     """
258     Sets the (robot) name of the model.
259     """
260     self._name = name
261
262   def getName(self):
263     """
264     Gets the (robot) name of the model.
265     """
266     return self._name
267
268   def getAuthor(self):
269     """
270     Gets the author of the model.
271     """
272     return self._author
273
274   def getConnection(self, name):
275     """
276     Gets a connection by its name.
277     """
278     return self._connections[name]
279
280   def addInterface(self, name, **kwargs):
281     """
282     Adds a local interface to the model.
283     """
284     return apply(self._addElement, [name, self._interfaces, Interface],
            kwargs)
285
286   def addPrimitive(self, name, **kwargs):
287     """
288     Adds a primitive tot he model.
289     """
290     return apply(self._addElement, [name, self._primitives, Primitive],
```

```
        kwargs)

  def addSerialConnection(self, name, **kwargs):
    """
    Adds a serial connection to the model.
    """
    return apply(self._addElement, [name, self._connections,
        SerialConnection],
        kwargs)

  def addStateVar(self, name, **kwargs):
    """
    Adds a state variable to the model.
    """
    return apply(self._addElement, [name, self._stateVariables,
        StateVar],
        kwargs)

  def addDomainInterface(self, name, **kwargs):
    """
    Adds a domain interface to the model.
    """
    return apply(self._addElement, [name, self._domainInterfaces,
      DomainInterface], kwargs)

  def addDomainOutput(self, name, **kwargs):
    """
    Adds a domain output to the model.
    """
    return apply(self._addElement, [name, self._domainOutputs,
      DomainOutput], kwargs)

  def _receiveMessage(self, who, msg):
    """
    Receives a messages from one of the lower-level components to send
```

```
        to the
    callback.
    """
    if self._callback == None:
      sys.stderr.write("Received message but no callback assigned!\n")
      return
    self._callback({'name': who.getName(), 'contents': msg})

  def _addElement(self, name, store, classObj, **kwargs):
    """
    Generalized method for adding a model component.
    """
    kwargs["name"] = name
    kwargs["parent"] = self
    store[name] = apply(classObj, [], kwargs)

  def getStateVarValue(self, name):
    """
    Gets the (Python) value for a state variable.
    """
    return self._stateVariables[name].getValue()

  def _getDomainInterface(self, name):
    """
    Gets a domain interface by its name.
    """
    return self._domainInterfaces.get(name)

  def _getInterface(self, name):
    """
    Gets a local interface by its name.
    """
    return self._interfaces.get(name)

  def _getPrimitive(self, name):
```

```
    """
    Gets a primitive by its name.
    """
    return self._primitives.get(name)

  def _getDomainOutput(self, name):
    """
    Gets a domain output by its name.
    """
    return self._domainOutputs.get(name)

  def _callDomainOutput(self, name):
    """
    Calls a domain output.
    """
    do = self._getDomainOutput(name)
    if do == None:
      sys.stderr.write("'{}': nonexistent domain output\n".format(name)
          )
      return
    do.call()

  def _callPrimitive(self, name, posArgs):
    """
    Calls a primitive with given positional arguments.
    """
    p = self._getPrimitive(name)
    if p == None:
      sys.stderr.write("'{}': called nonexistent primitive\n".format(
          name))
      return
    elif p.parameterCount() != len(posArgs):
      sys.stderr.write(
        "Arg mismatch for primitive '{}'. Got {}, expected {}\n".format
            (
```

```
389            name, len(posArgs), p.parameterCount()
390          )
391        )
392        return
393      p.call(posArgs)
394
395    def _callInterface(self, name, posArgs):
396      """
397      Calls an interface (with provided name) with given positional
         arguments.
398      """
399      i = self._getInterface(name)
400      if i == None:
401        sys.stderr.write("'{}': called nonexistent local interface\n".
           format(
402            name
403          )
404        )
405        return
406      i.call(posArgs)
407
408    def _buildStateVarEnvironment(self):
409      """
410      Builds a dictionary containing the names and values of the state
411      variables.
412      """
413      env = dict()
414      for (name, stateVar) in self._stateVariables.items():
415        env[name] = stateVar.getValue()
416      return env
417
418    def _updateStateVarsFromEnvironment(self, env):
419      """
420      Updates state variable values who share names with the keys of "env
         ".
```

```
421     """
422     for (name, value) in env.items():
423       stateVar = self._stateVariables[name]
424       stateVar.setValue(value)
425
426   def _fromContext(self, ctx):
427     """
428     Builds RDIS model from the given context object.
429     """
430     for key in ctx.keys():
431       if key == "domainInterfaces":
432         self._loadDomainInterfaces(ctx[key])
433       elif key == "interfaces":
434         self._loadInterfaces(ctx[key])
435       elif key == 'primitives':
436         self._loadPrimitives(ctx[key])
437       elif key == 'connections':
438         self._loadConnections(ctx[key])
439       elif key == 'domainOutputs':
440         self._loadDomainOutputs(ctx[key])
441       elif key == 'stateVariables':
442         self._loadStateVar(ctx[key])
443       elif key == 'author':
444         self._author = ctx[key]
445       elif key == 'name':
446         self._name = ctx[key]
447       else:
448         sys.stderr.write("Unknown element type: {}\n".format(key))
449
450   def _loadStateVar(self, varList):
451     """
452     Loads state variables from the context object representation.
453     """
454     for var in varList:
455       apply(self.addStateVar,
```

```
456            [var['name']], {
457            'type': var['type'],
458            'value': var['value']
459            }
460        )
461
462    def _loadDomainOutputs(self, doList):
463      """
464      Loads domain outputs from the context object representation.
465      """
466      for do in doList:
467        apply(self.addDomainOutput,
468            [do['name']], {
469            'returns': do['returns']
470            }
471        )
472
473    def _loadDomainInterfaces(self, diList):
474      """
475      Loads domain interfaces from the context object representation.
476      """
477      for di in diList:
478        apply(self.addDomainInterface,
479            [di["name"]], {
480            'parameters':di["parameters"],
481            'localInterface':di["calls"]["name"],
482            'interfaceArguments':di["calls"]["arguments"]
483          }
484        )
485
486    def _loadInterfaces(self, interfaceList):
487      """
488      Loads local interfaces from the context object representation.
489      """
490      for interface in interfaceList:
```

```
491        apply(self.addInterface,
492          [interface['name']], {
493            'type': interface['type'],
494            'parameters': interface.get('parameters'),
495            'triggers': interface.get('triggers')
496          }
497        )
498        i = self._getInterface(interface['name'])
499
500        for pc in interface['primitiveCalls']:
501          i.addPrimitiveCall(
502            pc['name'],
503            pc['arguments'],
504            pc['delay'],
505            pc['priority']
506          )
507
508    def _loadPrimitives(self, primitiveList):
509      """
510      Loads primitives from the context object representation.
511      """
512      for p in primitiveList:
513        apply(self.addPrimitive,
514          [p['name']], {
515            'connection': p['connection'],
516            'parameters': p.get('parameters', []),
517            'formatArgs': p.get('formatArgs', []),
518            'postActions': p.get('postActions', []),
519            'format': p.get('format'),
520            'regex': p.get('regex'),
521            'pack': p.get('pack'),
522            'unpack': p.get('unpack')
523          }
524        )
525
```

```
526   def _loadConnections(self, connections):
527     """
528     Loads connections from the context object representation.
529     """
530     self._loadSerialConnections(connections.get('spp'))
531
532   def _loadSerialConnections(self, connectionList):
533     """
534     Loads serial connections from the context object representation.
535     """
536     if connectionList == None: return
537     for c in connectionList:
538       apply(self.addSerialConnection,
539         [c['name']], {
540           'baud': c['baud'],
541           'singleThreading': c.get('singleThreading'),
542         })
543       conn = self.getConnection(c['name'])
544       k = c.get('keepalive')
545       t = c.get('terminate')
546       s = c.get('startup')
547       if k: apply(conn.setKeepalive, [], k)
548       if t: apply(conn.setTerminate, [], t)
549       if s: apply(conn.setStartup, [], s)
550
551 class StateVar:
552
553   """
554   A basic unit of memory for the robot model.
555
556   Only Primitives can manipulate state variables and they generally
        reflect a
557   value read from the sensor, or some sort of constant about the robot.
558   """
559
```

```
560   def __init__(self, name=None, value=None, type=None, parent=None):
561     """
562     Constructs a state variable.
563     """
564     if value == None:
565       value = mapDefaultValue(type)
566     self._name = name
567     self._parent = parent
568
569     self._setType(type)
570     self.setValue(safeEval(value,dict()))
571
572   def getValue(self):
573     """
574     Gets the value of the state variable.
575     """
576     return self._value
577
578   def _getDefaultValue(self):
579     """
580     Gets the default value for this state variable.
581     """
582     return mapDefaultValue(self._type)
583
584   def setValue(self,value):
585     """
586     Sets the value of the state variable.
587     """
588     if value == None or value == "":
589       value = self._getDefaultValue()
590
591     self._assertType(value)
592     self._value = value
593
594   def _assertType(self, value):
```

```
    """
    Asserts value is a valid type for this state variable.

    @raises TypeError if the assertion failed.
    """
    nameType = self._getPythonType()
    valType = safeType(value)
    if not isinstance(value, eval(nameType)):
      raise TypeError(
        "Attempt to set {} ({}) to value {} ({}).".format(self._name, nameType,
        repr(value), valType)
      )

  def _getPythonType(self):
    """
    Gets the equivalent Python type for this state variable.
    """
    return mapType(self._type)

  def _setType(self, newType):
    """
    Changes this StateVar's RDIS type.
    """
    mapType(newType) ## Asserts that it is a valid RDIS type.
    self._type = newType

class Primitive:

  def __init__(self, parent=None, pack=None, unpack=None, parameters=[],
      regex=None, format=None, name=None, connection=None, formatArgs=[],
      postActions=[]):
    """
```

```
    Constructs a Primitive.
    """
    self._parent = parent
    self._format = format
    self._name = name
    self._pack = pack
    self._unpack = unpack
    self._regex = regex
    self._connection = connection
    self._parameters = parameters
    self._formatArgs = formatArgs
    self._postActions = postActions

  def call(self, arguments=[]):
    """
    Calls a primitive with the provided positional arguments.
    """
    env = createEnvironment(self._parameters, arguments)

    ## Evaluate all the args we send out on the format string.
    values = [safeEval(k,env) for k in self._formatArgs]

    ## Write the information on the connection and read back.
    connection = self._getConnection()
    connection.write( self._doPack(values) )
    out = self._doUnpack( connection )

    ## Build environments for postActions and execute them.
    envs = self._buildPostActionEnv(env, out)
    safeExecs(self._postActions, envs[1])

    ## Delete local variables that we added.
    deleteAdditions(envs[0], envs[1])

    ## Updates stateVar from the global environment.
```

```
662     self._parent._updateStateVarsFromEnvironment(envs[1])
663
664   def parameterCount(self):
665     """
666     Returns the number of parameters this Primitive accepts.
667     """
668     return len(self._parameters)
669
670   def _buildPostActionEnv(self, parameters, out):
671     """
672     Builds environment for post actions.
673     """
674     globalEnv = self._parent._buildStateVarEnvironment()
675     globalCopy = dict(globalEnv)
676     globalCopy.update(parameters)
677     globalCopy["__out__"] = out
678     return (globalEnv, globalCopy)
679
680   def _getConnection(self):
681     """
682     Gets connection associated with this Primitive.
683     """
684     return self._parent.getConnection(self._connection)
685
686   def _doPack(self, values):
687     """
688     Decides whether or not we need to ASCII-encode or byte-pack the
          values we
689     are writing to the robot.
690     """
691     if self._format != None:
692       return bytes(apply(self._format.format, values))
693     else:
694       tmp = [str(self._pack)]
695       tmp.extend(values)
```

```
696         return apply(struct.pack, tmp)
697
698     def _doUnpack(self, connection):
699       """
700       Unpacks arguments read from the connection using regular
            expressions or
701       struct.unpack().
702       """
703       if self._regex != None:
704         return re.match(self._regex, str(connection.read())).groups()
705       elif self._unpack != None:
706         sz = struct.calcsize(str(self._unpack))
707         byteSequence = connection.read(sz)
708         return struct.unpack(str(self._unpack), byteSequence)
709
710
711 class Connection:
712
713     """
714     Represents the connection from the host application to the robot.
715     """
716
717     def __init__(self, name, parent):
718       """
719       Constructs a connection.
720       """
721 ## TODO: THREADING
722       self._name = name
723       self._parent = parent
724       self._startup = None
725       self._keepalive = None
726       self._terminate = None
727       self._initialized = False
728
729     def startup(self):
```

```
730     """
731     Initializes the connection if it is unintialized.
732     """
733     if self._initialized == True: return
734     self._initialized = True
735     pass
736
737   def tick(self):
738     """
739     Performs any periodic actions needed by this connection.
740     """
741     self.keepalive()
742
743     ## TODO: Periodic interfaces.
744
745   def keepalive(self):
746     """
747     Attempts to call the keepalive if it is past due.
748     """
749     ## TODO: obey keepalive frequency
750     self._callKeepalive()
751
752   def _callKeepalive(self):
753     """
754     Calls the keepalive interface.
755     """
756     self._callEventInterface(self._keepalive)
757
758   def _callStartup(self):
759     """
760     Calls the startup interface.
761     """
762     self._callEventInterface(self._startup)
763
764   def _callTerminate(self):
```

```
765     """
766     Calls the termination interface.
767     """
768     print "Terminating..."
769     self._callEventInterface(self._terminate)
770
771   def _callEventInterface(self, i):
772     """
773     Calls a generic eventful interface.
774     """
775     if i == None: return
776     env = self._parent._buildStateVarEnvironment()
777     args = safeEvalAll(i['arguments'], env)
778     self._parent._callInterface(i['name'], args)
779
780   def setStartup(self, name, arguments=[]):
781     """
782     Sets the startup interface.
783     """
784     self._startup = {
785       'name': name,
786       'arguments': arguments
787       ## TODO: Where is the periodicity?
788     }
789
790   def setTerminate(self, name, arguments=[]):
791     """
792     Sets the terminate interface.
793     """
794     self._terminate = {
795       'name': name,
796       'arguments': arguments
797     }
798
799   def setKeepalive(self, name, interval, arguments=[]):
```

```
800     """
801     Sets the keepalive interface.
802     """
803     self._keepalive = {
804       'interval': interval,
805       'name': name,
806       'arguments': arguments
807     }
808
809
810   def write(self, stuff):
811     """
812     Dummy method for writing to an abstract connection. Prints bytes to
813     stdout.
814     """
815     print "--- WROTE ---"
816     printBytes(stuff)
817
818   def read(self, num):
819     """
820     Dummy method for reading bytes. Returns arbitrary bytes.
821     """
822     print "--- READ ---"
823     byteSequence = bytes()
824     for i in range(num):
825       b = random.randint(0,20)
826       b = chr(3) if b==0 else chr(0)
827       byteSequence += b
828     printBytes(byteSequence)
829     return byteSequence
830
831 class SerialConnection(Connection):
832
833   """
834   Represents a connection over the serial port.
```

```
835     """
836
837     def __init__(self, baud=9600, name=None, parent=None, singleThreading
            =None):
838       Connection.__init__(self, name, parent)
839       """
840       Constructs a serial connection.
841       """
842       self._baud = baud
843
844     def startup(self, port='/dev/rfcomm0'):
845       """
846       Opens serial port and calls startup interface.
847       """
848       if self._initialized: return
849       self._serial = serial.Serial(port=port, baudrate=self._baud)
850       self._serial.open()
851       self._initialized = True
852       self._callStartup()
853
854     def terminate(self):
855       """
856       Executes termination interface and closes connection.
857       """
858       self._callTerminate()
859       self._serial.close()
860
861     def write(self, byteSequence):
862       """
863       Writes a byte sequence to the serial port.
864       """
865       self._serial.write(byteSequence)
866       print "--------- WROTE (serial) ----------"
867       printBytes(byteSequence)
868
```

```
869   def read(self, num=1):
870     """
871     Reads a byte sequence from the serial port.
872     """
873     byteSequence = self._serial.read(num)
874     print "--------- READ (serial) ----------"
875     printBytes(byteSequence)
876     return byteSequence
877
878
879 class DomainInterface:
880
881   """
882   Represents a domain interface.
883
884   A domain interface is supposed to be a standardized name and message
        format
885   which can pass into the model. It is then normally tied to some local
886   interface. Domain interfaces have no return values.
887   """
888
889   def __init__(self,
890     parent = None,
891     name=None,
892     localInterface=None,
893     interfaceArguments=[],
894     parameters=[]):
895
896     """
897     Constructs a domain interface.
898
899     A domain interface has some parameters (a set of names) which must
          be
900     satisfied by the calling entity. These parameters are then mapped
          into
```

```
    arguments that the local interface will accept.
    """

    self._parent = parent
    self._name = name
    self._localInterface = localInterface
    self._interfaceArguments = interfaceArguments
    self._parameters = parameters

  def call(self, givenEnv):
    env = dict()
    try:
      for key in self._parameters:
        env[key] = givenEnv[key]
    except KeyError as e:
      sys.stderr.write("Missing argument for {}: {}\n".format(self.
          _name,str(e)))
      return

    values = safeEvalAll(self._interfaceArguments,env)
    self._parent._callInterface(self._localInterface, values)

class DomainOutput:

  def __init__(self,
      parent=None,
      name=None,
      returns=None):
    self._parent = parent
    self._name = name
    self._returns = returns


  def call(self):
    outMsg = dict(self._returns)
```

```
935     stateVars = self._parent._buildStateVarEnvironment()
936
937     for (key, val) in outMsg.items():
938       outMsg[key] = safeEval(val, stateVars)
939
940     self._parent._receiveMessage(self, outMsg)
941
942   def getName(self):
943     return self._name
944
945
946 class Interface:
947
948   def __init__(self, parent=None, name=None, type="adhoc", parameters
        =[],
949       triggers=None):
950     self._parent = parent
951     self._name = name
952     self._type = type
953     self._parameters = parameters if parameters != None else []
954     self._triggers = triggers
955     self._primitiveCalls = []
956
957   def addPrimitiveCall(self, name, arguments, delay=0, priority=0):
958     self._primitiveCalls.append({
959       'name': name,
960       'arguments': arguments,
961       'delay': delay,
962       'priority': priority
963     })
964
965   def call(self, arguments):
966     env = createEnvironment(self._parameters, arguments)
967
968     for primCall in sorted(self._primitiveCalls, key=lambda x: x['
```

```
        priority']):
969         args = safeEvalAll(primCall['arguments'], env)
970         self._parent._callPrimitive(primCall['name'], args)
971         time.sleep(primCall['delay'] / 1000.)
972
973     if self._triggers != None:
974         self._parent._callDomainOutput(self._triggers)
```

APPENDIX B

AN EXAMPLE DESCRIPTION

```
1  {
2    "name": "irobotCreate",
3    "author": "Paul Kilgo",
4    "stateVariables": [
5      {
6        "type": "int",
7        "name": "bumperStates",
8        "value": ""
9      }
10   ],
11   "interfaces": [
12     {
13       "primitiveCalls": [
14         {
15           "priority": 0,
16           "delay": 100,
17           "arguments": [],
18           "name": "bumpers"
19         }
20       ],
21       "type": "periodic",
22       "name": "get_bumpers",
23       "triggers": "detect_bump"
24     },
25     {
26       "primitiveCalls": [
27         {
28           "priority": 0,
```

```
29            "delay": 100,
30            "arguments": [
31              "<velocity*1000>",
32              "<radius*1000>"
33            ],
34            "name": "drive"
35          }
36        ],
37        "type": "adhoc",
38        "name": "drive",
39        "parameters": [
40          "velocity",
41          "radius"
42        ]
43      },
44      {
45        "primitiveCalls": [
46          {
47            "priority": 0,
48            "delay": 1000,
49            "arguments": [],
50            "name": "start"
51          },
52          {
53            "priority": 1,
54            "delay": 1000,
55            "arguments": [],
56            "name": "full"
57          }
58        ],
59        "type": "adhoc",
60        "name": "startup"
61      },
62      {
63        "primitiveCalls": [
```

```
64          {
65            "priority": 0,
66            "delay": 100,
67            "arguments": [
68              "<0>",
69              "<0>"
70            ],
71            "name": "drive"
72          }
73        ],
74        "type": "adhoc",
75        "name": "shutdown"
76      }
77    ],
78    "domainOutputs": [
79      {
80        "returns": {
81          "distance": "<int((bumperStates&3)==0)>"
82        },
83        "name": "detect_bump"
84      }
85    ],
86    "connections": {
87      "spp": [
88        {
89          "baud": 57600,
90          "singleThreading": {
91            "freqHz": 40
92          },
93          "name": "btserial",
94          "startup": {
95            "arguments": [],
96            "name": "startup"
97          },
98          "terminate": {
```

```
99            "arguments": [],
100           "name": "shutdown"
101         },
102         "keepalive": {
103           "interval": 1000,
104           "arguments": [],
105           "name": "get_bumpers"
106         }
107       }
108     ]
109   },
110   "primitives": [
111     {
112       "name": "start",
113       "formatArgs": [
114         "<128>"
115       ],
116       "connection": "btserial",
117       "pack": "B"
118     },
119     {
120       "name": "full",
121       "formatArgs": [
122         "<132>"
123       ],
124       "connection": "btserial",
125       "pack": "B"
126     },
127     {
128       "name": "drive",
129       "parameters": [
130         "velocity",
131         "radius"
132       ],
133       "formatArgs": [
```

```
134          "<137>",
135          "<velocity>",
136          "<radius>"
137        ],
138        "connection": "btserial",
139        "pack": ">Bhh"
140      },
141      {
142        "postActions": [
143          "bumperStates = __out__[0]"
144        ],
145        "name": "bumpers",
146        "formatArgs": [
147          "<142>",
148          "<7>"
149        ],
150        "connection": "btserial",
151        "unpack": "B",
152        "pack": "BB"
153      }
154    ],
155    "domainInterfaces": [
156      {
157        "calls": {
158          "arguments": [
159            "<linear>",
160            "<linear/angular if angular != 0 else 32.767>"
161          ],
162          "name": "drive"
163        },
164        "name": "set_velocity",
165        "parameters": [
166          "angular",
167          "linear"
168        ]
```

```
        }
    ]
}
```

APPENDIX C

RDIS BROKER

```
1  #!/usr/bin/env python
2  import roslib; roslib.load_manifest('rdis')
3  import rospy
4  import sys
5  import rdis
6
7  from collections import deque
8
9  ## Imports resolved by rosmake
10 from std_msgs.msg import Bool
11 from geometry_msgs.msg import Twist
12
13 ## RDIS model interpreter.
14 gModel = None
15
16 ## Maps topics (string) to ROS publsiher for topic
17 gPublishers = dict()
18
19 ## RDIS model callback
20 def rdisCallback(msg):
21   (pub, callback) = getPublisher(msg["name"])
22   callback(msg["contents"], pub)
23
24 ## Named publishers functions
25 def registerPublisher(name, pub, callback):
26   global gPublishers
27   gPublishers[name] = (pub, callback)
28
```

```
29 def getPublisher(name):
30   global gPublishers
31   return gPublishers[name]
32
33 def mapDomainInterface2Topic(interface):
34   return
35
36 ## Auto-generated callback for a subscribed topic.
37 def setSpeed_callback(data):
38   global gModel
39
40   ## Flatten into tuples
41   angular = (data.angular.x, data.angular.y, data.angular.z)
42   linear = (data.linear.x, data.linear.y, data.linear.z)
43
44   env = dict()
45   env["angular"] = angular
46   env["linear"] = linear
47
48   env['angular'] = rdis.safeEval('<angular[0]>', env)
49   env['linear'] = rdis.safeEval('<linear[0]>', env)
50
51   gModel.callDomainInterface('set_velocity', env)
52
53 ## Auto-generated action function for publishing to a topic.
54 def publish_hitObject(rdisMsg, pub):
55   env = dict()
56
57   ## Generated initializations from model.
58   env['distance'] = rdisMsg['distance']
59
60   ## Generated mappings to ROS bool from any domain adapter of RDIS.
61   env['data'] = rdis.safeEval('<distance == 0>', env)
62
63   ## Statements above should have initialized env['data']
```

```
64    pub.publish(Bool( env['data'] ))
65
66
67  def main():
68    if len(sys.argv) != 3:
69      print "Usage: " + sys.argv[0] + " <rdis-file> <serial-port>"
70      sys.exit(1)
71
72    print "Starting up RDIS..."
73    global gModel
74    gModel = rdis.load( sys.argv[1] )
75    gModel.setCallback(rdisCallback)
76    gModel.startup(port=sys.argv[2])
77
78    rospy.init_node('rdisWanderer', anonymous=True)
79
80    ## Auto-generated statements for initializing publishers and
        subscribers
81    rospy.Subscriber('setSpeed',Twist,setSpeed_callback)
82    registerPublisher('detect_bump', rospy.Publisher('hitObject', Bool),
        publish_hitObject)
83
84    while not rospy.is_shutdown():
85      ## Lets model know a thread tick has passed. Internal
86      ## timers will govern whether or not it is time to call
87      ## any periodic interfaces.
88      gModel.tick()
89
90      ## Loop at 40.0 Hz
91      rospy.sleep(1.0 / 40.0)
92
93    gModel.terminate()
94
95  if __name__ == "__main__":
96    try:
```

```
    main()
  except rospy.ROSInterruptException:
    global gModel
    gModel.terminate()
    pass
```