Automatic Software Interface Generation
via Feature Oriented Analysis

by

Ross Corey Taylor, B.S.

Thesis

Presented to the Faculty of the Graduate School
of The University of Texas at Austin in
Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Engineering

The University of Texas at Austin

August 2004
Automatic Software Interface Generation
via Feature Oriented Analysis

APPROVED BY
SUPERVISING COMMITTEE:
To my wife
Elizabeth Mitchell Taylor
and my parents
Susan and John Taylor
Acknowledgments

I would like to thank Dr. Delbert Tesar for providing the opportunity to perform this research at the Robotics Research Group. I would also like to thank Dr. Chetan Kapoor for his technical guidance and his editorial review of my work. Lastly, I would like to thank the students of the RRG, who provided support at various levels for this report and who provided friendship in and out of the workplace.

This research was funded by the Department of Energy University Research Program in Robotics (grant number: DE-FG04-94EW37966) and the Texas Advanced Technology Program (grant number: 003658-0034-2001).
Automatic Software Interface Generation
via Feature Oriented Analysis

by

Ross Corey Taylor, M.S.E.
The University of Texas at Austin, 2004

SUPERVISOR: Delbert Tesar
CO-SUPERVISOR: Chetan Kapoor

Software interfaces are often defined as a layout of the controls for an application through which user interaction is performed. One of the fundamental concepts of this report requires a slightly different definition. A software interface is considered to be a well-defined algebraic specification of the inputs and outputs of a system. From this specification, interface representations can be generated that define the layout of the controls. With this view, a system has only one software interface but may have many distinct interface representations. This research builds on this concept to develop an approach to interface specification that allows algebraic reduction, optimization, and translation into desired representations.

The approach is based on the software engineering concept of Feature Oriented Programming (FOP), which relies on composition of features to build software systems. This research defines software interfaces as features of a system in terms of properties and actions. Properties are attributes of the system that have a single known value at any time, which can be monitored or modified by the user.
Actions are defined for each property as the means for monitoring and modifying the property.

This report presents an approach to interface synthesis that relies on domain analysis, product-line analysis, interface generation, and interface translation. Domain analysis and product-line analysis are performed by a domain expert to specify the features of the domain in an Extended Backus-Naur Form (EBNF). An algebraic specification of the interface components, from which the interface to a particular product can be generated, is inferred from the EBNF specification. Interface generation involves feature composition, interface reduction, and interface optimization. These steps are defined based on the algebraic specification of the interface components and a supplementary specification of semantic relationships between properties and actions. Lastly, interface translation is performed to produce the desired interface representation from the interface specification for a product.

The approach is demonstrated with a robotics domain example. The example covers a relatively complex system with thirteen properties, forty-two property values, and twenty-three actions that span system control, motion control, tool control, and various types of feedback. Also, twelve features were identified for the product family. Six of the features are composed to demonstrate interface generation and translation. The composition results in an interface represented by 49 algebraic statements with several redundancies. Interface reduction removes the redundancies and results in a syntactically complete interface with 28 algebraic statements (43% reduction). However, the effectiveness of the interface can be improved by applying semantic criteria to modify the syntax as is done during interface optimization. This semantics based optimization leads to a reduction of possible action combinations from 4096 to 1152. Lastly, interface translation is demonstrated via a sequential approach and a hierarchical approach. The first results in a C++ representation and the second results in a XML schema representation. Much of this work is conceptual, but the groundwork has been laid for future research into these areas.
# Table of Contents

List of Tables ............................................................................................................. xiii
List of Figures ............................................................................................................. xv
Nomenclature ............................................................................................................. xvi
Chapter One .................................................................................................................. 1
  1. Introduction....................................................................................................... 1
     1.1. Scope............................................................................................................. 3
     1.2. Research Objectives..................................................................................... 5
     1.3. Approach...................................................................................................... 7
         1.3.1. Feature Oriented Programming............................................................. 7
         1.3.2. Domain Analysis................................................................................... 8
         1.3.3. Product-Line Analysis .......................................................................... 9
         1.3.4. Interface Generation............................................................................ 10
         1.3.5. Interface Translation........................................................................... 10
     1.4. Synopsis ...................................................................................................... 11
Chapter Two................................................................................................................ 13
  2. Background and Literature Review .................................................................... 13
     2.1. The Human-Robot System.......................................................................... 14
     2.2. The Five Layers of the Human-Robot System ........................................... 16
     2.3. Human Operator.......................................................................................... 17
     2.4. Interface Device .......................................................................................... 18
         2.4.1. Manual Controllers ............................................................................. 18
             2.4.1.1. Types........................................................................................... 19
             2.4.1.1.1. Joint-Level ............................................................................... 20
             2.4.1.1.2. Kinematic Replica................................................................. 20
             2.4.1.1.3. Universal .................................................................................. 21
             2.4.1.1.4. Delta ......................................................................................... 22
             2.4.1.2. Control Methods ........................................................................ 22
                 2.4.1.2.1. Joint-to-Joint ............................................................................ 22
                 2.4.1.2.2. Cartesian .................................................................................. 23
                 2.4.1.2.3. Delta ......................................................................................... 23
             2.4.1.3. Force Reflection .......................................................................... 24
     2.4.2. Other Interface Tools .......................................................................... 24
         2.4.2.1. Computer Vision................................................................. 25
         2.4.2.2. Gesture Recognition ................................................................. 25
         2.4.2.3. Acoustic Speech Recognition .................................................... 26
         2.4.2.4. Hybrid Speech Recognition ....................................................... 28
         2.4.2.5. Graphical User Interface ............................................................. 28
     2.4.3. Interface Software............................................................................... 29
         2.4.3.1. On-Line Programming ................................................................. 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.3.1.1</td>
<td>Point-to-Point Training</td>
<td>30</td>
</tr>
<tr>
<td>2.4.3.1.2</td>
<td>Recorded Trajectories</td>
<td>31</td>
</tr>
<tr>
<td>2.4.3.2</td>
<td>Off-Line Programming</td>
<td>31</td>
</tr>
<tr>
<td>2.4.3.2.1</td>
<td>Specialized Robotic Languages</td>
<td>31</td>
</tr>
<tr>
<td>2.4.3.2.2</td>
<td>General-Purpose Languages</td>
<td>32</td>
</tr>
<tr>
<td>2.4.3.2.3</td>
<td>Existing Computer Languages</td>
<td>32</td>
</tr>
<tr>
<td>2.4.3.2.4</td>
<td>Other Programming Methods</td>
<td>33</td>
</tr>
<tr>
<td>2.4.3.2.5</td>
<td>Graphical Programming</td>
<td>34</td>
</tr>
<tr>
<td>2.4.3.2.6</td>
<td>Language Translation</td>
<td>34</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Robot Control Software</td>
<td>35</td>
</tr>
<tr>
<td>2.4.4.1</td>
<td>State Machine Representation</td>
<td>35</td>
</tr>
<tr>
<td>2.4.4.2</td>
<td>Virtual Robot Controller</td>
<td>36</td>
</tr>
<tr>
<td>2.4.4.3</td>
<td>Open Architecture Controller</td>
<td>36</td>
</tr>
<tr>
<td>2.4.4.4</td>
<td>Generated Controller</td>
<td>37</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Robot Hardware</td>
<td>37</td>
</tr>
<tr>
<td>2.4.5.1</td>
<td>Robot Types</td>
<td>38</td>
</tr>
<tr>
<td>2.4.5.2</td>
<td>Sensors</td>
<td>38</td>
</tr>
<tr>
<td>2.4.5.3</td>
<td>Tools</td>
<td>38</td>
</tr>
<tr>
<td>2.5</td>
<td>Summary</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Interface Syntax</td>
<td>40</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Algebraic Interface Specification</td>
<td>41</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Context-Free Grammar Interface Specification</td>
<td>44</td>
</tr>
<tr>
<td>3.1.2.1</td>
<td>Context-Free Grammars</td>
<td>46</td>
</tr>
<tr>
<td>3.1.2.2</td>
<td>Interface Generation Grammar</td>
<td>47</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Properties</td>
<td>48</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Actions</td>
<td>51</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Features</td>
<td>52</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Syntactic Interface Composition</td>
<td>55</td>
</tr>
<tr>
<td>3.2</td>
<td>Interface Semantics</td>
<td>56</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Semantic Definition Syntax</td>
<td>57</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Physical Relationships</td>
<td>58</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Time-Based Relationships</td>
<td>60</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Conditional Relationships</td>
<td>61</td>
</tr>
<tr>
<td>3.2.4.1</td>
<td>Transition Tables</td>
<td>63</td>
</tr>
<tr>
<td>3.2.4.2</td>
<td>Transition Matrices</td>
<td>64</td>
</tr>
<tr>
<td>3.2.4.3</td>
<td>Application Specific Representations</td>
<td>64</td>
</tr>
<tr>
<td>3.2.4.4</td>
<td>Conditional Relationship Formalization</td>
<td>65</td>
</tr>
<tr>
<td>3.3</td>
<td>Summary</td>
<td>67</td>
</tr>
<tr>
<td>Chapter Four</td>
<td></td>
<td>69</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1: Various Tool Types ................................................................. 39
Table 3.1: Sample Alarm Clock Program .............................................. 45
Table 3.2: Sample Alarm Clock XML Interface .................................... 45
Table 3.3: EBNF Grammar Symbols ...................................................... 46
Table 3.4: EBNF Grammar Bracket Pairs .............................................. 47
Table 3.5: Property Specification for an Alarm Clock ........................... 50
Table 3.6: Action Specification for an Alarm Clock ............................. 52
Table 3.7: Features for Alarm Clock Product Line ............................... 54
Table 3.8: Time Feature for a Self-Setting Clock ................................ 55
Table 3.9: Semantic Grouping Symbols ................................................. 57
Table 3.10: Physical/Time-Based Relationship Symbols ....................... 58
Table 3.11: Conditional Relationship Symbols ..................................... 58
Table 3.12: Format of State Machine Transition Tables ......................... 64
Table 4.1: Robot Control Software Features ......................................... 70
Table 4.2: Additional Features Identified for Interface Generation ........ 71
Table 3.12: State Diagram Symbols ....................................................... 73
Table 4.3: Basic System State Property Values ...................................... 75
Table 4.4: Complete System State Property Values ............................. 77
Table 4.5: System State Actions ............................................................. 77
Table 4.6: Joint Command Properties’ Values ...................................... 78
Table 4.7: Joint Command Properties’ Actions ...................................... 79
Table 4.8: End-Effector Command Properties’ Values ............................ 81
Table 4.9: End-Effector Command Properties’ Actions .......................... 82
Table 4.10: Tools Identified for the Product-Line ................................. 82
Table 4.11: Tool State Property and Actions for the Gripper ................. 83
Table 4.12: Tool State Property and Actions for the Drill ....................... 84
Table 4.13: Tool State Property and Actions for the Paint Gun ............. 85
Table 4.14: Tool State Property and Actions for the Saw ....................... 85
Table 4.15: Overall Tool State Property Values and Actions .................. 86
Table 4.16: Tool Property Values and Actions ....................................... 87
Table 4.17: Force/Torque Sensor Property Values ................................. 87
Table 4.18: Force Torque Sensor Property Actions ............................... 88
Table 4.19: Collision Distance Property Values and Actions .................. 88
Table 4.20: Obstacle Artificial Force/Torque Property Values and Actions 89
Table 4.21: Domain Model Summary for the Robotics Example (Properties 1-5) .... 89
Table 4.22: Domain Model Summary for the Robotics Example (Properties 6-13) .. 90
Table 4.23: Formally Specified Features .............................................. 91
Table 4.24: Value Motion Feature ......................................................... 92
Table 4.25: Joint Command Features .................................................... 93
Table 4.26: End-Effector Command Features ........................................ 93
List of Figures

Figure 1.1: Robotic System Diagram................................................................. 3
Figure 1.2: Two Distinct Interfaces for a Simple System................................. 5
Figure 2.1: Layers of the Human-Robot System.............................................. 17
Figure 2.2: GMFanuc Industrial Robot Teach Pendant.................................... 18
Figure 2.3: Various Hand Controllers.............................................................. 19
Figure 2.4: Teleoperation Systems (TOS) Robot............................................. 20
Figure 2.5: University of Texas 9-String 6-DOF Force Reflecting Joystick...... 21
Figure 2.6: Logitech Magellan 6-DOF Delta Controller.................................. 22
Figure 2.7: Simple Robotic State Diagram..................................................... 36
Figure 3.1: State Diagram for Alarm State Feature......................................... 61
Figure 3.2: State Diagram for Alarm Clock Interface...................................... 62
Figure 3.3: Digital Alarm Clock GUI............................................................... 67
Figure 4.1: State Diagram for Basic System State Property............................ 74
Figure 4.2: State Diagram for the Complete System State Property................. 76
Figure 4.3: State Diagram for the Gripper...................................................... 83
Figure 4.4: State Diagram for the Drill.......................................................... 84
Figure 4.5: State Diagram for the Paint Gun.................................................. 84
Figure 4.6: State Diagram for Tool Property.................................................. 86
Figure 4.7: System State Diagram with $val$ State...................................... 92
Figure 4.8: State Diagram for the Complete System State Property.............. 98
Figure 4.9: Combined State Diagram for the Tool State and Tool Properties... 101
Figure 5.1: Feature Selection GUI................................................................. 105
Figure 6.1: Diagram for the Drill Power State............................................... 131
Figure 6.2: Feature Selection GUI................................................................. 133
### Nomenclature

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>CC</td>
<td>Computational Components (of RCS)</td>
</tr>
<tr>
<td>CFG</td>
<td>Context Free Grammar</td>
</tr>
<tr>
<td>DI</td>
<td>Device Interface (of RCS)</td>
</tr>
<tr>
<td>DOF</td>
<td>Degree-Of-Freedom</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus-Naur Form</td>
</tr>
<tr>
<td>EE</td>
<td>End-Effector</td>
</tr>
<tr>
<td>FOP</td>
<td>Feature Oriented Programming</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model</td>
</tr>
<tr>
<td>HRS</td>
<td>Human-Robot System</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>MC</td>
<td>Manual Controller</td>
</tr>
<tr>
<td>OSCAR</td>
<td>Operational Software Components for Advanced Robotics</td>
</tr>
<tr>
<td>RCS</td>
<td>Robot Control Software</td>
</tr>
<tr>
<td>RPL</td>
<td>Robot Programming Language</td>
</tr>
<tr>
<td>RRG</td>
<td>Robotics Research Group (at UTA)</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>UIML</td>
<td>User-Interface Markup Language</td>
</tr>
<tr>
<td>UTA</td>
<td>the University of Texas at Austin</td>
</tr>
<tr>
<td>WIMP</td>
<td>Windows, Icons, Menus, and Point-and-click devices</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter One

1. Introduction

To most people, the term *interface* implies the physical presence of how humans interact with devices. Often, people think of the typical computer interface with windows, menus, icons, and a mouse pointer. Jef Raskin defines an interface as “the way that you accomplish tasks with a product—what you do and how it responds [45].” As this definition conveys, the interface to a device deals with the commands it understands, not the way they are presented to a user. To keep this distinction clear the term *interface representation* will be defined as the user interface or the physical display and controls that the user is given for communicating with a device. There is only one *interface* to a system while there may be many *interface representations*.

The interface representation to a computer can be graphical, textual, vocal, binary (TCP/IP), etc. Though some of these interfaces are of a higher level than others, in principle they all have access to the same capabilities. In other words, they are really the same interface, they just look different. This research builds on this idea to create an interface specification that can be reduced and optimized algebraically and then translated into desired representations.

This is particularly applicable in today’s factories where rapid deployment of automation is becoming a key focus. The interface requirements of devices are shifting from interfacing with humans to interfacing with other devices. Using the
concepts developed in this report, the software interface for a system can be specified algebraically, translated into a representation suitable for computer integration.

Traditionally, software and the interfaces to software have been developed as needed and with little concern for reuse and extensibility. If a device needs to be implemented in an automated system, it is programmed with the required functionality and interface. While the initial product may be well designed and implemented, neglecting these concerns in the design stage can cause various problems [5]. Reuse of system software reuse is difficult because the product has been developed for a specific application and often for a specific machine. It is unlikely that the device could be moved to a different automation cell and function properly. Also, extensibility is hindered as constant refinement of processes and addition of features creates large and often inefficient software. On the software level, various approaches have been developed to combat this problem. One approach is Feature Oriented Programming (FOP) which is based on the idea of product-lines of software (see Section 1.3.1). FOP develops a set of features for a product-line and defines the effect each feature has on the software. From this, product instances can be created by composing the desired features.

Similar difficulties arise in software interfaces and are compounded by interface development as an afterthought in software systems [2]. Raskin points out that even the simplest tasks are often given convoluted interfaces particularly when the simple task is part of a complex mechanism [45]. The classic example is the VCR which is notoriously complicated to use even when performing the simple task of setting the time.

No matter how the software for a device is developed, the present research can be applied to identify the singular interface for the device, improving its reuse, portability, and extensibility. This research provides a shift in the way software interfaces are approached, presenting the interface as a representation of an underlying system. The following section defines the scope of this research.
1.1. Scope

All of the information in this report must be prefaced with an understanding that the research was performed under the Robotics Research Group at the University of Texas and was, therefore, developed with the intent of application to the robotics domain. This disclaimer aside, these methods are applicable to any well-understood domain. The term software interface is used to denote any piece of software designed to facilitate the communication between two independent entities. The interface is not simply a communication language; it also provides information about the system and the validity of commands in various states.

In a robotic system, examples of software interfaces include software for interacting with the human, software for interacting with other devices, and software for communication within the system. This section gives a brief introduction to robotic systems based on the example system shown in Figure 1.1 (adapted from [6]). For a detailed discussion of the human-robot system, see Chapter 2.

![Figure 1.1: Robotic System Diagram](image)
The software involved in a robotic system can be separated into two layers: the Robot Control Software (RCS) and the interface. The RCS layer includes the Computational Components (CC) and the Device Interface (DI). The CC performs calculations based on input from the interface layer in order to control the movements of the robot. If the interface provides position commands, the CC layer may be nonexistent. It is likely, however, that the Robot Control Software (RCS) will at least include kinematic calculations or interpolation between the current position and the desired position. The CC could also include higher level computation of dynamics, deflection modeling, obstacle avoidance, path planning, and other sophisticated algorithms.

The Device Interface (DI) portion of the RCS includes the software to interact with the robotic hardware. Beyond a simple pass through of information, the DI can also provide functionality like limits checking, sensor data filtering, command queuing, etc. The DI might implement real-time functionality independent of the data transfer rate of the CC. For instance, actuators may require a fixed data rate for safety and stability. Also, the DI could provide additional, more complex functionality like the emulation of a velocity command by sending position commands to an actuator at a fixed rate.

The focus of this report is on the interface layer of the robotic system. The premise of this research rests on the assumption that there is only one interface to a device for a given objective. In other words, the interface to a device can be uniquely identified by some systematic, consistent, and repeatable method. Therefore, each of the interface components (RPL, GUI, etc.) shown in Figure 1.1 is a representation of the singular interface to the robot control system. The traditional definition of an interface is therefore reclassified into an interface representation and the term interface is defined as the singular definition of the communication with a device. While the interface representations may appear very different, they represent the same fundamental functionality to command the RCS.
1.2. Research Objectives

This research intends to separate the interface from the interface representation to provide consistency and interoperability among representations. An analytical method based on the product-line approach is defined for characterizing the interface to a system. This approach defines the underlying features of a family of products so that any individual product in the product-line can be generated by composing the features [36] (see Section 1.3.1). Typically, this approach is applied to software systems. This research will demonstrate its application to the interface of software and hardware systems including a detailed example of interface generation for robotic systems.

The analytical definition of the interface will include both the syntax and the semantics of the interface. The syntax addresses the actual format of the data that is transferred across the interface, its structure and order. This is defined by a textual specification and an algebraic specification each providing particular benefits (see Section 3.1). The semantics deals with the meaning of the data and its use. Nearly all devices are modal which affects the interface to the devices [45]. While the value of modes can be debated, the semantics of an interface must take them into account. A specific command laid out in the syntax of the interface may have different meanings or no meaning at all based on the mode. The two simple interface representations of Figure 1.2 demonstrate the difference in interface syntax and interface semantics.

![Figure 1.2: Two Distinct Interfaces for a Simple System](image)
While the two interface representations of Figure 1.2 are very similar, there is a subtle difference in the semantics that would represent the two. The syntax of each of these interfaces simply contains the commands on and off. In this respect, they are identical. At the semantic level, however, the switch in Figure 1.2a can only send an on command from the off state and similarly can only send an off command from the on state. The buttons in Figure 1.2b, on the other hand, do not restrict when commands are valid. This comes down to a semantic difference due to conditional relationships of the commands. Section 3.2.4 discusses conditional relationships in detail.

Other semantic relationships can also exist that are used for runtime analysis and optimization of the syntax. These include physical relationships and time-based relationships between actions. The button interface of Figure 1.2b demonstrates the importance of physical relationship identification. While the on button or the off button can be pressed at any time, it is invalid to press both at the same time. This is a physical relationship in that the system being controlled cannot be turned on and off at the same time. Time-based relationships are similar to physical relationships but with a time component included as part of the restriction. The simplest example of this type of restriction is the relationship between position and velocity of an object over a period of time. These semantic restrictions are covered in much more detail in Section 3.2.

The overall objective of this report is to begin a research thread that will lead to a system capable of generating interfaces and interface representations for complex devices within well-understood domains. This report presents a systematic approach to generating interfaces and interface representations. Particularly, a well-defined method for producing an algebraic definition of the syntax and semantics for an interface is presented. Also, an overview of the process for constructing, reducing, and optimizing the interface based on its features is given, and initial examples of interface representation are presented. This report lays the groundwork for future
research in interface generation leading to a system for interface specification, design, and implementation.

1.3. Approach

The approach includes the following steps, given in the order they should be addressed in interface definition and generation:

1) Domain Analysis
2) Product-Line Analysis
3) Interface Generation
4) Interface Translation

Each of these steps is introduced in this section and examples of their application can be seen throughout this work. As a foundation for the product-line approach used in this research, feature oriented programming is presented in the following subsection.

1.3.1. Feature Oriented Programming

Feature Oriented Programming (FOP) is a method of software design that synthesizes software by identification of its features. It is based on step-wise refinement, which composes software by making many incremental extensions to a product to produce the desired result. Feature oriented programming, however, uses large feature refinements to generate programs, thus making the number of refinements more manageable [5].

The techniques of feature oriented programming allow development of product-lines in much the same way they are used in other applications. A common application of product-lines is the automotive industry where the customer can customize a base vehicle. Options are provided for various features like engine, transmission, color, etc. In software as well, there is typically a base program for a product-line. The program is then refined by adding features to produce the desired product.
A current research thread at the Robotics Research Group (RRG) at the University of Texas at Austin is applying feature oriented programming to generation of robot control software [26]. This research addresses the interface to the robotic hardware and the computational components to control the robot. This work uses RRG’s Operational Software Components for Advanced Robotics (OSCAR) to create robot control software. OSCAR is a set of C++ classes that encapsulates the computation required for robotics, including redundancy resolution [28].

Applying FOP with OSCAR generates a control program that includes the proper classes. The resulting control software has a C++ Application Program Interface (API) that allows a user to create a control program in C++. If the user desires another interface, it must be developed by the user in conjunction with the API. The methods presented in this report will make the interface generation for the control software significantly less complicated.

1.3.2. Domain Analysis

The purpose of the domain analysis is to define the components from which the interface to products can be generated. Interfaces are a means for conveying information, either inbound to the system or outbound from the system. The definition of the core components of an interface is based on a general assumption: all outbound information consists of the value of some property and all inbound information is an action on a property. Defining these properties, property values, and actions in an analytical manner provides a clear understanding of the functions and capabilities of an interface.

Both syntactic and semantic definitions of the interface will result from the domain analysis. However, the specifics may be incomplete as this analysis will be valid for the overall domain (product-line) and will have to be adapted for a particular product instance. Clearly, the domain analysis and product-line analysis will likely not be independent, because a full analysis of a domain may go well beyond the scope
of any products being considered. However, it is important to keep them as distinct processes for generality. See Section 3.1 for details on syntax, Section 3.2 for details on semantics, and Chapter 4 for an example analysis of the robotics domain.

1.3.3. Product-Line Analysis

Product-line analysis is an extension of domain analysis. Often, the product-line will be addressed in conjunction with the domain because only one product-line is being defined within the domain. However, when it is logical to perform an independent domain analysis, the distinction between the domain analysis and product-line analysis must be clear.

Product-line analysis builds on domain analysis by restricting the syntax and semantics to the target product-line. This entails restricting the property values and actions to those applicable to the target product-line. Since the product-line always falls within the overall domain, the product-line property values and actions represent a subset of the domain’s property values and actions. With the properties and actions appropriately restricted, the next step in refining the syntactic specification is to identify features of the product-line. Features are generally intuitive and identified based on well understood extensions to a basic system. For each of these features, an interface can be identified that leverages the properties and actions specified for the product-line.

With the syntax specified, semantics is addressed to add context of the product-line to the syntactic definition. Because semantics is system dependent, it is expected that the majority of the semantic specification will come from product-line analysis. While some semantic restrictions may be defined at the domain level, more details are known about how the system will handle various commands at the product-line analysis. Semantics is discussed in detail in Section 3.2.
1.3.4. Interface Generation

Interface generation is where all the analysis comes together and begins to produce a product. First, from the identified features, the desired set for the given product is selected. Next, the interface components of the individual features are automatically combined into a single interface. A simple combination of these features would likely create an overly bloated interface with redundancies. Therefore, the interface is then automatically reduced and optimized using techniques discussed in Section 5.1. This reduction and optimization is only possible if the interface is defined algebraically.

1.3.5. Interface Translation

With the interface fully specified in this manner, translation into the desired interface representation is addressed. It is important to note that different interface representations may not use the complete definition of the interface in the same way. For instance, a single interface definition might be represented as a mechanical interface for the human to interact with, a software interface for the device, and a custom communications protocol between the two. Thus, the interface would have three distinct representations. Each of these interface representations should adhere precisely to the syntactic definition of the interface; although, they may only use a subset. Also, the use of the semantics may vary. It is expected that the software interface to the device would use the semantic definition in its entirety to guarantee safe operation while the communications protocol would likely adhere only to the syntax. This is because the communications relate only to the conveyance of information, not its meaning. At the same time, the user interface might use parts of the semantic definition but not all of the functions provided to the user.

For instance, a digital alarm clock often has a snooze mode that can only be entered when the alarm is sounding. This is a semantic restriction. Attempting to put the alarm in snooze mode is only valid at certain times. Typically, however, a clock
has a dedicated snooze button. Therefore, a user can push the snooze button at any time. The clock’s hardware interface representation ignores the semantic restriction, but the software interface uses the semantic definition to automatically ignore the command. Similarly, the alarm can only be turned on when it is off. This is clearly represented on the clock’s hardware interface representation by a switch. The digital alarm clock example will be used throughout this work as a simplified example of the interface generation process.

1.4. Synopsis

The previous sections of this chapter presented the motivation for development of a new approach to interface design. The specific objectives of this research were also presented along with the general approach taken to achieve the desired goals.

Chapter 2 presents background material on robotics interfacing. It gives an overview of the human-robot interface system and various devices for human-robot interaction. While the interface has been presented as encompassing much more than just human-machine interaction, an extensive knowledge of this complicated part of the domain provides perspective into the various development stages of the interface generation scheme.

Chapter 3 presents the interface syntax and semantics. It addresses the primary steps in developing the syntax components for an interface. Domain analysis produces properties and actions that lie within the domain that define the input and output of the devices within the domain. Product-line analysis groups these components into features of the particular product-line, further defining the syntax. A textual interface specification is developed that allows human and machine readability and easy translation into interface representations. Also, an algebraic specification is developed that can be inferred from the textual representation to provide algebraic optimization of the interface.
Semantics deals with the meaning of the syntax at various states of the system as well as defining the validity of concurrent actions. Within the domain analysis and product-line analysis, physical, time-based, and conditional semantic relationships are defined. Physical relationships define constraints about the combination of various properties and actions in a single communication due to physical constraints. Similarly, time-based relationships deal with time relationships among properties and actions that might affect the syntax of the interface, as well as how the system should use input information. Lastly, conditional relationships define conditions on the current state of the device for a given action to be valid. All of these are expressed in a well-defined algebraic representation.

Chapter 4 provides an example of the domain analysis and product-line analysis for analyzing the interface. The example used is within the robotics domain. The result is a full interface definition ready to be reduced, optimized, and translated into various representations.

Chapter 5 presents interface generation and representation. The selection of features from those defined for a product family is discussed followed by strategies for combining them into a single interface. The combination must address reduction of redundancy in the interface and optimization of the interface based on the defined semantics. The chapter then goes on to discuss the translation of the robotics interface into desired representations. This discussion is facilitated using various examples. The primary example is an eXtensible Markup Language (XML) representation of the interface. An XML interface representation would provide remote access to a robot using XML as the communication protocol. Various other representation examples are also briefly presented.

Chapter 6 summarizes all of the work presented in this report. Conclusions as to the effectiveness of the research are given and future work for improving and extending the methods presented are put forth. The future work is outlined in a manner that if followed would produce an effective interface generator.
Chapter Two

2. Background and Literature Review

This chapter discusses in detail the components of the Human-Robot System (HRS). The human interface to robots can range from hardwired connections to complex systems of input via voice, GUI, etc. Clearly, hardwired solutions are not encompassed by this research since it is intended to address interfaces to software components; fortunately, the vast majority of robotic systems include a software interface for passing commands from the user to the robot. In fact, many high level robotic interfaces encompass the principles of computer interfacing as well. The presentation of these components will give perspective to the wide variety of interface representations that may be implemented on a single system.

Much of the information presented in this chapter, particularly pertaining to Manual Controllers (MC), was collected from human-robot interaction research conducted at the RRG. In 1978, Lipkin and Tesar began identifying the current state of the human-machine interface and the requirements and goals for the hardware and software of MCs [35]. This continued with a roadmap for future teleoperation systems in 1988 [49]. Much of the work at the RRG has been focused on the development of MC hardware. This includes the design, construction, and implementation of a variety of prototype hand controllers as well as the analysis of existing hardware developed elsewhere [42]. More recently, the focus of the RRG has begun to shift toward software to interface with these MCs [44]. In this continuing
development, however, other types of human interfaces, beyond MCs, have not been explored thoroughly.

The subsections of this chapter will present existing research in the interface between human and machine. First, a high level view of the human-robot system is presented including a discussion of an ideal breakdown into various parts of the system. This is followed by a more detailed discussion of the system components.

2.1. The Human-Robot System

The human-robot system is becoming increasingly important in the world today, especially as robot production begins to move from overwhelmingly industry focused toward consumer based products. The Human-Robot System (HRS) is any robotic system where a human interacts with the robot. The importance of the overall HRS design significantly increases as the amount of human contact increases. The interface is the part of the HRS that directly affects the interaction with the human and can vary dramatically based on the application.

The design and implementation of a successful interface between human and robot is vital to the success of any human-robot system. Though the interface may take on many different forms, the goal is always the same: to convey important information between the user and the robot. The communication to the robot may be as low-level as a keyboard giving joint-level commands or as high-level as voice commands to execute an automated task. Similarly, the communication from the robot can include anything from raw sensor data or video to performance information from calculations performed on data.

The interface for robotics is far from standard, and typically a robotic system is built and delivered with a custom interface for that system. This interface usually includes programming capability, a teach pendant to aid in programming, and indicators for feedback to the user. More recently, off-line programming packages are being included as well. This manufacturer provided interface provides access to all of
the robot’s features; however, the access may not be optimal for a particular user’s application of the robot. Users may desire to add teleoperation, telerobotics, automation, etc, or they may like to take a different approach to programming the robot. Therefore, it is often desired that a new interface be developed.

The development of a new interface is a significant problem by itself, but the interface designer is also burdened with learning the robot’s control code and programming language. Because virtually every robot manufacturer uses a unique, proprietary programming language at the controller level, the development of a custom interface is difficult. Also, this inconsistency means that once an interface is developed it is unlikely to operate on robots other than the one for which it was developed and porting it to another system becomes as complicated a task as starting from scratch.

The lack of a standard for interfaces causes difficulty with integration, interoperability, and reuse. The software integration costs alone for the United States’ robotics industry are estimated to be from $500 million to $1 billion annually [43], so industry has begun to demand a solution to this problem. The common approach being taken to solve this problem is the development of generalized open architecture control software. Open architecture control software is designed to replace existing control software on robots using standard interfaces. Developing truly universal control software, designing the interface language to the control software, and unanimously selecting a standard from the various approaches (OSCAR, SMART, RIPE, RCCL, OROCOS, ARCL, ROBOOP, etc.) is a monumental challenge [25].

The computer industry has taken a different approach in developing a standardized system for device integration. Using drivers, a standard interface can be put on top of whatever underlying interface a device may use. Similarly, using languages like HyperText Markup Language (HTML) and eXtensible Markup Language (XML), user interfaces can be described in a text format from which a system can implement the desired results in its own native language. This is
demonstrated by the internet which can be viewed on all platforms. These approaches effectively separate the interface from the implementation, likely improving the performance of each.

In order for the robotics industry to continue to grow, similar improvements need to be made for the integration of robots. Generalized, open architecture robotics software is a good goal to strive for, but the interface for robot integration should be developed separately removing this burden from control software developers. **Also, since it would be presumptuous to assume that one interface standard could encompass all the functionality that users might require, it would be more beneficial to develop a systematic approach to interface generation with absolute repeatability. In this manner, given an interface definition, two users could independently create interface representations that communicate successfully.**

The following sections present a structured description of the human-robot system and existing literature on the various topics.

### 2.2. The Five Layers of the Human-Robot System

In order to simplify the entanglement of hardware and software that makes up the Human-Robot System (HRS), it will be addressed as five distinct layers. These layers exist in nearly every HRS though they may be intermingled so that they cannot easily be separated. The layers of the HRS are identified as: the human operator, the interface device, the interface software, the robot control software, and the robot hardware (see Figure 2.1). These layers are discussed in detail in the following sections.
2.3. Human Operator

Clearly the human operator is an important part of every human-robot system. Even the most automated systems, such as space exploration robots, return data and video to the user. The user is included as part of the HRS because the user’s abilities, desires, and experience can directly affect the components of the system. Particularly, interface devices and interface software timing issues are affected by humans. The system must be able to achieve a rate suitable for human response, particularly when force-feedback is involved (see Section 2.4.1.3), and it is unnecessary to provide a rate that is excessively greater than what a human can comprehend. Lipkin and Tesar present the human operator’s limited output and input capabilities as a bottleneck in the HRS for which the other components should be designed [35]. Similarly, Adams
presents a human factors based argument for including the human operator as part of the design of an HRS in [2]. She addresses the human issues of decision making, situational awareness, vigilance, workload levels, and error.

2.4. Interface Device

The interface device is the portion of the interface system that interacts directly with the human operator. Some common interface devices in research include manual controllers, microphones, cameras, and speech synthesizers. In industry, the most common interface device is the *teach pendant* (see Figure 2.2). Robot programming languages, graphical user interfaces, command line interfaces, and other software driven interfaces are also lumped into this group due to their direct interaction with the user.

![GMFanuc Industrial Robot Teach Pendant](image)

**Figure 2.2: GMFanuc Industrial Robot Teach Pendant**

2.4.1. Manual Controllers

With the prevalence of teleoperation in robotic applications today, manual controllers are an integral part of the HRS. While some systems eliminate MC input with enough automation or other interface types, the overwhelming majority still use MCs. Even industrial robots have simple teach pendants that allow incremental movements in joint or end-effector (EE) space. This section briefly presents a few
types of MCs, some control methods used for MCs, and force feedback through MCs. This section also contains a brief overview of MC technology; see [49] for a more exhaustive discussion of MC types and control methods, [42] for a detailed discussion of MC performance evaluation, and [16] for a fairly extensive survey of commercially available MCs.

2.4.1.1. Types

Manual controllers can be categorized into four basic types. Each is designed with a specific goal in mind, and they are usually intended for a specific control method, but they can often be used in other ways. The types presented here are joint-level MCs, kinematic replica MCs, universal MCs, and delta MCs. Though some controllers may not quite fit any of these descriptions, the data that they provide can always be classified as absolute joint, absolute Cartesian, or delta commands. Several hand controllers are shown in Figure 2.3.

![Figure 2.3: Various Hand Controllers](image)

(a) BG Systems Flybox  
(b) TOS Force Reflecting Mini-Master  
(c) RSI 6-DOF Controller  
(d) Kraft Force Reflecting Controller  
(e) PerForce Force Reflecting Controller
2.4.1.1.1. Joint-Level

Joint-level manual controllers include any device that is intended to simply command the joint position of the robot. These devices provide no direct information to the operator as to the location of the end-effector or its change in position when a command is given. Therefore they require many hours of practice to develop useful operating skills. The most common types of joint-level MCs are lever-boxes or button-boxes. An example is the interface to most construction equipment (e.g. backhoes) where a lever is directly linked to each joint on the device. In robotics it is more common to find a teach pendant that uses two buttons for each joint to provide positive and negative incremental change. These devices are not discussed in any of the control methods of Section 2.4.1.2 because controlling a robot with one of these devices simply requires the conveyance of the information to the robot controller.

2.4.1.1.2. Kinematic Replica

Kinematic replica MCs are very common in robotic teleoperation tasks where the robot is intended to mimic the operator’s movements exactly. They are kinematic arms usually modeled after a robot arm for which they were intended to serve as a master. These types of controllers, such as the TeleOperation Systems (TOS) controller (Figure 2.3b) and robot (Figure 2.4), are often simple to implement because scaled joint-to-joint control signals (see Section 2.4.1.2.1) can be used.

![Figure 2.4: Teleoperation Systems (TOS) Robot](image)
These controllers can also be used more widely as delta or Cartesian controllers by performing kinematic analysis on them. However, this may introduce singularity issues. The decoupled geometry of the controller and the robot may make controlling the robot more difficult. However, despite the drawbacks, using the kinematics of the controller provides much more reusability of the MC.

2.4.1.1.3. Universal

Universal manual controllers are kinematic devices designed to optimize performance within a particular work envelope. Like the RSI controller shown in Figure 2.3c, some of these devices are intended for use in delta control (see Section 2.4.1.2.3) due to their limited range of motion. Others are intended for Cartesian control (see Section 2.4.1.2.2) like the 9-string 6-DOF force reflecting joystick (Figure 2.5) constructed by Lindemann et al. which calculates end-effector position from the parallel information provided by the nine strings [34]. This manipulator also uses the nine strings and three pneumatic actuators to provide force feedback information (see Section 2.4.1.3). While most universal manual controllers are designed for either delta or Cartesian control, they often can be used in either of these control schemes.
2.4.1.4. Delta

Delta Controllers have a small range of motion and a fixed base that make them suitable only for delta control (see Section 2.4.1.2.3). They also have a fixed zero position allowing them to easily supply a delta value; although it must usually be scaled for the application. The most common type of delta controller is a simple push button interface like the teach pendant of most industrial robots. More useful, however, are devices like the three axis joystick (Figure 2.3a) and 6-DOF delta controllers like the Magellan (Figure 2.6) that allow tactile feedback as to the commands being given to the system.

![Logitech Magellan 6-DOF Delta Controller](image)

2.4.1.2. Control Methods

The geometry of many manual controllers provides some options as to how the input should be used to control a manipulator. The following sections present three methods for controlling a robot in end-effector space based on the input from MCs. These methods have parallels in velocity and torque control; however, only position control of robots is presented here.

2.4.1.2.1. Joint-to-Joint

Joint-to-joint control is limited to kinematic replica manual controllers (see Section 2.4.1.1.2). The controller and robot must have similar geometry so a simple conveyance of the controller’s joint positions to the manipulator will provide end-effector positions that are consistent with the controller positioning. This type of
control is easy to use when a controller and robot are matched because no kinematic calculations need to be performed. However, despite the ease of use that joint-to-joint control provides, there are many drawbacks. Chan and Dubey present a detailed explanation of the drawbacks to joint-to-joint control in [12] from which the following list was compiled.

A. Scaling of motion and force between the slave and master is fixed by physical parameters and cannot be altered
B. The master’s reference position is fixed in Cartesian space
C. Impedance control cannot be implemented for transmission of forces
D. Redundancy is inconvenient as the master must also be redundant leaving the operator to control the extra degrees-of-freedom

2.4.1.2.2. Cartesian
Cartesian control or generalized bilateral control uses the end-effector position of the controller to determine the actions of the slave robot at its end-effector. This decoupling of the geometries allows scaling, fixturing, filtering, impedance control, coordinate transformation, dead-bands, etc. Many of these functions are described in [35]. However, due to the complicated transformations required, Cartesian control may include computational singularity problems and time delays.

2.4.1.2.3. Delta
Delta control simply consists of incremental position changes given to the control software from the manual controller. Usually, this delta information is drawn from a delta controller (see Section 2.4.1.1.4); however, it can also be extracted from other controllers by determining the distance traveled since the last information exchange. Though delta control appears to be relatively simple, mathematical complications in the combination of rotation representations can affect the results.

Delta motion in translation is simple to handle by adding the delta to the current position; however, orientation can be much more complicated. A single
orientation change could be represented in fixed orientation angles, Euler angles, quaternions, rotation matrices, etc [14]. How this value is combined with the current position can cause different results. For instance, Euler angles could simply be added, but the combination of equivalent rotation matrices requires matrix multiplication and does not necessarily return an equivalent result as adding the Euler angles. This is not the focus of this report, but it is something that must be considered in all robot control software. For more information on orientation combination, see [14] and [54].

2.4.1.3. Force Reflection

Many manual controllers, particularly kinematic replica MCs, have force reflection capability. This means the joints are not only used for position control of the robot but they can also be actuated so that the user can feel the equivalent forces to those applied to the robot. This is essential for true understanding of the interaction between the robot and its environment, especially when this contact is critical to the application [11]. In this situation reflection of the contact forces gives the operator the feel of actually performing the physical task.

2.4.2. Other Interface Tools

Manual controllers are a powerful tool in manipulating robots, yet the communication is at a low level. In order to bring the level of robotic commands up from that which manual controllers can provide, some automation must be included in the system. Once the robot has a list of commands that it can automatically accomplish, there are many ways to communicate the desired command to the robot. Sometimes the robot can make the appropriate decision without human input by using other inputs such as computer vision (see Section 2.4.2.1). Other times, spoken commands may be used to elicit a response such as “STOP” (see Section 2.4.2.3). More often than not, however, commands require a combination of technologies to successfully complete a complex task. The following sections present an overview of semi-autonomous robot interface technologies.
2.4.2.1. **Computer Vision**

Computer vision is a broad topic covering many applications. Basically, it deals with image analysis for recognition of expected images. This recognition could be the identification of specific objects or shapes from a library of known shapes or it could just be the recognition of movement and the leading edge of the moving object. No matter what the system is ‘looking’ for, the algorithms are complex, and dealing with noise and uncertainty can cause crippling delays. Even a seemingly simple task like recognizing the two-letter state code on handwritten mail can become overwhelming. A 95% success rate is considered good [51]. However, as technology continues to progress, computer vision will play a continuously greater role in everyday life. For an interesting and detailed look at the many facets of machine vision, see Davies’ book [15]. Within the robotic interface, machine vision plays several roles. It can be used as a tool for the robot in execution of automated tasks, or it can play a more dynamic role in the interface as discussed in the next section and in Section 2.4.2.4.

2.4.2.2. **Gesture Recognition**

Gestures are a natural method of communication among humans and thus have received much attention as a possible augmentation to the human-computer interface. Two principle methods are used for reading gestures from a user: glove-based systems and computer vision-based systems. Glove-based systems use a glove with sensors providing angles of various joints in the hand. Similar devices also exist for full-body data retrieval for posture analysis. These systems require the user to wear cumbersome, wire laden devices that can hinder the naturalness of the communication, yet they provide more reliable data and are less expensive than vision-based systems [23]. Vision-based systems analyze images of the human in order to identify gestures in much the same way as other objects are identified by computer vision (see Section 2.4.2.1). These systems are more desirable because the
interface is non-contact; however, even more variables are added to the computer vision pattern recognition problem, such as skin color, which can cause decreases in recognition success.

Gesture recognition deals with the identification of both static and dynamic gestures. Static gesture recognition includes the recognition of predefined body postures or hand positions requiring only pattern matching to determine the desired interaction. Dynamic gestures might include the movement of an arm to indicate the desired movement of an object. This adds complexity to the identification process, such as the difficulty of determining the beginning and end of a gesture. Also, two-dimensional or three-dimensional models can be used, each presenting their own benefits and disadvantages [41]. All of these variables lead to complex and only somewhat reliable systems with relatively low success rates. Eickeler et al. present a system for identifying 24 gestures with a success rate of 93% [18]. However, the dynamic nature of gesture recognition does provide learning capability [22] which means a system struggling with specific gestures can be retaught the gestures or an additional gesture can be added to the training set. For a detailed review of gesture recognition technology, particularly in computer vision-based systems see [41].

2.4.2.3. Acoustic Speech Recognition

Like gesture recognition, speech recognition involves processing a signal to match with known signals. In this case, the signal is a spoken command. There are many difficulties inherent to acoustic speech recognition. The following list was compiled from [38]:

A. A separator does not necessarily exist in spoken language to separate words like a space separates words in written language.
B. Any elementary sound is affected by its surrounding elementary sounds.

Movements in the vocal apparatus to prepare for the following sounds
affect the current sounds being made. Also, position in a sentence can affect a sound as in the inflection given to the end of a question.

C. Variability is inherent in human speech. This includes intra-speaker variability due to singing, shouting, stress, illness, etc, as well as inter-speaker variability due to age, sex, tone, etc. Also, even with an identical spoken signal, variations in background noise, microphones, etc. can cause a variation in the recorded signal.

D. The amount of processing required to resolve the variability due to B and C can become enormous.

E. The voice signal carries more information than the computing system may be interested in, such as the speaker’s sex and mood. A system must filter out the data needed.

F. Rules are not defined for formalizing the information contained in a vocal signal. Particularly in fluent speech, the levels of decoding become dependant.

Many tools and approaches are used for acoustic speech recognition. A detailed discussion of these techniques is beyond the scope of this report, but an overview of frequency analysis, linear prediction analysis, vector quantization, pattern classification techniques, time normalization, statistical techniques, neural networks, sub-word units, and knowledge-based techniques can be found in [3]. The technique that dominates speech recognition research is statistical. Specifically, the Hidden Markov Models (HMM) are used. HMMs can recognize communication because they allow the user to teach various words separately from their context. Then, a state based system can be set up that defines the allowable ordering of the words. This is all done probabilistically by defining the likelihood that the system will move from its current state to the various other states [3]. A simple example of an HMM system is described in detail in [8].
2.4.2.4. Hybrid Speech Recognition

Though acoustic speech recognition has gotten fairly accurate and continues to improve, several methods exist for improving speech recognition by adding other resources. These methods are particularly useful in environments where acoustic speech recognition struggles, such as noisy environments. The most common hybrid speech recognition model uses video of the speakers face for lip-reading. Bregler et al. present techniques for learning lip movements [9] and Kaynak et al. discuss fusion of the lip-reading and acoustic data as well as experimental results [32]. Another method for improving acoustic speech recognition uses an ElectroGlottoGraph (EGG). An EGG uses transducers on the speaker’s neck to measure changes in impedance across the vocal folds. The use of an EGG with acoustic speech recognition is presented along with experimental results in [17].

2.4.2.5. Graphical User Interface

The graphical user interface (GUI) is well known to anyone who has used a computer in recent years. GUIs are based on the WIMP (windows, icons, menus, and point-and-click devices) paradigm, which is easy to learn and has become relatively standardized by Macintosh and Microsoft GUI designs. However, pure WIMP based interfaces are slow and require memorization of keyboard shortcuts for efficient operation [45]. Despite drawbacks, GUIs are powerful interface tools and are a significant contributor to human-computer interaction.

Traditionally, GUIs are platform dependant meaning that GUI code developed for one operating system will not work on other operating systems. Though this limits reusability of GUIs, it has not hurt their popularity. Many people, however, have been working on approaches to fix this reusability problem. Guthrie presents many commercial applications for platform-independent GUI development [21]. These applications are typically based on a common language and provide support for some subset of operating systems. A newer approach to this problem that appears to
provide a better solution is the XML based User-Interface Markup Language (UIML) [1]. UIML is intended to provide a GUI description that can be displayed according to an operating system’s guidelines. This is similar to the way HTML provides the same interface to users of various web browsers on different operating systems.

2.4.3. Interface Software

The interface software is the part of the HRS that deciphers the commands from the interface device and calls the appropriate commands on the robot controller. At this level, even the best designed HRSs typically get some intermingling of the interface software and the robot control software. The interface software designer is required to know what functions are available on the robot controller and how to use them. Often, the interface designer must also deal with various programming languages and the processes for communicating between the languages. An effective separation of the interface software and robot control software would facilitate better interface design because the designer could focus on one part or the other. This would parallel hardware drivers used for computer hardware which act as a buffer between the hardware device and the operating system giving them a standardized interface through which to communicate [4].

Though many generalized languages for robot programming have been developed, none have attained widespread use. A number of factors prevent the robotics industry from adopting a standard from the available languages, including integration difficulties and extensibility. Adoption of a standard would provide reusability among interfaces and controllers using the standard; however, integration with older devices would be difficult. Also, it is unlikely that a single programming language has been developed to cover all of the possible communication scenarios that might be faced.

The following subsection presents various existing robot programming technologies. These are particularly applicable to robotic systems that do not have
dynamically defined tasks but instead have fixed environments and predefined tasks. Particularly in industry, robots are given a single task that is repeated indefinitely until a new task is assigned. The interface between the user and robot is often brought down to the programming level for this type of robot control. The two major programming methods for robots are on-line and off-line programming.

2.4.3.1. On-Line Programming

On-line programming uses the robot as the programming tool. The operator directs the robot using available interface tools and trains the robot to complete the desired tasks. Robot training is the most basic and most common type of robot programming. Since the first robotic systems were introduced, this method has been used for a wide variety of applications [40]. Training methods, however, have drawbacks in that training can take too long or the application can be too complex to describe with simple training. Also, on-line programming is typically robot specific. Therefore, a user in each case must learn the interface to the robot, how the training is performed, and how the motion parameters are recorded.

2.4.3.1.1. Point-to-Point Training

Point-to-point training requires the operator to specify various points in the desired path along with parameters for the trajectory between each point. This can be performed by manual movement or teleoperation. Manual movement is not common because it requires certain physical conditions of the robot and produces inaccurate results [40]. Teleoperation allows the user to move the motors to the desired position using the available interface. For either of these modes of training, additional information must be given to the program, such as timing, speed, trajectory type, etc.

Early point-to-point training simply moved the actuators to the specified positions. This has typically been replaced with the option for straight line paths between the points. Also, some training-based languages allow for smooth trajectories by not stopping at each defined point but instead rounding the corner to the next point
Point-to-point training is an effective and relatively accurate method for programming robots when the length of time a robot will spend performing a task is significantly longer than the length of time it takes to train the robot. Thus it is frequently used for industrial robots.

2.4.3.1.2. Recorded Trajectories

When tasks get too complicated to adequately describe them with point-to-point training, recorded trajectories can be used. With recorded trajectories the user moves the arm in the desired path either manually or by teleoperation. The path is recorded exactly as the user gives it, so this method is often referred to as ‘teaching by doing.’ Recorded trajectory programming languages do not typically require the entire task to be taught; instead, the user defines the important parts of the task, and the programming language provides an interpolation between defined trajectories. Due to this functionality, these trajectories can be edited by subsequent training providing the ability to fix mistakes without starting over [40]. This method is still inefficient and suffers from many of the same drawbacks as point-to-point teaching.

2.4.3.2. Off-Line Programming

Off-line programming separates the robot execution from the programming. Typically, some type of model, either mathematical or graphical, accompanies off-line programming to verify that the program will not cause critical problems. The programming can be textual (similar to computer programming) or it can use simulations to program in a similar manner to on-line programming. Like on-line programming, most off-line programming languages are manufacturer specific; however, a big push has been made recently to provide general robot programming.

2.4.3.2.1. Specialized Robotic Languages

These languages are designed specifically for the control of robots. The commands are typically motion commands, and logic statements are minimal [33].
Specialized robot programming languages are very common in the robotics industry. These languages resemble the languages used for on-line programming with position parameters given numerically rather than being taught to the robot. One common specialized robotic language is VAL, which was developed to optimize trajectory planning but left out the programming constructs required to perform truly complex tasks [47].

2.4.3.2.2. General-Purpose Languages

General-purpose languages refer to complex languages developed by robot manufacturers to facilitate robotic commands in conjunction with common computer programming constructs. These languages include similar motion commands to the specialized robotic languages, but they also include more advanced programming such as conditional statements, loops, functions, etc. Nearly all robot manufacturers have their own Robot Programming Languages (RPL) for use with their robots. These languages are often similar to some existing computer programming language, but they contain robot specific functionality. Examples of this type of language are KAREL and VAL-II [19].

2.4.3.2.3. Existing Computer Languages

A powerful method of robot programming has developed in the use of existing computer programming languages. Typically, a robot programming language will be designed and implemented for an existing language using libraries from which robot functions can be called. This method offers many benefits over the previous examples. The following list of benefits was compiled from [33]:

A. Most robot program code is not specific to robots, such as initialization, logic, and branching. Existing languages have been developed to efficiently handle these things.
B. Major existing languages provide robust support for input/output control, interface development, and debugging, which contribute to the ease of programming and the addition of powerful code not specific to robotics.

C. The syntax is familiar to a user of the existing language and can, therefore, be used by a much broader group of programmers.

D. The language is likely to be more stable than typical RPLs which become obsolete when the manufacturer produces a new iteration.

This approach to robot programming effectively creates an Application Program Interface (API) to the robot control software. Applying an API to various robots provides consistency in programming between the robots. A programmer familiar with the API can more efficiently build new programs using the API. One particularly beneficial result of generalized robot control software development is the establishment of a common API for all robots using the control software. Control software is discussed in Section 2.4.4.

2.4.3.2.4. Other Programming Methods

There are many other robot programming methods which have received research mention. Some of these seek to define RPLs in non-traditional ways. The US military has modified preexisting languages for numerical control of lathes, mills, etc. for use with robots. PostScript by Adobe Systems and other languages developed for specific, non-robotic applications have also been modified for robot control. Frenger presents these in detail in [19].

Other programming methods seek to raise the level of the commands given by the programmer. Two common approaches to this type of language are object level languages and objective level languages [40]. Object level languages deal with the movements of objects, and the robot is intended to automatically perform those movements. Objective level languages are even less specific by giving a desired result, and all of the intermediate steps are left for the intelligence of the robot. Both
of these types of control rely heavily on automation and thus begin to look much like a textual form of some of the interface types discussed in Section 2.4.2. A common example of this type of programming is packages developed exclusively for welding that have special functionality to make programming for welding much easier [46].

2.4.3.2.5. Graphical Programming

Due to its very nature, off-line programming often includes a graphical simulation of the robot. At the very least, graphical simulations allow the programmer to verify that the program is not unstable. However, more frequently these systems are being developed with graphical programming interfaces that allow the user to focus on the work that the robot is to perform rather than the underlying programming language. Any of the programming language types discussed in the previous sections can be fitted with a graphical programming interface. These tools provide quick and easy off-line programming for simple, common tasks, yet they can limit the functionality of some languages.

2.4.3.2.6. Language Translation

As more general RPLs are developed with or without graphical interfaces, integration with existing robots becomes a critical issue. There are two ways of integrating a newly developed language with an existing robot. The first is replacing the entire control software package for the robot (discussed in Section 2.4.4). The second is language translation. Until recently, this has been accomplished using post-processing of the RPL to translate its internal representation of the data into the native robot language of the robot being implemented [53]. This requires post-processors for every language that is to be supported. Furthermore, any time these languages change, the general RPL developer must update the post-processor.

Mamas and Kontogiannis have demonstrated an XML based representation of computer programming languages [37]. They propose a method for using a language’s grammar to develop an equivalent XML representation that would
facilitate portability. Vollmann and Sett have built on this concept, proposing a similar method to facilitate language translation [53]. If a general RPL could prove itself to be widely usable, then the producer could get robot manufacturers to provide the XML description of their language, and thus remove the translation aspect from the concern of the general RPL developer.

2.4.4. Robot Control Software

The next part of the HRS is robot control software which provides the functionality for controlling the robot hardware. This will always include software for hardware interfacing. It will usually also include various other functions, such as: limits checking, kinematics, path planning, teach-points, etc. Robot control software is typically proprietary for industrial robots and requires the use of a unique programming language. It is also very rarely distinguishable from the software interface, but it is feasible and desirable to consider these components independently so the computation for controlling the robot is treated separately from the interface to that functionality.

2.4.4.1. State Machine Representation

Robotic systems and their control software can be modeled as state machines. A state machine is any system consisting of a list of distinct states connected by transitions that are caused by events [50]. Perusing any robotics manual makes it clear that robots fall in this category. There are distinct steps that must be followed to transition the robot from its initial state to the desired control state. Figure 2.7 shows an illustrative example of how a simple robot might be represented in a state diagram. This robot transitions through a simple initialization sequence to get into its teleoperation state where it remains until the robot is powered down. The robot also contains no error states for simplification. Though an actual robot would have many more states than this example, it can be seen that each of these states could be inserted into a similar diagram.
Methods are being developed that allow for easy definition of the states and transitions found within state machines that can then be automatically converted into operation code. Martin presents one such method using C++ [39]. Defining a robot control language in this manner would produce clear, robust, and extensible code. Defining the interface to a robot relies heavily on the ability to model the robot as a state machine or set of state machines (see Chapter 4).

2.4.4.2. Virtual Robot Controller

Virtual robot controller development is an effort of a conglomerate of automotive and robotic manufactures attempting to develop the Realistic Robot Simulation (RRS) system [5]. The RRS system is intended to create a standard interface for robotic simulation packages so that control software used on robotic systems can be integrated directly into the simulation package. This would correct common problems in off-line programming that arise from variation in motion planning and kinematics used in simulation packages and on the actual robot. These differences can cause significant error in execution on the factory floor [7].

2.4.4.3. Open Architecture Controller

The design of an open architecture controller for robotics is an ongoing research topic for many research groups. Though the actual definition of ‘open architecture’ is not generally agreed upon by these groups, the goal of open architecture controller research programs can be generalized as the development of
common robot control components for implementation on any robot [43]. Of course, new robot control software also adds a new non-standardized interface for robot control. Some existing projects in this area include OSCAR, OROCOS, URC, and others [30][10][33]. Each of these is intended to work on all robots and provide extensive functionality.

Operational Software Components for Advanced Robotics (OSCAR) has been under development for nearly ten years at the University of Texas Robotics Research Group (RRG) [29]. As such, it is of particular interest for this research and has been used for much of the planning and testing of this research. OSCAR is a robot control software development framework that includes kinematics, dynamics, criteria, obstacle avoidance, etc. OSCAR easily encompasses the functionality of the typical industrial robot control software and matches or exceeds most other research control software in functionality [31]. For this reason, using OSCAR based controllers for development of this report is not a limiting decision.

2.4.4.4. Generated Controller

Current work at the RRG is attempting to develop automatic robot control software generation based on OSCAR [25]. This work will allow a robot manufacturer to specify the robot hardware, inputs, outputs, goals, etc. to a generator, and a robot controller will be developed. This approach uses an open architecture controller as discussed in the previous section but allows the control libraries and robotic language to be customized to the robot and application. This does not, however, affect the generality of the open architecture approach because the generator would create a subset of the language that still follows the rules of the overall system.

2.4.5. Robot Hardware

Robot hardware is the final part of the HRS system. It includes the actuators, sensors, tools, etc. at the robot level. This could include anything from one degree-of-freedom (DOF) machines with no sensory feedback to hyper-redundant manipulators
with many types of feedback. The following sections present robot types, sensors, and tools along with their impact on the HRS.

2.4.5.1. Robot Types

Each type of robot provides unique requirements to an HRS. Mobile robots, for instance, have actuators that may act independently; whereas, the joints on serial manipulators affect the position of other joints and typically have limited ranges of motion. However, in Cartesian space, a mobile platform acts no differently than a limitless planar serial chain with prismatic joints.

Redundancy in a system adds significant complexity when controlling the system in Cartesian space. Redundancy is when a system has more inputs than controlled outputs. This results in multiple ways to achieve a desired position and requires some definition of how to select one of the calculated solutions. Additionally, workcells can combine all of these systems into a large system of manipulators and mobile robots able to work independently, but this often requires synchronized motion to successfully complete a task. Again, as this report is a generalized approach to interface definition and generation, all of these situations should be able to use the methods presented.

2.4.5.2. Sensors

Sensors and sensor feedback are a difficult challenge in the development of an HRS because the number and types of sensors cannot be feasibly identified before hand. This would test the extensibility of a generalized robot controller. Common sensors used in robotics include: position, velocity, force, torque, and temperature. Also, other types of feedback such as video can exist.

2.4.5.3. Tools

Tools are an extremely important part of robot hardware. Many robots are able to change tools as well as control the active tool. Much like sensors, tools cannot
be defined a priori. Therefore, extension of a generalized robot controller must be possible. The following table gives some of the different types of tools.

<table>
<thead>
<tr>
<th>Tool Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-state</td>
<td>On/Off</td>
<td>Welding torch</td>
</tr>
<tr>
<td>3-state</td>
<td>Forward/Reverse/Off</td>
<td>Drill</td>
</tr>
<tr>
<td>n-state</td>
<td>Discrete set of states</td>
<td>5-speed saw</td>
</tr>
<tr>
<td>Continuous rate</td>
<td>Infinitely adjustable speed</td>
<td>Variable speed grinder</td>
</tr>
<tr>
<td>Continuous position</td>
<td>Infinitely adjustable position</td>
<td>Gripper</td>
</tr>
</tbody>
</table>

Table 2.1: Various Tool Types

2.5. Summary

Many powerful interface tools exist in robotics use. For direct manipulation of robots, manual controllers of various geometries and control methods can be used. These techniques provide good control of robots while separating the operator from the environment. Though manual controllers are a powerful tool, they can rarely control a system independently. Typically, a GUI or other automation interface must also be included.

Many techniques have been developed to bridge the gap between teleoperation and automation in order to improve the communication between human and robot. However, these systems are still robot specific because of the wide variety of robot programming languages and the inability to establish a standard. Furthermore, these systems cannot cooperate to perform even more complex tasks on a single robot.

This chapter demonstrated that while the search for a standard control software and interface is not futile, perhaps what is needed is a better way to integrate interfaces. Techniques to thoroughly define the interface to a device and emulate a new interface based on selection of features from the interface definition are presented in this research. Particularly, the robotics domain is used as an example of domain analysis, and various examples of implementation are presented.
3. Interface Syntax and Semantics

There are two major components to fully describe a system’s interface. The syntax of an interface specifies the content of the interface. It is defined by the valid commands and does not address how validity changes with time. Similarly, the English language has a structure defining the construction of sentences. This does not address what the sentences mean or whether the sentences even make sense. The interface semantics defines runtime restrictions on the commands of the interface. These restrictions are based on physical, time-based, and conditional relationships among the properties and actions defined by the syntax.

3.1. Interface Syntax

The syntactic structure of an interface is defined by analyzing several components. It is clear that an interface consists of incoming and outgoing information. Outgoing information is modeled as current values for specific properties of the system and represents the current state of that property. Information coming in is modeled as an action on a property. As such, an interface is defined by this collection of properties and actions.

One of the techniques for implementing product-line architectures is Feature Oriented Programming (FOP). FOP involves modeling a domain in terms of its features where features are somewhat independent building blocks of the domain.
These features can then be combined to create distinct products belonging to a product-line (a family of related systems). This work applies FOP principles to interface generation. The first step is the specification of the interface for each feature. This is done by specifying the properties that the feature supports and the actions that can be performed on these properties. These properties and actions are a subset of the overall domain’s properties and actions. As features are composed in FOP, the interface to these features (specified in terms of domain properties and actions) are also composed. This research develops a systematic technique for describing the interface that lends itself to algebraic composition, reduction, and optimization.

This chapter fully describes this technique so that a domain expert might be able to perform the interface analysis for a well-understood domain. Two notations are used in this chapter and the rest of this report to specify these components. The first is an algebraic specification that is necessary for the interface optimization presented in Chapter 5. This representation does not need to be developed by the domain expert; it can be inferred from the more easily understood textual specification based on Context Free Grammars (CFG) for defining programming languages. The CFG based specification provides a straightforward approach that is easily understood by humans and computers. Also, the CFG specification can be easily translated into other representations by defining the translation of small components of the CFG interface as in Section 5.2.

### 3.1.1. Algebraic Interface Specification

The algebraic specification of interface syntax presented in this section is laborious, but is required for algebraic optimization of the interface. As defined, the overall interface for a domain can be modeled in terms of properties and the actions associated with each property. The domain properties are defined by the list $P$ where each element, $p' \in P$, defines a single property of the domain. To complete the
definition of the outbound communication from the system, each property defines its possible property values in the list $P^i_v$ given by

$$P^i_v = \{v_1^i, v_2^i, \ldots, v_n^i\}$$

where $v_j^i \in P^i_v$ is the $j^{\text{th}}$ value in the list of values for property $p^i$. Each value can be discrete as in an enumeration or a value can represent numerical data. In implementation, the algebraic representation of property values would be a hook into the human readable specification presented in the next section, because the algebra is designed solely for reducing and optimizing the human readable specification.

Next, the list $A^i$ defines the actions associated with the property $p^i$. As presented in Section 3.1.4, the interface has been defined such that, with properly identified properties, the only actions necessary are get and set. While the list of actions is not explicitly restricted, additional actions should only be added in special circumstances where get and set are insufficient. At this time, no such circumstances have been identified but the interface specification does not assume that none exist. As such the list of actions is given by

$$A^i = \{a\_\text{get}^i, a\_\text{set}^i\}$$

Each action then identifies a set of values that represent the arguments to that action. In general, no explicit restrictions are given to the definition of the action arguments; however, the get and set actions do have explicit restrictions that must be adhered to. The get action is intended solely to request information from the system and thus requires no arguments, while the set action is used to set the system’s state to one of its possible values. Therefore the get and set actions are defined by

---

1 Within the mathematical notation, capital letters are used to represent lists and lower case letters represent an item in a list. Indexing into lists is performed via subscript except for the superscript $i$ applied to various symbols. This superscript identifies the symbol as having a direct relationship to the $i^{\text{th}}$ property of the domain. The exact nature of the relationship can be determined by the symbol itself. In the case of property values and action arguments, the capital V subscript denotes the value list associated with the single property or action to which it is attached.
The set action is a subset of its associated property’s possible values because the action need not have the ability to command the system into all of its possible states. With the properties and actions for a given domain defined, the domain itself can be represented by a list, \( D \), of tuples containing the property values and action arguments for \( p^i \) where each element of the list is given by

\[
d^i = (p^i, A^i)
\]

With the interface to a domain fully described by its properties and actions, FOP principles are applied. As mentioned before, the FOP paradigm is used for software development and algebraic optimization. In FOP, a product instance is modeled as a set of features, \( F \), where each feature \( f_k \in F \) applies to the domain \( D \).

The interface to a feature is defined in terms of domain properties it supports. This list of feature properties \( \hat{P} \subseteq P \) defines which properties in \( P \) apply to the feature\(^2\). At the domain level, a property, \( p^i \), has a set of values \( p_v^i \) as defined by Equation (3.1). A feature that supports property \( p^i \) need not support all of its associated values. Therefore, the feature’s set of values for \( p^i \) is given by

\[
\hat{p}_v^i \subseteq p_v^i
\]

Similarly, each feature property, \( \hat{p}^i \), has an action list that is a subset of the domain level actions for property \( p^i \). This is represented as

\[
\hat{A}^i \subseteq A^i
\]

This allows the feature to apply only a subset of the domain property’s associated actions and the arguments of the individual actions can be a subset of the associated domain action’s arguments. For instance, the set action is given by

\[ a \_ set^i_v \subseteq p_v^i \]

\[ a \_ get^i_v = \emptyset \]  

\[ a \_ set^i_v \subseteq p_v^i \]  

---

\(^2\) \( A^\land \) indicates that a particular property, property value, or action applies to a given feature.
\[ \hat{a}_{set_i} \subseteq a_{set_i} \] (3.7)

However, the definition of the set action requires that the feature action’s arguments also be a subset of the associated feature property’s values so the set action is redefined as

\[ \hat{a}_{set_i} \subseteq \left( a_{set_i} \cap \hat{p}_i \right) \] (3.8)

Finally, with the interface to the individual features of the product-line defined, the interface to a product instance within the product-line can be defined as a subset of the interface to all the features in the product-line as

\[ I \subseteq F \] (3.9)

3.1.2. Context-Free Grammar Interface Specification

In computer science, a program can be thought of as a long sentence. Context-Free Grammars (CFGs) are mathematical sets of rules that define the syntax of these sentences [13]. An interface can also be approached in this same manner by defining the sentences for inbound and outbound communication. This provides a straightforward approach to generating an interface specification that is both human and machine readable. Also, with some naming conventions defined for the development of the specification, the optimizer can easily extract the algebraic representation for reduction and optimization. Lastly, the CFG based specification can be easily translated into various representations by specifying the translation for each component.

For instance, a simple command from the alarm clock interface is defined as

\[ set :: alarm \_ state :: on \] (3.10)

The CFG grammar could be translated into a C++ program like that shown in Table 3.1 by defining the translation of the individual parts. The grammar itself specifies the interface to the alarm clock given and is represented in C++ by the alarm_clock class and the functions specified. The interface could then be used within the main function of the program to perform the action given in Equation (3.9).
### Interface Representation

```cpp
class alarm_clock{
public:
    alarm_clock()
    {  // perform initialization
        current_state = off;
    }
    enum alarm_state{on, off};
    void set(alarm_state s)
    {  // perform error checking
        // execute change
        current_state = s;
    }
private:
    alarm_state current_state;
};
```

```cpp
int main()
{
    alarm_clock myClock;
    myClock.set(alarm_clock::alarm_state::on);
    return 0;
}
```

### Usage

Table 3.1: Sample Alarm Clock Program

Similarly, the same interface could be represented as an eXtensible Markup Language (XML) schema as shown in Table 3.2. The schema represents the interface definition and is translated from the grammar. The usage shows what the command of Equation (3.9) would look like in an actual communication. The process of producing interface representations, including C++ and XML, is addressed in Section 5.2.

<table>
<thead>
<tr>
<th>Interface Representation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xs:schema targetNamespace=&quot;alarm&quot;</code>&lt;br&gt;<code>elementFormDefault=&quot;qualified&quot; xmlns=&quot;alarm&quot;</code>&lt;br&gt;<code>xmlns:xs=&quot;http://www.w3.org/2001/XMLSchema&quot;&gt;</code>&lt;br&gt;<code>&lt;xs:element name=&quot;set&quot;&gt;</code>&lt;br&gt;<code>    </code><a href="">xs:complexType</a><code>&lt;br&gt;</code>        <code>&lt;xs:sequence&gt;</code>&lt;br&gt;<code>            </code>&lt;xs:element name=&quot;alarm_state&quot;&gt;<code>&lt;br&gt;</code>                <code>&lt;xs:simpleType&gt;</code>&lt;br&gt;<code>                    </code>&lt;xs:restriction base=&quot;xs:string&quot;&gt;<code>&lt;br&gt;</code>                        <code>&lt;xs:enumeration value=&quot;on&quot; /&gt;</code>&lt;br&gt;<code>                        </code>&lt;xs:enumeration value=&quot;off&quot; /&gt;<code>&lt;br&gt;</code>                    &lt;/xs:restriction&gt;<code>&lt;br&gt;</code>                &lt;/xs:simpleType&gt;<code>&lt;br&gt;</code>            &lt;/xs:element&gt;<code>&lt;br&gt;</code>        &lt;/xs:sequence&gt;<code>&lt;br&gt;</code>    &lt;/xs:complexType&gt;<code>&lt;br&gt;</code>&lt;/xs:element&gt;<code>&lt;br&gt;</code>&lt;/xs:schema&gt;`</td>
<td><code>&lt;set xmlns=&quot;alarm&quot;&gt;</code>&lt;br&gt;<code>    &lt;alarm_state&gt;</code>&lt;br&gt;<code>        on</code>&lt;br&gt;<code>    &lt;/alarm_state&gt;</code>&lt;br&gt;<code>&lt;/set&gt;</code></td>
</tr>
</tbody>
</table>

Table 3.2: Sample Alarm Clock XML Interface

45
The interface used in these examples is a simplified version of the actual interface that would be generated, but is representative of the results. The details of the interface generation process are presented in Chapter 5. The steps involved include defining the interface for each feature in a product line, combining the interfaces for a set of features that represent a product instance, reducing and optimizing this combined interface, and translating this interface to a specific interface representation. At this point, this is all done manually, but the approach is defined in a manner that could be implemented automatically. The following two sections introduce CFGs in general and the approach for using it for interface generation. The particular CFG syntax used in this research is Extended Backus-Naur Form (EBNF); its syntax is given in the following section.

### 3.1.2.1. Context-Free Grammars

Context-free grammars are used to specify programming languages. They contain two basic components: terminals and non-terminals. Terminals are characters or strings in the language. The final sentence is a combination of many terminals. Non-terminals are abstractions of the language used by the language writer to provide structure. The entire language is defined in terms of the starting non-terminal. This report uses EBNF which has been standardized by the International Organization for Standardization in ISO/IEC 14977 [25]. A summary of the core operators used in EBNF is given in Table 3.3. The operators are presented in precedence order with the highest precedence first.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>repetition symbol: defines a specific number of occurrences</td>
</tr>
<tr>
<td>-</td>
<td>except symbol: allows exclusion of an item from a option list</td>
</tr>
<tr>
<td>,</td>
<td>concatenate symbol: separates items in a sequence list</td>
</tr>
<tr>
<td></td>
<td>separator symbol: separates items in an option list</td>
</tr>
<tr>
<td>=</td>
<td>defining symbol: defines non-terminals</td>
</tr>
<tr>
<td>;</td>
<td>terminator symbol: indicates the end of a rule</td>
</tr>
</tbody>
</table>

Table 3.3: EBNF Grammar Symbols
In addition to the core operators, several bracket pairs are defined. Bracket pairs surround characters, strings, and sequences to give special meaning. They are also used to interrupt the default precedence of the operators. While not all of the bracket pairs are used in this report, all of the defined bracket pairs for EBNF are included for completeness. They are given in Table 3.4.

<table>
<thead>
<tr>
<th>Bracket Pair</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>' ' or “ “</td>
<td>quotes: signify a terminal</td>
</tr>
<tr>
<td>[]</td>
<td>option symbols: sequences can appear once or not at all</td>
</tr>
<tr>
<td>{}</td>
<td>repeat symbols: sequences can appear zero or more times</td>
</tr>
<tr>
<td>()</td>
<td>group symbols: sequences are grouped overriding precedence</td>
</tr>
<tr>
<td>(* *)</td>
<td>comment symbols: sequences are not part of the grammar</td>
</tr>
<tr>
<td>??</td>
<td>special sequence symbols: allow extension of EBNF</td>
</tr>
</tbody>
</table>

Table 3.4: EBNF Grammar Bracket Pairs

As discussed previously, EBNF is intended to define the lexical definition of a language. Since this research does not intend to create a language, the resulting lexical definition of the interface must be taken as a pseudo-language. It provides all of the syntactic structure to support translation into any desired representation.

3.1.2.2. Interface Generation Grammar

As presented in Section 3.1.1, concrete relationships exist among the various components of an interface specification. In order to maintain these relationships, the EBNF specification uses some naming conventions. The primary non-terminal in EBNF is the starting point. In interface generation, the starting point is the interface \( I \); it will be identified by an arbitrary name followed by the “_interface” extension. Interfaces include a set of feature interfaces from the set of product-line features. They are defined by a subset of the product-line features \( I \subseteq F \), so the interface definition consists of a list of features which have arbitrary names followed by the “_feature” extension. Each uniquely named feature is an element of the list of features \( f_k \in F \). Interfaces for features are defined using properties and actions where each
set of feature actions ($\hat{A}'$) is associated with the feature property ($\hat{p}'$). The set of actions is identified by the property name followed by the “_actions” extension.

The set of actions is defined by a list of actions where each action is identified by the property name followed by an underscore and the action name. The action definition is then given by a terminal that is the name of the action followed by a scope operator (::). This is followed by the argument for the action. The set action argument must be defined in terms of the property itself and the get action argument is simply the property name. The argument definition for other actions is unrestricted. Each feature property ($\hat{p}'$) is identified by its name followed by the “_property” extension and defined as its name, the scope operator, and the property values. An example of this naming convention is given by

$$\begin{align*}
\text{interfaceName}_\text{interface} &= \text{featureName}_\text{feature}; \\
\text{featureName}_\text{feature} &= \text{propertyName}_\text{actions}; \\
\text{propertyName}_\text{actions} &= [\text{propertyName}_\text{set}, \text{propertyName}_\text{get}]; \\
\text{propertyName}_\text{set} &= "set::", \text{propertyName}_\text{property}; \\
\text{propertyName}_\text{get} &= "get::propertyName"; \\
\text{propertyName}_\text{property} &= "propertyValue1"|"propertyValue2"; \\
\end{align*}$$

(3.11)

Notice that the property actions are each placed in square brackets at this step. This represents the assumption that all actions are assumed independent. This assumption can then be modified via semantic definition as discussed in Section 3.2.

3.1.3. Properties

Properties are the most basic building blocks of an interface. The assumption has been made that all outbound information from a system is the value of a property. Based on this assumption, the first step to building an interface is to identify all of the properties for a given domain. Further definition of a property will help in this
process; a property consists of a single value at any given time that is known to the system and can be monitored or modified externally.

Using this definition of a property, the digital alarm clock example introduced in Section 1.3.5 is continued. The properties are relatively easy to identify. They are the current time, the time for which the alarm is set, and the state of the alarm. While in reality, many alarm clocks have more properties than these three; this simple example represents a subset within the larger alarm clock domain. As is shown in the robotic interface domain analysis in Chapter 4, the property identification is typically much more difficult. The scope of this report is not intended to cover domain analysis techniques, so it is assumed that these properties can be determined for well understood domains by a domain expert.

With the properties identified for the domain, the outbound communication is fully detailed. Each of the properties has a set of values that the interface might communicate to the user. The set of values need not be finite, but, if possible, data with infinite values should be constrained. A good example is the current time of the digital alarm clock. While the time values that can be displayed by the digital clock are finite, time itself has infinite values. Time can be constrained by a grammar based on the standard representation of time set forth by the International Organization for Standardization (ISO) in ISO 8601 [24].

Properties with finite property values should have them enumerated. The alarm state for instance can be enumerated as on, off, sounding, or snooze to represent the basic states of an alarm. Providing this well constrained definition of values allows translation into interface representations that can be duplicated consistently. The properties identified for the alarm clock example are given as their EBNF specification and algebraic representation in Table 3.5 with their values explicitly defined. The EBNF specification would be defined by the domain expert and the algebraic specification can be inferred by the optimizer based on the naming conventions to develop the algebraic representation. Notice that the property list is
not defined in EBNF, because it is implied by the various non-terminals with the “_property” extension. Also, as discussed in 3.1.1, the algebraic specification of property values would really only hook into the EBNF; the description of the values in the algebraic representation is only included for clarity.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P = { \text{time}, \text{alarm_time}, \text{alarm_state}} )</td>
<td>( p^1_r = { hh:mm:ss } )</td>
</tr>
<tr>
<td>( \text{time_property} = &quot;time::&quot;, time; )</td>
<td>( p^2_r = { hh:mm:ss } )</td>
</tr>
<tr>
<td>( \text{time} = hh,:'', mm,:'', ss; )</td>
<td>( p^3_r = { \text{on}, \text{off}, \text{sounding}, \text{snooze} } )</td>
</tr>
<tr>
<td>( hh = [ '0'</td>
<td>'1' ], \text{digit},'2','([ '0'</td>
</tr>
<tr>
<td>( mm = [ '0'</td>
<td>'1'</td>
</tr>
<tr>
<td>( ss = [ '0'</td>
<td>'1'</td>
</tr>
<tr>
<td>( \text{alarm_time_property} = &quot;alarm_time::&quot;, time; )</td>
<td>( p^7_r = { \text{on}, \text{off}, \text{sounding}, \text{snooze} } )</td>
</tr>
<tr>
<td>( \text{alarm_state_property} = &quot;alarm_state::&quot;, )</td>
<td>( p^8_r = { \text{on}, \text{off}, \text{sounding}, \text{snooze} } )</td>
</tr>
<tr>
<td>( \text{on}</td>
<td>&quot;\text{off}</td>
</tr>
</tbody>
</table>

Table 3.5: Property Specification for an Alarm Clock

The properties define the outbound communication from the device. All properties within an interface can be used for outbound communication at anytime. For this reason, properties are defined by the property name followed by “_property” as in \( \text{alarm\_state\_property} \). Also, the properties are given an identifier within their definition as a type of namespace for the property. Various properties can share values (time and alarm time, e.g.), so it is important that the property contain this identifier to clarify which property is being given in the communication. This identifier should always match the name of the property.

With the properties and values defined in this fashion, the outbound communication is fully defined and needs no further restriction. Various properties are independent from the interface perspective, so any value listed for a property is
valid as outbound communication at any time. The inbound communication is much more complicated, however. Actions are defined to provide the syntax. Then, the semantics of the actions must be addressed (see Section 3.2).

3.1.4. Actions

With the properties for the domain defined, actions become the next building block for interface generation. Two actions are defined as the core actions of interface generation. These are *get* and *set*. While actions for a domain are not restricted to these two, in most instances they will cover all of the necessary information exchange. With all outbound information being the value of a property, the necessary action to retrieve the value is *get*. Similarly, actions performed on a property are typically intended to change the property value, thus *set* is appropriate with an argument that is a subset of the total property values defined for the domain.

Actions are defined in the EBNF grammar in much the same way that property values are defined. The standard naming scheme is the property name followed by “_actions” as in *alarm_state_actions* because each set of actions (\(A_i\)) is associated with a particular property (\(p_i\)). By default, actions are assumed to be independent in that any combination of actions is valid. This is represented by square brackets around all of the actions in EBNF. This assumption can be modified in the semantic definition of the interface as described in Section 3.2.

The set of actions is defined by a list of actions where each element is the property name followed by an underscore and the action name. The individual actions are then defined by a terminal that consists of the action name followed by the scope operator (::). This is followed by the arguments of the action. The *get* action only needs to specify the property it refers to so that is given within the terminal. The *set* action arguments are restricted to be a subset of its property values and therefore must be given in terms of the property itself. It is a subset of the property values because it may be useful to allow the user to only command the system into a subset of its
possible states. This type of restriction on the set action is discussed further in the following section. The EBNF and algebraic specification of the actions for the alarm clock example are given in Table 3.6 and include references to items in Table 3.5.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_actions = [time_set], [time_get];</td>
<td>( A^1 = { \text{set, get} } )</td>
</tr>
<tr>
<td>time_set = &quot;set::&quot;, time_property;</td>
<td>( a_{set,v} = p_v )</td>
</tr>
<tr>
<td>time_get = &quot;get::time&quot;;</td>
<td>( a_{set,v} = \emptyset )</td>
</tr>
<tr>
<td>alarm_time_actions = [alarm_time_set], [alarm_time_get];</td>
<td>( A^2 = { \text{set, get} } )</td>
</tr>
<tr>
<td>alarm_time_set = &quot;set::&quot;, alarm_time_property;</td>
<td>( a_{set,v} = p_v^2 )</td>
</tr>
<tr>
<td>alarm_time_get = &quot;get::alarm_time&quot;;</td>
<td>( a_{set,v} = \emptyset )</td>
</tr>
<tr>
<td>alarm_state_actions = [alarm_state_set], [alarm_state_get];</td>
<td>( A^3 = { \text{set, get} } )</td>
</tr>
<tr>
<td>alarm_state_set = &quot;set::&quot;, alarm_state_property;</td>
<td>( a_{set,v} = p_v^3 )</td>
</tr>
<tr>
<td>alarm_state_get = &quot;get::alarm_state&quot;;</td>
<td>( a_{set,v} = \emptyset )</td>
</tr>
</tbody>
</table>

Table 3.6: Action Specification for an Alarm Clock

The items in this table specify the actions that can be performed on the alarm clock. This includes the ability to monitor and modify the time, alarm time, and alarm state. The algebraic specifications identify the set and get actions and define the set action arguments as the entire set of property values. The get action in each case has no arguments and is, therefore, presented as the null set (\( \emptyset \)).

3.1.5. Features

Features are the basic building blocks for a product line. Each feature can have multiple representations including the code that defines feature functionality, test programs, documentation, makefiles, etc. A particular product instance is created through a logical combination of features belonging to the product line where the
interface to each feature in a product line \((f_k \subseteq F)^3\) is defined as a set of property-action tuples.

In applications like internet-based computer sales, constructs that parallel feature analysis are often used. A customer can choose to add a monitor, for instance, to the product being ordered. With the monitor feature added, the customer is often free to choose among representations of the feature (CRT, LCD, etc.). This is similar to the application of features to interface generation. Once the features are laid out for the product family, they are simply selected and combined for a particular product. For the alarm clock example, three features are identified. They are time, alarm, and snooze. The time feature simply includes the time property and its actions, but the other two features have more complicated definitions.

Obviously, in order for a clock to have an alarm, it requires both the ability to set the time for the alarm and the ability to turn the alarm on and off, so it requires alarm time actions and a subset of alarm state actions. The alarm state set action arguments are restricted using \(\text{alarm\_state\_set}\). As a subset of the property values, the action arguments are defined as the property values with exclusions. Exclusions in EBNF (the ‘-’ sign) are defined as anything that satisfies the terms before the ‘-’ but does not satisfy the terms after the ‘-’. For this fictional product-line, the alarm feature will not include sounding because the sounding state is set internally. The snooze feature obviously requires the entire alarm feature along with the snooze alarm state. The EBNF and algebraic specifications for the alarm clock features are given in Table 3.7.

---

3 As a reminder, \(F\) represents the total list of features in a product line, and \(f_k\) is a particular feature.
Table 3.7: Features for Alarm Clock Product Line

To abbreviate Table 3.7, items that have been previously defined (e.g. `time_actions`) are left out. This makes it appear that the feature has no reference to the properties it is associated with. In fact, however, the actions are defined in terms of the properties in such a way that they implicitly include their associated property and its values into the feature. In the case where no `set` action is defined for a property that is part of a feature, the EBNF definition of the property still needs to be included. An example is an alarm clock that sets itself based on radio signals. In such a system, the time property need not have a `set` action defined for it. The time feature to this system would be defined as in Table 3.8. Notice that the time property is simply included in the EBNF definition so that it is defined even though it is never referenced.
3.1.6. **Syntactic Interface Composition**

Section 5.1 details the process of interface generation which takes the syntactic interface as defined above and the semantic definition in the following section to create and optimize a complete interface. This section will briefly explain the process for the syntactic part of the interface. Ultimately, the overall interface to a product instance is defined as an addition of individual interfaces of the features supported by the product instance. It can include any legal combination of the defined features. The EBNF interface specification for an alarm clock that supports all of the defined alarm clock features is given by

$$\text{alarm\_clock\_interface} = \text{time\_feature, alarm\_feature, snooze\_feature};$$  \hspace{1cm} (3.12)

Simply composing the interface definition of each of these features would result in a large interface description including redundancies. For instance, all of the features include the time actions. While, EBNF does not forbid these redundancies, they may cause ambiguity in instances of the grammar because there is more than one path to some instances. Again, this is not forbidden, but it is desirable to resolve these
redundancies as addressed in Section 5.1. This is one of the instances where the algebraic specification becomes useful. An EBNF combination would appear like a concatenation of components whereas the algebraic specification allows combination via union.

3.2. Interface Semantics

Defining the syntax of an interface does not completely describe the interface. This is often true for programming languages defined with EBNF as well. Common programming languages, such as C, require variables to be declared before they are used. The syntax of the language, defined in EBNF, is unable to guarantee this requirement is fulfilled. Typically, a less formal declaration of these restrictions accompanies the EBNF grammar.

This section addresses context sensitive components of the interface definition and methods of formalizing their descriptions. A formal definition of the semantic aspects of the interface allows the interface to be optimized more effectively and allows better runtime operation of the system. The semantic constructs addressed are: physical relationships among actions, time-based relationships among actions, and conditional semantics. These semantic definitions provide information that may be enforced at runtime, and the optimization scheme may use the information to modify the syntax. The semantics is defined algebraically using the specification given in the next section. The three semantic constructs are then thoroughly explained in the following sections.

The semantic definition applies to lower level components of the interface and therefore it is important to clarify the terms used in this section. The term “action” is used to represent an action as before (e.g. $a\_set^1$ and $\text{set::alarm\_state}$) or it may be used to suggest the use of an action with a particular argument value (e.g. $a\_set_{v2}^1$ and $\text{set::alarm\_state::on}$). While there is an important distinction between these two cases, they can be used interchangeably in the semantic definition syntax. The other
term used for semantic definition is “state;” it is used to reference a particular value of a property (e.g. $v^1_2$ and alarm_state::on). As such, the current state is the value that would be returned from the interface if a get action was performed. With these terms defined, the semantic specification can be clearly given.

### 3.2.1. Semantic Definition Syntax

In order to have a well defined semantic definition for the interface, the syntax of the definition must be defined. While the specification of this syntax is somewhat arbitrary, common algebraic symbols are used for familiarity. These symbols are thoroughly explained in the following sections but are briefly introduced in this section so they are all defined in one place. Three sets of symbols are important. The first is used in all three of the semantic definitions and contains symbols for grouping actions or states and negating items. Table 3.9 gives these symbols in precedence order from highest to lowest.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>group symbols: used to group items into a list and interrupt the precedence order of the other symbols</td>
</tr>
<tr>
<td>¬</td>
<td>not symbol: negates the item following the symbol (e.g. $\neg v^1_2$)</td>
</tr>
<tr>
<td>&amp;</td>
<td>and symbol: separates items in an action or state list (e.g. $v^1_2$ &amp; $v^3_1$)</td>
</tr>
<tr>
<td></td>
<td>or symbol: separates options in an action or state list (e.g. $v^1_2$</td>
</tr>
</tbody>
</table>

*Table 3.9: Semantic Grouping Symbols*

The second set of symbols (Table 3.10) includes symbols used for defining the physical and time-based relationships of actions. These two symbols provide the ability to specify whether seemingly unrelated actions have restrictions on their concurrent use. The specification of these restrictions is independent of the cause for the restriction because, from the interface perspective, only the existence of and adherence to the restrictions are important.
Symbol | Definition
---|---
+ | requires symbol: if the action(s) on the left side is part of a communication, the action(s) on the right side must also be included (e.g. \(a\_set^1 + a\_set^3\))
− | excludes symbol: if the action(s) on the left side is part of a communication, the action(s) on the right side must not be included (e.g. \(a\_set^1 - a\_set^3\))

Table 3.10: Physical/Time-Based Relationship Symbols

The last set of symbols (Table 3.11) defines conditional relationships among actions. Conditional relationships are implemented primarily at runtime and have a unique set of symbols to define the validity of commands based on the current state of the system. The only time the optimizer might use conditional relationships to change the interface syntax is if there is no state in which an action is valid. In this case, the action could be removed from the syntax.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
</table>
\(\times\) | validity symbol: the action(s) on the left side is valid if the system is in the state(s) on the right side (e.g. \(a\_set^1 \times v^3\)) |
\(\rightarrow\) | state transition symbol: given the action(s)/state(s) combination on the left side, the system will transition into the state(s) on the right side (e.g. \(a\_set^1 \times v^3 \rightarrow v^4\)) |

Table 3.11: Conditional Relationship Symbols

3.2.2. Physical Relationships

Systems may have physical links between properties identified separately. A simple example within the robotics domain is the physical link between the velocity of a serial robot’s end-effector \(v_{ee}\) and the position \(\theta\) and velocity \(\dot{\theta}\) of its joints. While these may be modeled as distinct properties with distinct values, they are linked by the Jacobian matrix \(J(\theta)\) which defines the geometric representation of the robot as a function of the joint position. This relationship is given by

\[
v_{ee} = J(\theta)\dot{\theta}
\]  

(3.13)
While, end-effector velocity and joint velocity might both be controllable as properties of the domain, this physical relationship between the two shows that they cannot be independently set. It is assumed that the domain expert developing the semantics can extract more concrete definitions of the validity of concurrent commands from these relationships. A likely restriction developed from the relationship between joint velocity and end-effector velocity is defined by

\[ \text{set::eef\_velocity} - \text{set::joint\_velocity}; \]  

(3.14)

This statement is both human and machine readable and clearly states that if end-effector velocity is set in a communication then the joint velocity cannot also be set in the same communication. Clearly, the inverse is automatically true which defines the two actions as mutually exclusive. Similarly, if an action(s) requires another action(s), the ‘+’ operator can be used to define that the action(s) on the left side requires the action(s) on the right side. In this case, the inverse is not necessarily true and must be explicitly defined if desired. To accommodate the special case where an action(s) should always appear in a communication or should only appear alone, a ‘+’ or ‘−’ is placed in front of the action(s), respectively as in

\[ +\text{required\_action}; \]  

(3.15)

Also, an action(s) can require or exclude all but a few other actions by using the not symbol (¬) as in

\[ \text{first\_action} - \neg (\text{second\_action} | \text{third\_action}); \]  

(3.16)

It is important to realize that the notation defined in this section is not part of the EBNF definition, but is part of a completely separate semantic definition to accompany the syntactic definition. This information can be used by the interface optimization scheme discussed in Chapter 5 and can also be used at runtime by the system. For instance, the syntax may be modified to remove the possibility of action conflicts or the system can just throw out conflicting commands at runtime. For completeness, Equation (3.17) gives an informal EBNF definition of this syntax.
3.2.3. Time-Based Relationships

Time-based relationships are similar to physical relationships with the connection being made by a time component. For instance, the joint level velocity \( \dot{\theta} \) is clearly the first-derivative of the joint level position \( \theta \) with respect to time \( t \) as defined by

\[
\dot{\theta} = \frac{d\theta}{dt}
\]  

(3.18)

Time-based relationships are a special case in that they might restrict the interface if it contains a time component such as the ability to specify a length of time for the position to move to the commanded value; however, if the interface does not contain a time component, this relationship is likely to have no effect. It is left to the domain expert to define time-based restrictions in the same way they are defined for physical relationships. A restriction for the relationship between joint velocity and joint position when time is involved is given by

\[
(\text{set::joint\_position \& set::duration}) - \text{set::joint\_velocity};
\]

\[
(\text{set::joint\_velocity \& set::duration}) - \text{set::joint\_position};
\]  

(3.19)

This would allow control of joint velocity, joint position, and duration in any combination as long as they are not all included in a single communication. The use of these definitions will be addressed further in Section 5.1. Future work should address other temporal information in interfaces. For instance, certain commands may have an execution time related to them that can be defined a priori. Similarly, the user may be able to specify a range of times along with the fixed time discussed above. These issues are discussed in Chapter 6.
3.2.4. Conditional Relationships

While the previous two semantic restrictions relate elements of the interface syntax to define the validity of actions, conditional relationships relate actions to the current state of the machine to define validity. This type of condition gives meaning to the actions defined for the interface. Again, the logic used to combine individual feature interfaces may use this information to modify the syntax; however, the only time this might happen is if there is no state of the machine that allows an action to be valid. In this instance, the action could be removed from the interface.

A system is often required to be in a specific state for a command to be valid. While the command does not need to be eliminated from all interface representations when it is invalid, certain interface representations may use the information to ignore invalid communication. In the alarm clock example, the clock is likely to be defined so that a snooze action is only valid when in the sounding state. The physical interface does not change in that the snooze button is always available; the software simply ignores the input unless the system is in the sounding state. This type of relationship is easily represented in a state diagram (see Figure 3.1).

![Figure 3.1: State Diagram for Alarm State Feature](image)

In this diagram, solid arrows represent transitions triggered by external events, and dashed arrows represent transitions triggered internally. In this example, the clock is internally responsible for transitioning into the sounding state when required. All other transitions are obviously triggered by the appropriate set action. Since this
diagram defines that there is no state in which set::alarm_state::sounding is a valid action, the optimizer could automatically produce this restriction in the syntax to duplicate the manual restriction given in Table 3.7.

Most of the restrictions defined by the diagram, however, provide additional information about runtime functionality and cannot be implemented into the syntax. It shows that while a set::alarm_state::off action is valid from any other state, a set::alarm_state::on action is only valid if the alarm is off. If an action is received when in a state without a valid transition defined for that action, it is assumed that the system ignores the action and remains in the current state. To change this default behavior, a transition that sends the system into an error state could be defined for invalid actions. A complete state diagram of the digital alarm clock is given in Figure 3.2 that includes all of the state machines required to maintain the alarm clock.

Figure 3.2: State Diagram for Alarm Clock Interface

This diagram includes the entire alarm clock system as the three concurrent state machines required to maintain the three properties of time, alarm time, and
alarm state. The transitions have been labeled with the event that causes them with
the exception of the transitions that occur when the originating state’s process is
completed. The names of the states represent the value of the property when queried
in that state unless an alternative value is indicated under the state name. To simplify
the diagram, the various states the alarm state can achieve once the alarm is turned on
are grouped into a superstate that can be exited with a set::alarm_state::off action.
Lastly, the black dots represent the unpowered state of the machine. When the clock
is given power, it immediately transitions from the black dots to the next states. While
this diagram now fully describes the system, a software system cannot read the
diagram to determine how to handle events. It is important, therefore, to define an
algebraic representation of this diagram in much the same way that EBNF
algebraically defines a language.

It has long been desired to algebraically represent the information in a state
machine. Particularly, this information is useful when automatically generating
software to implement the state machine. Many representations have been developed
to suit the particular applications being addressed. This section will briefly present
transition tables, transition matrices, and the general format of most application
specific representations. An argument is then presented as to why a different
representation should be used that is specific to interfaces.

3.2.4.1. Transition Tables

Transition tables are relatively simple definitions of a state machine, although
they can grow large very quickly. Basically, two tables are used to define the next
state and output as a function of the current state and input. Therefore the size is
given by

\[ size = 2 \cdot N \cdot M \]  

(3.20)

where \( N \) is the number of states in the system and \( M \) is the number of input events.
As adapted from [20], the format of a transition table for a machine with an input alphabet \{\xi_1, \xi_2, \ldots, \xi_m\}, an output alphabet \{\zeta_1, \zeta_2, \ldots, \zeta_p\}, and a state set \{\sigma_1, \sigma_2, \ldots, \sigma_n\} is shown in Table 3.12.

<table>
<thead>
<tr>
<th></th>
<th>\xi_1</th>
<th>\xi_2</th>
<th>\ldots</th>
<th>\xi_m</th>
<th>\xi_1</th>
<th>\xi_2</th>
<th>\ldots</th>
<th>\xi_m</th>
</tr>
</thead>
<tbody>
<tr>
<td>\sigma_1</td>
<td>\xi_1</td>
<td>\xi_2</td>
<td>\ldots</td>
<td>\xi_m</td>
<td>\xi_1</td>
<td>\xi_2</td>
<td>\ldots</td>
<td>\xi_m</td>
</tr>
<tr>
<td>\sigma_2</td>
<td></td>
<td></td>
<td>Output value selected from {\zeta_1, \zeta_2, \ldots, \zeta_p}</td>
<td></td>
<td></td>
<td></td>
<td>Next state selected from {\sigma_1, \sigma_2, \ldots, \sigma_n}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\sigma_n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.12: Format of State Machine Transition Tables

3.2.4.2. Transition Matrices

Transition matrices are very similar to transition tables with the data simply formatted differently. Each element \((M_{ij})\) in the transition matrix represents all of the transitions from state \(\sigma_i\) to state \(\sigma_j\). A single transition is represented by an input/output pair \((\xi, \zeta)\) and multiple transitions are separated by an option symbol \((\vee)\). In this way, a state machine can be fully defined mathematically. [20] provides a much more detailed description of transition matrices, including all of their uses beyond a simple representation of the state diagram.

3.2.4.3. Application Specific Representations

Many application specific representations of state machines have been developed for various purposes. Typically, these representations are used by some type of state machine generator to implement a state machine in software automatically. The structure of the representations is usually very similar, but the syntax may be different. One such representation is briefly presented in [50]. The basic syntax of this representation could be modeled in EBNF (see Section 3.1.1) as a set of state productions defined by
3.2.4.4. Conditional Relationship Formalization

While each of the previously discussed state machine representations is good for particular applications, there are several reasons why none of them is ideal for interface application. First, and most importantly, these methods are not well suited for application to concurrent state machines like Figure 3.2. Not only can a single action affect the state of multiple machines, but also the validity of an action within one machine may depend on the state of another machine. This would be difficult to model in the representations discussed.

Similarly, the transition matrix and application specific representations require information to be addressed in a particular order. The current state is found first, and then the event is processed to get the results. While it may be possible, it would be inefficient to reverse this order. For interface communication processing, it is more convenient to lookup the action first and then to determine if that action applies to the current state. While this is a subtle difference, it can be very important for efficiency of the communication validation.

Another difficulty with the transition table and transition matrix representations is in human readability. While, the representation should be machine readable, it is also desirable that a human reader can easily create, modify, and understand the representation. Since the state machine representation for interfacing is not intended for use in implementing the state machine, but only to define the validity of actions, it need not contain a complete definition of the state machine. Conditional restrictions are defined in an informal EBNF grammar as

\[
\begin{align*}
state \_ production & = state \_ name, ', event \_ result, \{', event \_ result\}; \\
event \_ result & = event \_ name, ' \rightarrow (', state \_ name, ', output \_ name, ')';
\end{align*}
\]
The basic definition contains an action(s) followed by a validity symbol (\(\times\)). If nothing is included on the right side of the validity symbol or it is immediately followed by the state transition symbol (\(\rightarrow\)), the action(s) is always valid no matter what the current state. If a state(s) appears, the action(s) is only valid when the current state(s) of the machine matches the state(s) given. If desired, a state transition symbol (\(\rightarrow\)) can then be used to define the new state(s) of the machine as additional information that may be useful to the human reader of the definition. Equation (3.23) gives the conditional relationship definition for the digital alarm clock example.

\[
set :: \text{alarm} \_\text{state} :: \text{on} \times \text{alarm} \_\text{state} :: \text{off} \rightarrow \text{alarm} \_\text{state} :: \text{on};
\]

\[
set :: \text{alarm} \_\text{state} :: \text{off} \times \neg \text{alarm} \_\text{state} :: \text{off} \rightarrow \text{alarm} \_\text{state} :: \text{off};
\]

\[
set :: \text{alarm} \_\text{state} :: \text{snooze} \times \text{alarm} \_\text{state} :: \text{sounding} \rightarrow \text{alarm} \_\text{state} :: \text{snooze}; \quad (3.23)
\]

\[
set :: \text{alarm} \_\text{time} \times;
\]

\[
set :: \text{time} \times;
\]

Clearly this does not fully define the state machine of Figure 3.2 nor is it intended to. It does, however, define the validity of the actions based on the current state. An interface representation could use this information to modify its behavior based on the current state of the machine, but cannot know the current state without querying the machine. A simple example would be a Graphical User Interface (GUI) for the alarm clock like the one shown in Figure 3.3. This representation implements the semantic restrictions specified in Equation (3.23) by making the snooze button unavailable when the alarm is not sounding and using an on/off switch.
3.3. Summary

The interface generation approach presented in this report requires a well-defined specification of the interface to product-line features. As such, this chapter has presented properties, property values, and actions as the core components for specifying interfaces to features. Properties are defined as parts of the system that have a known value at any given time that can be monitored or modified by a user, property values are the enumeration of the possible values for the property, and actions define the interface for monitoring or modifying the property.

These components are specified in two forms defined in this chapter. The algebraic specification is designed as a concise specification of the components that is well-suited for algebraic addition, reduction, and optimization of the combined features. The Extended Backus-Naur Form (EBNF) specification provides an easily understandable, human-readable specification. The domain expert can specify the interface components in the EBNF specification and the algebraic representation can be inferred from the result.

With the interface components specified, the relationships among components are addressed. This semantic specification provides information to the interface reducer and optimizer used in the compilation of product interfaces. As such, the semantic specification contains context specific information that could not be defined in the syntax of the interface.
This chapter provides the backbone of the interface generation approach presented in this report. Chapter 4 demonstrates the concepts of this chapter on the robotics domain and Chapter 5 discusses feature combination, interface reduction, and interface optimization. The material in this chapter is crucial for the development of the rest of this approach.
Chapter Four

4. Robotics Product-Line Interface Specification

Interface generation for a robotics product-line demonstrates use of the specification syntax defined in the previous chapter. Also, the robotics interface specification provides a robust background for discussing interface reduction, optimization, and representation in the next chapter. The following sections discuss the properties, actions, and features defined for the robotics domain and present the syntax and semantics of the product-line. This work is not intended to present domain analysis techniques in detail, so the analysis in this chapter is given informally with the assumption that a domain expert can identify the necessary components for well understood domains. Desired features are described first without a formal definition. From this, properties and actions are defined and then the features can be formally defined. Lastly, semantic restrictions on the interface are given.

While ideally the properties and actions would be identified independently of features, in reality it is likely that the product-line features desired are more easily identifiable. This may be because the software already exists and its features are known or because Feature Oriented Programming (FOP) is being applied to the software as well, and it is desired to support the same features identified for the software. In the case of this robotics analysis, the features addressed have been identified for control software generation via FOP [26][27]. The following section identifies the desired features informally.

69
4.1. Scope

Application of FOP principles to interface generation parallels a concurrent research thread at the Robotics Research Group (RRG) applying the same principles to Robot Control Software (RCS) generation [26]. While the interface generation techniques presented in this report are applicable to manually programmed software systems, it is logical to address FOP based generation of the software and the software interface simultaneously. Therefore the product-line addressed will encompass many of the features identified for RCS generation.

Initially, only the Device Interface (DI) portion of the RCS (see Figure 1.1) is being addressed with FOP. The initial features identified for DI generation are given in Table 4.1. Properties and actions have been defined for application of FOP to interfaces and are not traditionally part of feature identification. Thus, the features defined for DI generation must be analyzed to extract the properties and actions for interface specification.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Position Command</td>
<td>allows control and monitoring of robotic joint positions</td>
</tr>
<tr>
<td>Joint Velocity Command</td>
<td>allows control and monitoring of robotic joint velocities</td>
</tr>
<tr>
<td>Joint Torque Command</td>
<td>allows control and monitoring of robotic joint torques</td>
</tr>
<tr>
<td>Joint Current Command</td>
<td>allows control and monitoring of robotic joint currents</td>
</tr>
<tr>
<td>Limits Checking</td>
<td>provides filtering of joint commands that are beyond the hardware supported range</td>
</tr>
<tr>
<td>Excess Error Check</td>
<td>provides filtering of joint commands that require a change in value beyond that supported by hardware</td>
</tr>
</tbody>
</table>

Table 4.1: Robot Control Software Features

While this table appears to present only six features, in actuality, RCS application will require a minimum of eight because excess error checking and limits checking require different implementation for each of the four command features. Similarly, depending on the granularity of the features, many more could be defined that are encompassed by the set given. For instance, a feature might implement only monitoring or control of various commands instead of both. Also, joint positions, for
instance, can be given as radians or degrees. It becomes clear that the number of features can quickly explode. For this reason, several assumptions are made in this section to restrict the product-line addressed.

This analysis focuses on the interface requirements of typical six Degrees-Of-Freedom (DOF) industrial robots with rotary joints. While these are not simple systems, this assumption removes the complexity of redundant robots and redundancy resolution. It should be possible to model these complexities as features, yet the number of features that might be required could easily skyrocket. Similarly, restricting the joints to be rotary restricts the possible joint configurations to one. Without this restriction, each joint could be rotary or prismatic, resulting in sixty-four possible configurations for a six DOF robot. Lastly, it is assumed that joint and end-effector commands cover the entire six DOF output space. While it is possible to command a robot to an end-effector position in less than six DOF and to simultaneously control a subset of the joints, this is a special case that does not merit consideration at this time. With these assumptions in mind, several other features are given in Table 4.2 to demonstrate the flexibility of the approach given in this report.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-Effector Commands</td>
<td>all of the features in Table 4.1 with the exception of Joint Current Command have parallels at the end-effector level</td>
</tr>
<tr>
<td>Multiple End-Effector Position</td>
<td>end-effector position commands can include orientation specification via fixed frame rotations, Euler rotations, or rotation matrix</td>
</tr>
<tr>
<td>Multiple End-Effector Position</td>
<td>representation via fixed frame rotations, Euler rotations, or rotation matrix</td>
</tr>
<tr>
<td>Tool State Control</td>
<td>allows control and monitoring of various types of tools</td>
</tr>
<tr>
<td>Tool Change</td>
<td>allows changing the current tool</td>
</tr>
<tr>
<td>Force/Torque Sensor</td>
<td>allows monitoring the force and torque at the end-effector through a hardware sensor</td>
</tr>
<tr>
<td>Collision Detection</td>
<td>monitors distance from potential collisions of any part of the robot and provides artificial force values at the end-effector signaling safeness of movements</td>
</tr>
</tbody>
</table>

Table 4.2: Additional Features Identified for Interface Generation
The features of Table 4.2 are identified in [27] as features of the Computational Components (CC) layer of the RCS (see Figure 1.1) that will be addressed in the next iteration of RCS generation research. They encompass several different aspects of robot control and represent the complexity that FOP can support. The following section presents the properties and actions required to define the features identified in this section.

### 4.2. Properties and Actions Specification

From the informal features identified in the previous section, the following properties were identified.

- A. System State
- B. Joint Position
- C. Joint Velocity
- D. Joint Torque
- E. Joint Current
- F. End-Effector Position
- G. End-Effector Velocity
- H. End-Effector Force/Torque
- I. Tool State
- J. Tool Change
- K. Force/Torque Sensor
- L. Collision Distance
- M. Obstacle Artificial Force/Torque

Algebraically, the properties are listed as

\[
P = \{\text{sys}, \text{j}_\text{pos}, \text{j}_\text{vel}, \text{j}_\text{torq}, \text{j}_\text{curr}, \text{ee}_\text{pos}, \text{ee}_\text{vel}, \\
\text{ee}_\text{ft}, \text{tool}_\text{st}, \text{tool}, \text{ft}_\text{sens}, \text{coll}_\text{dist}, \text{ob}_\text{ft}\} \tag{4.1}\]

Most of these properties have a relatively straightforward correspondence to the informal features. Some of the features, however, do not have a dedicated property. Also, the system state property is affected by nearly all of the. This section presents each of these properties in detail. The correlation to the informal features is discussed and the properties and their associated actions are formally defined. Many of the properties presented in this section may be represented in various numerical forms and may have varying units. This work does not fully address representation of numerical values including units. At this stage, the domain expert simply specifies the
appearance of values and the valid units; however, future work should address additional methods of representing these components. The entire EBNF and algebraic domain model specifications are given in Appendix A. Because state diagrams play a key role in the specification of properties, actions, and semantic relationships, the state diagram format used in this report is summarized in the following subsection.

### 4.2.1. State Diagrams

Table 4.3 summarizes the format used for state diagrams in this report. The format is loosely based on the Unified Modeling Language state diagram specification [52].

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="initial-state.png" alt="initial state" /></td>
<td>initial state: defines the startup state of the machine, system immediately transitions out of this state upon startup</td>
</tr>
<tr>
<td><img src="final-state.png" alt="final state" /></td>
<td>final state: when this state is reached the machines life cycle is completed, no transitions can leave this state</td>
</tr>
<tr>
<td><img src="state-transition.png" alt="state transition" /></td>
<td>state transition: this symbol is used to represent state transitions caused by user interaction, typically it is labeled with the action that causes the transition</td>
</tr>
<tr>
<td><img src="auto-state-transition.png" alt="automatic state transition" /></td>
<td>automatic state transition: this symbol is used to represent state transitions caused by the machine in response to completion of a state or internal stimulus</td>
</tr>
<tr>
<td><img src="state.png" alt="state" /></td>
<td>state: this symbol represents a state of the machine with the state name included inside</td>
</tr>
<tr>
<td><img src="superstate.png" alt="superstate" /></td>
<td>superstate: a large state symbol can be placed around several states to designate a superstate which can defined entry and exit behavior for the included group</td>
</tr>
<tr>
<td><img src="title.png" alt="title" /></td>
<td>title box: occasionally used to label state diagrams and superstates</td>
</tr>
<tr>
<td><img src="shallow-history.png" alt="shallow history" /></td>
<td>shallow history: pseudostate that represents the previous state of the machine was in on the same level as the symbol, the previous condition of substates is not retained</td>
</tr>
<tr>
<td><img src="deep-history.png" alt="deep history" /></td>
<td>deep history: pseudostate that represents the previous state of the machine at all levels, all states and substates will be returned to their previous states</td>
</tr>
</tbody>
</table>

Table 4.3: State Diagram Symbols
4.2.2. System State

The system state property is an important property in the robotics domain. It provides the ability to initialize and shutdown the robot as well as providing access to the current state of the overall system. A subset of the overall property values is required by most features for typical robot functionality. The state machine in Figure 4.1 includes only the basic states required for robot initialization and shutdown (see Section 4.2.1 for explanation of state diagram symbols). Some organization of the diagram is given so that adding elements is more straightforward.

Figure 4.1: State Diagram for Basic System State Property

Upon being powered on, the system enters an inactive state where it remains until enabled. Once enabled, the system requires homing before moving into an idle state. The robot is intended to be powered off from the inactive, shutdown, or estop states only. The shutdown command can be given from either the enabled or idle states while the estop state can be activated from any other state. The error state can
be entered from all states except the estop state and the error is given a letter and number to clarify the type of error. In this case, the ‘S’ specifies a system error.

The history (H) and deep-history (H*) pseudo-states instruct the system to move into the state that the system was in most recently. The history (H) pseudo-state puts the system in the previous state at the same level while the deep-history (H*) pseudo-state places the system in the previous substate at any level. Because the results of the history pseudo-states are unknown, special actions are used that do not correspond to actual states of the system. At this stage in this research, additional property values are added to the property to support these actions; however, this special case should be more thoroughly investigated in the future. The formal definition of the basic system state property is given in Table 4.4.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_property = &quot;sys::&quot;, sys_t;</td>
<td>( p_v^1 = { \text{inactive, enabled, idle, shutdown, estop, homing, hold, cancel, resume, error_S#, error_clear} } )</td>
</tr>
<tr>
<td>sys_t = init_t</td>
<td>end_t</td>
</tr>
<tr>
<td>end_t = &quot;shutdown&quot;</td>
<td>&quot;estop&quot;;</td>
</tr>
<tr>
<td>move_t = &quot;homing&quot;;</td>
<td>hold_t = &quot;hold&quot;</td>
</tr>
<tr>
<td>err_t = &quot;error_&quot;, ('S', digit</td>
<td>&quot;clear&quot;);</td>
</tr>
</tbody>
</table>

Table 4.4: Basic System State Property Values

This part of the system state is vital to any robotic system within the domain being addressed. It does not take into account any of the informal features identified in Section 4.1, however. When these features are taken into account, changes must be made to the state diagram to reflect the effects that the various informal features have on the system as a whole. The complete system state diagram is shown in Figure 4.2.
Only a few changes have been made to the system to support the informal features that were identified. The `tool` state implies that the system is currently performing a tool change and the `val` state indicates a value has been given to the system toward which it is moving. Neither of these states can be entered directly; they are entered by setting a value to an associated property. For instance, the tool state would be entered by the command `set::tool::drill` which is discussed in Section 4.2.6.

Also, the error state has been modified to allow other error types beyond system errors. For this example, the additional error types are: limits errors (L), excess errors (E), tool errors (T), and collision errors (C) with the additional digit further identifying the error. Adding limits errors and excess errors covers the interface requirements of limits checking and excess error checking. While these are not simple features to implement in the control software, they require only this small
amount of user interaction. The reason that informal features requiring a significant amount of user interaction only require minor changes to the system state property is that they have distinct properties for the majority of their interaction. These additional properties of the domain are discussed in the following sections. The formal specification of the complete system state property is given in Table 4.5.

Table 4.5: Complete System State Property Values

The actions identified for the system state property are simply get and set. They are formally given in Table 4.6.

Table 4.6: System State Actions

Based on the state diagram of Figure 4.2, some of the system state property values are not valid arguments of the set action. These restrictions could be specified here; however, they can also be applied at the semantic level as in this example (see Section 4.4).
4.2.3. Joint Position, Velocity, Torque, and Current

The various joint commands given in Table 4.1 have very similar specifications and their properties are thus all presented in this section. Each command is represented by a single property and its get and set actions. Defining the property values requires valid numerical representations and units. For this example, the RCS can accept joint positions in degrees (deg) or radians (rad), joint velocities in degrees per second (deg/s) or radians per second (rad/s), joint torques in Newton-meters (N-m), and joint current in amperes (amp). The numerical values are defined as the set of real numbers and the six values are listed in parentheses. Clearly, in an actual application, it is likely that the values would be restricted to a particular subset of real numbers. More formal definition of variables and units is discussed as an issue for future work. Table 4.7 defines the property values for each of the joint command properties.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
</table>
| \_j\_pos\_property = \"j\_pos\::(\"real\_val,\ 5*\'(\',real\_val)\'),("deg"|"rad")\); \  
  \real\_val = [\'+\',\'-\'].\{\digit\}.\{\digit\}\_; \  
  \digit = \"0\"|\"1\"|\"2\"|\"3\"|\"4\"|\"5\"|\"6\"|\"7\"|\"8\"|\"9\"_; | \(p_j^5 = (R,R,R,R,R,R)\) |
| \_j\_vel\_property = \"j\_vel\::(\"real\_val,\ 5*\'(\',real\_val)\'),("deg/s"|"rad/s")\); | \(p_j^3 = (R,R,R,R,R,R)\) |
| \_j\_torq\_property = \"j\_torq\::(\"real\_val,\ 5*\'(\',real\_val)\'),"N-m"; | \(p_j^4 = (R,R,R,R,R,R)\) |
| \_j\_curr\_property = \"j\_curr\::(\"real\_val,\ 5*\'(\',real\_val)\'),"amp"; | \(p_j^6 = (R,R,R,R,R,R)\) |

Table 4.7: Joint Command Properties’ Values

Next, the actions are identified for these properties. The only actions required for these properties are the get and set actions. Table 4.8 specifies the actions associated with these properties.
### Table 4.8: Joint Command Properties’ Actions

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j_{pos_actions} = [j_{pos_set}, j_{pos_get}] )</td>
<td>( A^2 = {\text{set, get}} )</td>
</tr>
<tr>
<td>( j_{pos_set} = &quot;\text{set::&quot;, } j_{pos_property}; )</td>
<td>( a_{set_y} = p^2_y )</td>
</tr>
<tr>
<td>( j_{pos_get} = &quot;\text{get::j_pos;})</td>
<td>( a_{get_y} = \emptyset )</td>
</tr>
<tr>
<td>( j_{vel_actions} = [j_{vel_set}, j_{vel_get}] )</td>
<td>( A^3 = {\text{set, get}} )</td>
</tr>
<tr>
<td>( j_{vel_set} = &quot;\text{set::&quot;, } j_{vel_property}; )</td>
<td>( a_{set_z} = p^3_z )</td>
</tr>
<tr>
<td>( j_{vel_get} = &quot;\text{get::j_vel;})</td>
<td>( a_{get_z} = \emptyset )</td>
</tr>
<tr>
<td>( j_{torq_actions} = [j_{torq_set}, j_{torq_get}] )</td>
<td>( A^4 = {\text{set, get}} )</td>
</tr>
<tr>
<td>( j_{torq_set} = &quot;\text{set::&quot;, } j_{torq_property}; )</td>
<td>( a_{set_z} = p^4_z )</td>
</tr>
<tr>
<td>( j_{torq_get} = &quot;\text{get::j_torq;})</td>
<td>( a_{get_z} = \emptyset )</td>
</tr>
<tr>
<td>( j_{curr_actions} = [j_{curr_set}, j_{curr_get}] )</td>
<td>( A^5 = {\text{set, get}} )</td>
</tr>
<tr>
<td>( j_{curr_set} = &quot;\text{set::&quot;, } j_{curr_property}; )</td>
<td>( a_{set_z} = p^5_z )</td>
</tr>
<tr>
<td>( j_{curr_get} = &quot;\text{get::j_curr;})</td>
<td>( a_{get_z} = \emptyset )</td>
</tr>
</tbody>
</table>

#### 4.2.4. End-Effector Position, Velocity, and Force/Torque

The end-effector commands are closely related to the joint-level commands. In fact, the most common way of implementing end-effector control of position and velocity is by calculating the joint values that produce the desired end-effector values. The likely method of providing end-effector control of force/torque, however, is by using values collected from a sensor at the end-effector. At the interface level, the implementation of features is not a concern, only the incoming and outgoing data is relevant.

Again, a representation of the values needs to be established for each of the end-effector properties. In Table 4.2, one of the informal features specifies that the interface should support multiple end-effector orientation representations. This example supports specification via FixedXYZ rotations, EulerZYZ rotations, and rotation matrix. These representations and many others are thoroughly detailed by
Craig in [14]. The displacement \( \mathbf{d} \) of the end-effector is specified in terms of the fixed frame axes in this example and is given in inches (in) or centimeters (cm).

FixedXYZ and EulerZYZ are representations of the orientation of a point via three rotation values. FixedXYZ is given by

\[
R_f = \begin{bmatrix}
\alpha_f \\
\beta_f \\
\gamma_f
\end{bmatrix}
\]  

(4.2)

where \( \alpha_f \), \( \beta_f \), and \( \gamma_f \) represent the rotations about the fixed x, fixed y, and fixed z axes, respectively and each value can be given in degrees (deg) or radians (rad). EulerZYZ is given by

\[
R_e = \begin{bmatrix}
\gamma_e' \\
\beta_e' \\
\gamma_e
\end{bmatrix}
\]  

(4.3)

where \( \gamma_e \) is the rotation about the fixed z axis, \( \beta_e' \) is the rotation about the rotated y axis, and \( \gamma_e' \) is the rotation about the rotated z axis, performed in that order.

A rotation matrix \( \mathbf{R} \) specifies the orientation as a three by three orthogonal matrix. All of the representations can be used to specify the same orientation in space as

\[
\mathbf{R} = \begin{pmatrix}
c_{\gamma_f} \cdot c_{\beta_f} & c_{\gamma_f} \cdot s_{\beta_f} \cdot c_{\alpha_f} - s_{\gamma_f} \cdot s_{\alpha_f} & c_{\gamma_f} \cdot s_{\beta_f} \cdot c_{\alpha_f} + s_{\gamma_f} \cdot s_{\alpha_f} \\
s_{\gamma_f} \cdot c_{\beta_f} & s_{\gamma_f} \cdot s_{\beta_f} \cdot c_{\alpha_f} + c_{\gamma_f} \cdot s_{\alpha_f} & s_{\gamma_f} \cdot s_{\beta_f} \cdot c_{\alpha_f} - c_{\gamma_f} \cdot s_{\alpha_f} \\
-s_{\beta_f} & c_{\beta_f} \cdot s_{\alpha_f} & c_{\beta_f} \cdot c_{\alpha_f}
\end{pmatrix}
\]

(4.4)

where \( c_x \) represents the cosine of x and \( s_x \) represents the sine of x.
End-effector velocity is specified in FixedXYZ only and is given by a displacement velocity in inches per second (in/s) or centimeters per second (cm/s) and a rotational velocity in degrees per second (deg/s) or radians per second (rad/s). End-effector force/torque is also specified in FixedXYZ only. Its units are Newtons (N) for force and Newton-meters (N-m) for torque. The formal specification of the end-effector command properties is given in Table 4.9.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
</table>
| $ee\_pos\_property = "ee\_pos::{"vals,\("in\"|\"cm\"),
\',\,((\ fixed|\ euler),\("deg\"|\"rad\")|\ mat),\}"
vals = \'\,real\_val,\,2*\(',\,real\_val\')\};
fixed = "fixed:\",\,vals;
euler = \"euler:\",\,vals;
mat = \"mat:\»,\,real\_val,\,8\*(\',\,real\_val\')\};$ | $p_v^6 = \left\{ \begin{array}{c}
(R,R,R),(R,R,R) \\
(R,R,R) \\
(R,R,R)
\end{array} \right. \right\}$ |
| $ee\_vel\_property = "ee\_vel::{"vals,\("in/s\"|\"cm/s\"),
\',\,fixed,\("deg/s\"|\"rad/s\"),\}"
vals = \'\,real\_val\};
fixed = "fixed:\",\,vals;$ | $p_v^7 = \left[ (R,R,R),(R,R,R) \right]$ |
| $ee\_ft\_property = "ee\_ft::{"vals,\"N\",\n\,fixed,\"N-m\"\};$ | $p_v^8 = \left[ (R,R,R),(R,R,R) \right]$ |

Table 4.9: End-Effector Command Properties’ Values

Again, in actual application, the valid values for the properties would probably be restricted to a subset of real numbers. Particularly, the values within a rotation matrix are restricted in several ways that could be implemented in the interface if desired. For instance, all values of the rotation matrix must be between negative one and one. This example does not demonstrate these restrictions because variables and units are not the primary concern and they are an issue that has been identified as important for future work. The actions for these properties are get and set and are specified in Table 4.10.
### EBNF

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ee_pos_actions = [ee_pos_set], [ee_pos_get];</code></td>
<td>$A^b = { \text{set, get} }$</td>
</tr>
<tr>
<td><code>ee_pos_set = &quot;set::&quot;, ee_pos_property;</code></td>
<td>$a_{\text{set}}^b = p^b$</td>
</tr>
<tr>
<td><code>ee_pos_get = &quot;get::ee_pos&quot;;</code></td>
<td>$a_{\text{get}}^b = \emptyset$</td>
</tr>
<tr>
<td><code>ee_vel_actions = [ee_vel_set], [ee_vel_get];</code></td>
<td>$A^i = { \text{set, get} }$</td>
</tr>
<tr>
<td><code>ee_vel_set = &quot;set::&quot;, ee_vel_property;</code></td>
<td>$a_{\text{set}}^i = p^i$</td>
</tr>
<tr>
<td><code>ee_vel_get = &quot;get::ee_vel&quot;;</code></td>
<td>$a_{\text{get}}^i = \emptyset$</td>
</tr>
<tr>
<td><code>ee_ft_actions = [ee_ft_set], [ee_ft_get];</code></td>
<td>$A^s = { \text{set, get} }$</td>
</tr>
<tr>
<td><code>ee_ft_set = &quot;set::&quot;, ee_ft_property;</code></td>
<td>$a_{\text{set}}^s = p^s$</td>
</tr>
<tr>
<td><code>ee_ft_get = &quot;get::ee_ft&quot;;</code></td>
<td>$a_{\text{get}}^s = \emptyset$</td>
</tr>
</tbody>
</table>

### 4.2.5. Tool State

When a robot has a tool connected, the interface must have the ability to monitor and control the state of the tool. The states may not be the same for various types of tools, which adds complexity to the definition. To address several types of tools, this example assumes the ability to control the tools given in Table 4.11.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gripper</td>
<td>open/close/opening/closing</td>
</tr>
<tr>
<td>drill</td>
<td>forward/reverse/off</td>
</tr>
<tr>
<td>paint gun</td>
<td>on/off</td>
</tr>
<tr>
<td>saw</td>
<td>infinite</td>
</tr>
</tbody>
</table>

Table 4.11: Tools Identified for the Product-Line

Each individual tool is relatively simple to model as a set of property values as presented in the following subsections. It is also easy to combine the syntactic specifications of the various tool types into an overall tool state property specification as discussed in Section 4.2.5.5. However, this presents difficulty in defining the allowable actions because the syntax does not take into account which tool is currently in use. This is addressed in the semantic specification of Section 4.4.
4.2.5.1. Gripper

Grippers are extremely common tools for robots. There are several control methods used for grippers. Some grippers are controlled with position commands and feedback while others are given velocity commands. These grippers could be represented as infinite state tools like the saw in Section 4.2.5.4. Another type of gripper, which is used in this example, is commanded to either open or close. This type of gripper is common in industrial systems where the gripper is simply used for grasping and releasing objects. The state diagram for the open/close gripper is given in Figure 4.3.

![State Diagram for the Gripper](image)

The gripper is designed to both start and end in the closed state. The importance of specifying this becomes clear when semantics specification is discussed. The properties and actions for the gripper are given in Table 4.12.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>tool_st_property = &quot;tool_st::grip; grip = &quot;open&quot;</td>
<td>p^v = { open, closed, opening, closing }</td>
</tr>
<tr>
<td>tool_st_actions = [tool_st_set], [tool_st_get]; tool_st_set = &quot;set::tool_st&quot;, tool_st_get = &quot;get::tool_st&quot;;</td>
<td>A^v = { set, get }</td>
</tr>
</tbody>
</table>

Table 4.12: Tool State Property and Actions for the Gripper

While it is clear that the system never allows the user to set the state to opening or closing, it is left for the semantics to define this restriction. This shows how the semantic definition can be used to modify the syntax.
4.2.5.2. Drill

Another simple tool type is the drill. Again, drills can be modeled in various ways. In this example, the drill is represented by three states (forward, reverse, and off). Transition between forward and reverse is restricted forcing the operator to turn the tool off before reversing its direction. This is defined in the semantics and must be enforced at runtime. The state diagram for the drill is given in Figure 4.4.

![State Diagram for the Drill](image)

The property values and actions for the drill are given in Table 4.13. The actions definition is unchanged but included for completeness.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tool_st_property = &quot;tool_st::&quot;,drill;</code></td>
<td><code>p_r^g = \{fwd, rev, off\}</code></td>
</tr>
<tr>
<td>`drill = &quot;fwd&quot;</td>
<td>&quot;rev&quot;</td>
</tr>
<tr>
<td><code>tool_st_actions = [tool_st_set],[tool_st_get];</code></td>
<td><code>A^g = \{set, get\}</code></td>
</tr>
<tr>
<td><code>tool_st_set = &quot;set::&quot;,tool_st_property;</code></td>
<td><code>a_set_r^g = p_r^g</code></td>
</tr>
<tr>
<td><code>tool_st_get = &quot;get::tool_st;&quot;;</code></td>
<td><code>a_get_r^g = \O</code></td>
</tr>
</tbody>
</table>

Table 4.13: Tool State Property and Actions for the Drill

4.2.5.3. Paint Gun

The paint gun is represented as either on or off. Obviously, this is the simplest type of controllable tool, but it is a common control type for many robotic tools. The state diagram for the paint gun is given in Figure 4.5.

![State Diagram for the Paint Gun](image)

The property values and actions for the paint gun are given in Table 4.14.
Tool State Property and Actions for the Paint Gun

Table 4.14: Tool State Property and Actions for the Paint Gun

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>tool_st_property = &quot;tool_st::&quot;, paint_gun;</td>
<td>$p_r^o = {on, off}$</td>
</tr>
<tr>
<td>paint_gun = &quot;on&quot;</td>
<td>&quot;off&quot;;</td>
</tr>
<tr>
<td>tool_st_actions = [tool_st_set], [tool_st_get];</td>
<td>$A_r^o = {set, get}$</td>
</tr>
<tr>
<td>tool_st_set = &quot;set::&quot;, tool_st_property;</td>
<td>$a_set_r^o = p_r^o$</td>
</tr>
<tr>
<td>tool_st_get = &quot;get::tool_st&quot;;</td>
<td>$a_get_r^o = \emptyset$</td>
</tr>
</tbody>
</table>

Table 4.15: Tool State Property and Actions for the Saw

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>tool_st_property = &quot;tool_st::&quot;, saw;</td>
<td>$p_r^o = {0-1, off}$</td>
</tr>
<tr>
<td>saw = $[{'0'}, {'\cdot', {\text{digit}}}]-</td>
<td>'1', {'\cdot', {'0'}'}]$</td>
</tr>
<tr>
<td>tool_st_actions = [tool_st_set], [tool_st_get];</td>
<td>$A_r^o = {set, get}$</td>
</tr>
<tr>
<td>tool_st_set = &quot;set::&quot;, tool_st_property;</td>
<td>$a_set_r^o = p_r^o$</td>
</tr>
<tr>
<td>tool_st_get = &quot;get::tool_st&quot;;</td>
<td>$a_get_r^o = \emptyset$</td>
</tr>
</tbody>
</table>

4.2.5.4. Saw

Saws typically are on/off tools or consist of a finite set of states. For this example, the saw is modeled as an infinite state tool. It is a variable rate tool given by a value from zero to one, defining the fraction of its maximum rate. The saw also supports an off state. Since the saw supports infinite states, no state diagram will be given to represent the property. The values and actions can be defined, however, as in Table 4.15.

4.2.5.5. Overall Tool State Property

The overall tool state property is the combination of all the tool state types defined. For the property values, this becomes a union of the different tool types. The semantic definition of valid input based on the current tool is more complicated, however. This aspect is discussed in Section 4.4. The overall tool state property and actions are given in Table 4.16.
Defining the overall tool state property in terms of the individual components produces many instances of “off” within the possible values. This redundancy will be removed by the interface reducer.

4.2.6. Tool Change

The tool property provides monitoring and changing the active tool on robots with changeable tools. The values of the property must be customized to each system to incorporate all of the possible tools. As previously discussed, this product-line supports a gripper, drill, paint gun, and saw as shown in Figure 4.6.

This state diagram is different from others that have been presented because the operator has no direct control of any of the transitions. Instead, the operator commands the desired tool using a set action. Then the control software progresses
through the required motions to change the tool and updates the state diagram accordingly. Semantically, this diagram appears to restrict the use of the set action at any time; however, the system state diagram (see Figure 4.2) defines that set::tool is a valid action and defines when it is allowed. Table 4.17 gives the property and actions for tool change capability.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>tool::property = &quot;tool::&quot;,</td>
<td>$p_t^{10} = {\text{none}, \text{gripper}, \text{drill}, \text{paint_gun}, \text{saw}}$</td>
</tr>
<tr>
<td>(&quot;none&quot;</td>
<td>&quot;gripper&quot;</td>
</tr>
<tr>
<td>tool::actions = [tool::set], [tool::get];</td>
<td>$A_t^{10} = {\text{set}, \text{get}}$</td>
</tr>
<tr>
<td>tool::set = &quot;set::&quot;, tool::property;</td>
<td>$a_{_set_t}^{10} = p_t^{10}$</td>
</tr>
<tr>
<td>tool::get = &quot;get::tool&quot;;</td>
<td>$a_{_get_t}^{10} = \emptyset$</td>
</tr>
</tbody>
</table>

Table 4.17: Tool Property Values and Actions

4.2.7. Force/Torque Sensor

As discussed in Section 4.2.4, a force/torque sensor is likely to be used to implement end-effector force/torque control. This property exists separately for two reasons. First, end-effector force/torque control is not required to be implemented with a force/torque sensor, so this property provides access to the information at the end-effector if there is a sensor. Secondly, end-effector force/torque feedback is given in the base coordinates, meaning the actual readings are transformed before being given to the user. This property provides the actual values at the sensor in its own coordinate system. Otherwise, the property is very similar to the end-effector force/torque property including the specification of the property values.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft::sens::property = &quot;ft::sens::[&quot;vals&quot;,&quot;N&quot;,&quot;vals&quot;,&quot;N-m&quot;]&quot;;</td>
<td>$p_v^{11} = \begin{bmatrix} R &amp; R &amp; R \ R &amp; R &amp; R \end{bmatrix}$</td>
</tr>
</tbody>
</table>

Table 4.18: Force/Torque Sensor Property Values
As with any sensor, the force/torque sensor property only supports the `get` action. The definition of the `get` action is given in Table 4.19.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ft_sens_actions = [ft_sens_get];</code></td>
<td>( A^{11} = { \text{get} } )</td>
</tr>
<tr>
<td><code>ft_sens_get = &quot;get::ft_sens&quot;;</code></td>
<td>( a_{\text{get}}^{11} = \emptyset )</td>
</tr>
</tbody>
</table>

Table 4.19: Force Torque Sensor Property Actions

4.2.8. Collision Distance

To support collision detection, the user is given access to the current closest distance to a collision. This value represents the closest distance between any part of the robot and any obstacle. The user could use this information to decide when collisions are a concern. To move away from collisions, the information provided by the following property is useful. Again, collision distance cannot be directly controlled by the user so only the `get` action is provided. Table 4.20 defines the collision distance property values and the `get` action. The distance is given as a positive real number in inches or centimeters.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>`coll_dist_property = &quot;coll_dist::pos_real_val,(&quot;in&quot;</td>
<td>&quot;cm&quot;);`</td>
</tr>
<tr>
<td><code>pos_real_val = (\{digit\},[\',\{digit\}]);</code></td>
<td></td>
</tr>
<tr>
<td><code>coll_dist_actions = [coll_dist_get];</code></td>
<td>( A^{12} = { \text{get} } )</td>
</tr>
<tr>
<td><code>coll_dist_get = &quot;get::coll_dist&quot;;</code></td>
<td>( a_{\text{get}}^{12} = \emptyset )</td>
</tr>
</tbody>
</table>

Table 4.20: Collision Distance Property Values and Actions

4.2.9. Obstacle Artificial Force/Torque

The obstacle artificial force/torque property provides values of an artificial force on the end-effector that would move the robot away from obstacles. When a collision becomes imminent, this information could be used to make a decision about appropriate movements of the arm. The information is provided in the robot’s global coordinates and, thus, looks much like an actual end-effector force/torque property.
reading. The obstacle artificial force/torque property values and get action are given in Table 4.21.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ob} _\text{ft} _\text{property} = &quot;\text{ob} _\text{ft}[:&quot;\text{vals}, &quot;N,&quot;, \text{fixed}, &quot;N-m&quot;]&quot;; )</td>
<td>( p^3_{13} = \left( \begin{array}{ccc} \mathbb{R} &amp; \mathbb{R} &amp; \mathbb{R} \end{array} \right) )</td>
</tr>
<tr>
<td>( \text{ob} _\text{ft} _\text{actions} = \left[ \text{ob} _\text{ft} _\text{get} \right]; )</td>
<td>( A^{13} = { \text{get} } )</td>
</tr>
<tr>
<td>( \text{ob} _\text{ft} _\text{get} = \text{&quot;get::ob}_ft&quot;; )</td>
<td>( a _\text{get}_{13} = \emptyset )</td>
</tr>
</tbody>
</table>

Table 4.21: Obstacle Artificial Force/Torque Property Values and Actions

4.2.10. Summary

The collection of properties and actions identified in the previous sections describes the domain model for the robotics domain example. As a reference for the following sections, the domain model is summarized here algebraically. Table 4.22 and Table 4.23 contain the property names, property values, actions, and brief descriptions for the properties in the domain model. Also, Appendix A contains the entire EBNF and algebraic specifications. Chapter 6 discusses benefits and drawbacks of constructing a domain model in this manner and evaluates the feasibility of other approaches.

<table>
<thead>
<tr>
<th>Property</th>
<th>Values</th>
<th>Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p^1 = \text{sys} )</td>
<td>( p^1_v = { \text{inactive, enabled, idle, shutdown, estop, homing, tool val hold, cancel, resume, error} _S #, \text{error} _L #, \text{error} _E #, \text{error} _T #, \text{error} _C #, \text{error} _clear } )</td>
<td>( A^{1} = { \text{set, get} } )</td>
<td>provides access to the current state of the system including initialization, shutdown, and errors</td>
</tr>
<tr>
<td>( p^2 = j _\text{pos} )</td>
<td>( p^2_v = (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) )</td>
<td>( A^{2} = { \text{set, get} } )</td>
<td>provide access to the joint position, velocity, torque, and current</td>
</tr>
<tr>
<td>( p^3 = j _\text{vel} )</td>
<td>( p^3_v = (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) )</td>
<td>( A^{3} = { \text{set, get} } )</td>
<td></td>
</tr>
<tr>
<td>( p^4 = j _\text{torq} )</td>
<td>( p^4_v = (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) )</td>
<td>( A^{4} = { \text{set, get} } )</td>
<td></td>
</tr>
<tr>
<td>( p^5 = j _\text{curr} )</td>
<td>( p^5_v = (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) )</td>
<td>( A^{5} = { \text{set, get} } )</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.22: Domain Model Summary for the Robotics Example (Properties 1-5)
4.3. Feature Specification

The previous section presented an informal domain analysis resulting in the identification of properties, property values, and actions for a robotics domain. The next step is to formally define the features of the domain in terms of this domain model. The domain model was constructed from informal features identified for RCS generation. The formally defined features in this section provide the same overall functionality as the informal features that were used. The feature set for the robotics product family (defined by the domain expert) is given by

\[
F = \{val\_move, j\_pos, j\_vel, j\_torq, j\_curr, ee\_pos, ee\_vel, ee\_ft, error\_check, grip, tool, coll\_detect\}
\] (4.5)

Each of these features is formally defined in the following subsections. The formal feature specification is used for selection of features in order to synthesize an
interface to a product. A brief description of the features is provided in Table 4.24 and the entire EBNF and algebraic feature specifications are given in Appendix A.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Motion</td>
<td>adds the val state to the basic system state property to facilitate value based motion</td>
</tr>
<tr>
<td>Joint Position Command</td>
<td>allows control and monitoring of robotic joint positions</td>
</tr>
<tr>
<td>Joint Velocity Command</td>
<td>allows control and monitoring of robotic joint velocities</td>
</tr>
<tr>
<td>Joint Torque Command</td>
<td>allows control and monitoring of robotic joint torques</td>
</tr>
<tr>
<td>Joint Current Command</td>
<td>allows control and monitoring of robotic joint currents</td>
</tr>
<tr>
<td>End-Effector (EE) Position Command</td>
<td>allows control and monitoring of robotic EE positions and encompasses multiple EE orientation representations</td>
</tr>
<tr>
<td>EE Velocity Command</td>
<td>allows control and monitoring of robotic EE velocities</td>
</tr>
<tr>
<td>EE Force/Torque Command</td>
<td>allows control and monitoring of robotic EE forces and torques and encompasses a force/torque sensor</td>
</tr>
<tr>
<td>Error Checking</td>
<td>encompasses limits checking to filter commands beyond hardware limits and excess error checking to filter rates beyond hardware capability</td>
</tr>
<tr>
<td>Gripper</td>
<td>gives the robot control of a gripper</td>
</tr>
<tr>
<td>Tool Control</td>
<td>all robots in the product-line either have no tool, a gripper only, or can change between all the defined tools as encompassed by this feature</td>
</tr>
<tr>
<td>Collision Detection</td>
<td>monitors distance from potential collisions of any part of the robot and provides artificial force values at the EE signaling safeness of movements</td>
</tr>
</tbody>
</table>

Table 4.24: Formally Specified Features

4.3.1. Value Motion

The value motion feature is developed to help with the specification of the various command features. Each of the command features requires the basic system state as well as the val state. The value motion feature encompasses the system state portion and, therefore, lays the groundwork for value based motion. One of the command features must be selected however in order to actually achieve motion. The state machine representing the value feature is given in Figure 4.7.
The specification of the value motion feature is given in Table 4.25.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val_move_feature = sys_actions;</code></td>
<td></td>
</tr>
<tr>
<td>`move_t = &quot;homing&quot;</td>
<td>&quot;val&quot;;`</td>
</tr>
<tr>
<td>`err_t = &quot;error&quot;, (&quot;S&quot;, digit</td>
<td>&quot;clear&quot;);`</td>
</tr>
<tr>
<td><code>\hat{p}_t = p_t - \{v_{7}, v_{13}, v_{14}, v_{15}, v_{16}\}</code></td>
<td><code>f_1 = (\hat{p}_t, A_t)</code></td>
</tr>
</tbody>
</table>

Table 4.25: Value Motion Feature

4.3.2. Joint Position, Velocity, Torque, and Current Command

The joint command features all have an associated property which they encompass along with the get and set actions. In order to support motion commands, the system must also include the basic system state as well as the `val` state. Therefore, the features also include the value motion feature. This provides the system with the capability to startup, shutdown, and allow value commands. When a joint command is given, the associated property is modified appropriately and the system state is
placed into the val state until the motion is completed. Formal specification of the joint command features is given in Table 4.26.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j _ pos _ feature = j _ pos _ actions, val _ move _ feature; )</td>
<td>( f_2 = (p^2, A^2) \cup f_1 )</td>
</tr>
<tr>
<td>( j _ vel _ feature = j _ vel _ actions, val _ move _ feature; )</td>
<td>( f_3 = (p^3, A^3) \cup f_1 )</td>
</tr>
<tr>
<td>( j _ torq _ feature = j _ torq _ actions, val _ move _ feature; )</td>
<td>( f_4 = (p^4, A^4) \cup f_1 )</td>
</tr>
<tr>
<td>( j _ curr _ feature = j _ curr _ actions, val _ move _ feature; )</td>
<td>( f_5 = (p^5, A^5) \cup f_1 )</td>
</tr>
</tbody>
</table>

Table 4.26: Joint Command Features

4.3.3. End-Effector Position, Velocity, and Force/Torque Command

The end-effector command features are virtually identical to the joint command features. Again, they include their associated properties and actions as well as the same subset of the system state property and its actions. The specification of the end-effector command features is given in Table 4.27.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ee _ pos _ feature = ee _ pos _ actions, val _ move _ feature; )</td>
<td>( f_6 = (p^6, A^6) \cup f_1 )</td>
</tr>
<tr>
<td>( ee _ vel _ feature = ee _ vel _ actions, val _ move _ feature; )</td>
<td>( f_7 = (p^7, A^7) \cup f_1 )</td>
</tr>
<tr>
<td>( ee _ ft _ feature = ee _ ft _ actions, val _ move _ feature; )</td>
<td>( f_8 = (p^8, A^8) \cup f_1 )</td>
</tr>
</tbody>
</table>

Table 4.27: End-Effector Command Features

4.3.4. Error Checking

The error checking feature adds the ability to check commands for limits errors and excess errors. At the interface level, the only effect is the addition of the associated errors to the system state property. While error checking is not useful unless at least one of the joint or end-effector command features is included in the system, it does not fundamentally require them. The specification of the error checking feature is given in Table 4.28.
4.3.5. Gripper

In this product-line, all robots have no tool, a gripper only, or can handle all of the defined tools. This feature provides the ability to control the gripper only. It, therefore, includes the gripper portion of the tool state property and does not require any of the tool change property. It is specified as in Table 4.29.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_check_feature = sys_actions;</td>
<td>( f_9 = (\hat{p}^1, A^1) )</td>
</tr>
<tr>
<td>move_t = &quot;homing&quot;;</td>
<td>( \hat{p}^1 = p^1_v - {v^1_1, v^1_8, v^1_{15}, v^1_{16}} )</td>
</tr>
<tr>
<td>err_t = &quot;error&quot;, ( {('S'</td>
<td>'L'</td>
</tr>
</tbody>
</table>

Table 4.28: Error Checking Feature

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>grip_feature = tool_st_actions;</td>
<td>( f_{10} = (\hat{p}^g, A^g) )</td>
</tr>
<tr>
<td>tool_st_property = &quot;tool_st::&quot;, grip;</td>
<td>( \hat{p}^g = p^g_v - {v^g_1, v^g_2, v^g_3, v^g_4, v^g_9} )</td>
</tr>
</tbody>
</table>

Table 4.29: Gripper Feature

4.3.6. Tool Control

The tool control feature supports the ability to change among all of the defined tools and gives control of the tool state regardless of the current tool. As such, it encompasses the tool change property and actions, as well as, the tool state property and actions. Also, the ability to change tools requires the basic system state property with the tool state and the tool error functionality. specification is given in Table 4.30.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>tool_feature = tool_st_actions,tool_actions, sys_actions;</td>
<td>( f_{11} = (p^9, A^9) \cup (p^{10}, A^{10}) \cup (\hat{p}^1, A^1) )</td>
</tr>
<tr>
<td>move_t = &quot;homing&quot;</td>
<td>&quot;tool&quot;;</td>
</tr>
<tr>
<td>err_t = &quot;error&quot;, ( {('S'</td>
<td>'T'),)digit&quot;clear&quot;;</td>
</tr>
</tbody>
</table>

Table 4.30: Tool Control Feature
4.3.7. Collision Detection

The collision detection feature includes the ability to retrieve the closest distance to an obstacle and the artificial forces to move the robot away from obstacles. Also, the system state property is included with collision error reporting. The specification is given in Table 4.31

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>coll_detect_feature = coll_dist_actions, ob Atkins, sys actions; move_t = &quot;homing&quot;; err_t = &quot;error_&quot;, (&quot;S&quot;</td>
<td>&quot;C&quot;), digit</td>
</tr>
</tbody>
</table>

Table 4.31: Collision Detection Feature

4.4. Semantic Relationship Specification

This section presents the semantics of this product-line. Each of the three types of semantic restriction is addressed independently in the following subsections. As presented in Section 3.2, the three types of semantic restrictions include physical relationships, time-based relationships, and conditional relationships. The full semantic restriction specification is given in Appendix A.

4.4.1. Physical Relationships

As discussed in Section 3.2.2, the robotics domain includes physical relationships among its joint and end-effector properties. Specifically, the geometry of the robot directly relates the end-effector values to those of the joints. For instance, the relationship between joint velocity ($\dot{\theta}$) and end-effector velocity ($v_{\text{ef}}$) is formed by the geometry based Jacobian matrix ($J(\theta)$) as

$$v_{\text{ef}} = J(\theta) \cdot \dot{\theta} \tag{4.6}$$

From this relationship and similar relationships, the set actions on the various end-effector properties are restricted from use with the corresponding joint properties’ set actions. Another physical restriction is the relationship between current sent to an
actuator and the torque exerted by the actuator. The electromechanical construction of
the actuator causes the output torque to increase as the current is increased. Therefore,
these two properties cannot be modified simultaneously. The physical relationships
identified for the product-line are given in Table 4.32

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>set :: ee_pos - set :: j_pos;</td>
<td>$a_set^6 - a_set^2$;</td>
</tr>
<tr>
<td>set :: ee_vel - set :: j_vel;</td>
<td>$a_set^7 - a_set^3$;</td>
</tr>
<tr>
<td>set :: ee_ft - set :: j_torq;</td>
<td>$a_set^8 - a_set^4$;</td>
</tr>
<tr>
<td>set :: j_curr - set :: j_torq;</td>
<td>$a_set^5 - a_set^4$;</td>
</tr>
</tbody>
</table>

Table 4.32: Physical Semantic Restrictions

4.4.2. Time-Based Relationships

As presented in Section 3.2.3, time-based relationships can exist within an
interface specification, such as the relationship between joint velocity ($\dot{\theta}$) as the
derivative of joint position ($\theta$) with respect to time ($t$) given by

$$\dot{\theta} = \frac{d\theta}{dt} \tag{4.7}$$

While the interface specification presented in this report does not contain a
time component, time-based relationship restrictions are enforced for simplicity. For
instance, joint velocity and position could be set simultaneously, but if the velocity
sends a joint in the opposite direction of the goal position, problems arise. To avoid
these difficulties, Table 4.33 defines time-based restrictions for the product-line.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>set :: j_vel - set :: j_pos;</td>
<td>$a_set^1 - a_set^2$;</td>
</tr>
<tr>
<td>set :: j_torq - set :: j_vel;</td>
<td>$a_set^4 - a_set^3$;</td>
</tr>
<tr>
<td>set :: j_torq - set :: j_pos;</td>
<td>$a_set^4 - a_set^3$;</td>
</tr>
<tr>
<td>set :: ee_vel - set :: ee_pos;</td>
<td>$a_set^7 - a_set^6$;</td>
</tr>
<tr>
<td>set :: ee_ft - set :: ee_vel;</td>
<td>$a_set^8 - a_set^7$;</td>
</tr>
<tr>
<td>set :: ee_ft - set :: ee_pos;</td>
<td>$a_set^8 - a_set^6$;</td>
</tr>
</tbody>
</table>

Table 4.33: Time-Based Semantic Restrictions
Additional restrictions can also be identified based on the combination of the time-based and physical relationships. For instance, since joint current is directly related to joint torque, which in turn is directly related to end-effector force/torque, joint current is directly related to end-effector force/torque. The consequence is that each of the joint and end-effector properties can only be controlled independently. The additional restrictions needed are given in Table 4.34.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{set} :: j _ \text{curr}</td>
<td>\text{set} :: ee _ \text{vel}</td>
</tr>
<tr>
<td>\text{set} :: j _ \text{pos};</td>
<td></td>
</tr>
<tr>
<td>( \text{set} :: j _ \text{curr}</td>
<td>\text{set} :: ee _ \text{pos}</td>
</tr>
<tr>
<td>\text{set} :: j _ \text{vel};</td>
<td></td>
</tr>
<tr>
<td>( \text{set} :: ee _ \text{pos}</td>
<td>\text{set} :: ee _ \text{vel} )−</td>
</tr>
<tr>
<td>\text{set} :: j _ \text{torq};</td>
<td></td>
</tr>
<tr>
<td>( \text{set} :: ee _ \text{pos}</td>
<td>\text{set} :: ee _ \text{vel}</td>
</tr>
<tr>
<td>\text{set} :: j _ \text{curr};</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.34: Physical/Time-Based Semantic Restrictions

4.4.3. Conditional Relationships

As discussed in Section 3.2.4, conditional relationships indicate the validity of actions based on the current state of the system. A good method of determining conditional relationships is to construct state diagrams to represent the system. In Section 4.2, the properties of the product-line were presented and several of them included state diagrams. However, only the system state diagram and tool state diagrams affect the semantics. The actions identified for other properties are either restricted by one of these two diagrams or they are not restricted at all. The following subsections present condition restrictions for this product-line.

4.4.3.1. System State

The system state diagram is the most important state diagram for determining the validity of actions. It monitors the overall system and restricts actions on many of
the properties based on the state of the overall system. For reference, the system state diagram is duplicated in Figure 4.8.

Figure 4.8: State Diagram for the Complete System State Property

To specify the semantic restrictions from this diagram, each of the labeled transitions must be addressed. The specifications of the transitions caused by system state actions are given by Equation (4.8) and Equation (4.9).
set :: sys :: enabled × (sys :: inactive | sys :: estop) → sys :: enabled;
set :: sys :: enabled × sys :: shutdown → sys :: idle;
set :: sys :: shutdown × (sys :: enabled | sys :: idle) → sys :: shutdown;
set :: sys :: estop × → sys :: estop;
set :: sys :: homing × sys :: enabled → sys :: homing;
set :: sys :: hold × (sys :: homing | sys :: tool | sys :: val) → sys :: hold;
set :: sys :: cancel × (sys :: hold | sys :: homing | sys :: tool | sys :: val) →
( sys :: enabled | sys :: idle);
set :: sys :: resume × sys :: hold → (sys :: homing | sys :: tool | sys :: val);
set :: sys :: error _##× ¬sys :: estop → sys :: error _##;
set :: sys :: error _clear × sys :: error _## → ¬sys :: estop;

\[ \begin{align*}
    a_{set_{i2}}^{l} & \times (v_{i1}^{l} | v_{5}^{l}) \rightarrow v_{2}^{l}; \\
a_{set_{i2}}^{l} & \times v_{4}^{l} \rightarrow v_{3}^{l}; \\
a_{set_{i4}}^{l} & \times (v_{2}^{l} | v_{3}^{l}) \rightarrow v_{4}^{l}; \\
a_{set_{i5}}^{l} & \times \rightarrow v_{5}^{l}; \\
a_{set_{i6}}^{l} & \times v_{2}^{l} \rightarrow v_{6}^{l}; \\
a_{set_{i9}}^{l} & \times (v_{6}^{l} | v_{7}^{l} | v_{8}^{l}) \rightarrow v_{9}^{l}; \\
a_{set_{i10}}^{l} & \times (v_{6}^{l} | v_{7}^{l} | v_{8}^{l}) \rightarrow (v_{2}^{l} | v_{3}^{l}); \\
a_{set_{i11}}^{l} & \times v_{3}^{l} \rightarrow (v_{6}^{l} | v_{7}^{l} | v_{8}^{l}); \\
(a_{set_{i12}}^{l} | a_{set_{i13}}^{l} | a_{set_{i14}}^{l} | a_{set_{i15}}^{l} | a_{set_{i16}}^{l}) & \times \neg v_{5}^{l} \rightarrow
(v_{12}^{l} | v_{13}^{l} | v_{14}^{l} | v_{15}^{l} | v_{16}^{l}); \\
a_{set_{i17}}^{l} & \times (v_{12}^{l} | v_{13}^{l} | v_{14}^{l} | v_{15}^{l} | v_{16}^{l}) \rightarrow \neg v_{5}^{l};
\end{align*} \]

(4.9)

The other transitions in the diagram are caused by actions performed on other properties. This is important to the stability of the system because many of the properties do not have inherent restrictions on their validity. The system state diagram ensures that they are only modified when the system is in an appropriate state. For example, the joint position property does not include any restrictions on valid values because the system can move from any position to any other. From a system
perspective, however, the system must be in the idle state before such a command can be given. These restrictions are given by Equation (4.10) and Equation (4.11).

\[
\text{set} :: (j\_\text{pos} | j\_\text{vel} | j\_\text{torq} | j\_\text{curr}) \times \text{sys} :: \text{idle} \rightarrow \text{sys} :: \text{val};
\]
\[
\text{set} :: (ee\_\text{pos} | ee\_\text{vel} | ee\_\text{ft}) \times \text{sys} :: \text{idle} \rightarrow \text{sys} :: \text{val}; \tag{4.10}
\]
\[
\text{set} :: \text{tool} \times \text{sys} :: \text{idle} \rightarrow \text{sys} :: \text{tool};
\]

\[
\begin{align*}
(a\_\text{set}^2 | a\_\text{set}^3 | a\_\text{set}^4 | a\_\text{set}^5) & \times v_3 \rightarrow v_6^1; \\
(a\_\text{set}^6 | a\_\text{set}^7 | a\_\text{set}^8) & \times v_3^1 \rightarrow v_6^1; \\
 a\_\text{set}^{10} & \times v_3^1 \rightarrow v_7^1;
\end{align*} \tag{4.11}
\]

With these restrictions defined, there are four properties whose actions have not yet been restricted. These are tool state, force/torque sensor, collision distance, and obstacle artificial force/torque. The latter three do not include a set action and therefore need no further restriction. Tool state is addressed in the following section.

4.4.3.2. Tool State

Various tool state diagrams were presented in Section 4.2.5. Based on the current tool being used by the robot, the valid actions change. Therefore, a relationship exists between the current state of the tool change property and the valid actions on the tool state property. This relationship is modeled by combining the various state machines for the individual properties into a single state machine that describes both properties. This combined state machine is given in Figure 4.9.
Figure 4.9: Combined State Diagram for the Tool State and Tool Properties

Each of the individual tool state diagrams has been placed within a superstate representing the tool change state that activates the given diagram. A transition has been specified from each tool state diagram to the state that represents no tool on the robot. This transition defines the exiting condition that was previously identified by the final state symbol. Lastly, several transitions have been added in order to allow the user to specify a tool change regardless of the current power state. When a tool change action is given, the system transitions into the appropriate tool state before continuing the tool change. Therefore, the tool change property and its actions are
unaffected by this modified state diagram. The semantic restrictions identified for the tool state property are given by

\[
\begin{align*}
\text{set} :: & \text{tool } \text{st} :: \text{off} \times \text{tool } \text{st} :: \text{drill} \& (\text{tool } \text{st} :: \text{fwd} \mid \text{tool } \text{st} :: \text{rev}) \rightarrow \\
& \quad \text{tool } \text{st} :: \text{off}; \\
\text{set} :: & \text{tool } \text{st} :: \text{off} \times \text{tool } \text{st} :: \text{paint} \_ \text{gun} \& \text{tool } \text{st} :: \text{on} \rightarrow \text{tool } \text{st} :: \text{off}; \\
\text{set} :: & \text{tool } \text{st} :: \text{off} \times \text{tool } \text{st} :: \text{saw} \& \text{tool } \text{st} :: \# \# \# \rightarrow \text{tool } \text{st} :: \text{off}; \\
\text{set} :: & \text{tool } \text{st} :: \text{on} \times \text{tool } \text{st} :: \text{paint} \_ \text{gun} \& \text{tool } \text{st} :: \text{off} \rightarrow \text{tool } \text{st} :: \text{on}; \\
\text{set} :: & \text{tool } \text{st} :: \text{fwd} \times \text{tool } \text{st} :: \text{drill} \& \text{tool } \text{st} :: \text{off} \rightarrow \text{tool } \text{st} :: \text{fwd}; \\
\text{set} :: & \text{tool } \text{st} :: \text{rev} \times \text{tool } \text{st} :: \text{drill} \& \text{tool } \text{st} :: \text{off} \rightarrow \text{tool } \text{st} :: \text{rev}; \\
\text{set} :: & \text{tool } \text{st} :: \text{on} \times \text{tool } \text{st} :: \text{gripper} \& \\
& (\text{tool } \text{st} :: \text{closed} \mid \text{tool } \text{st} :: \text{closing}) \rightarrow \text{tool } \text{st} :: \text{opening}; \\
\text{set} :: & \text{tool } \text{st} :: \text{closed} \times \text{tool } \text{st} :: \text{gripper} \& \\
& (\text{tool } \text{st} :: \text{open} \mid \text{tool } \text{st} :: \text{opening}) \rightarrow \text{tool } \text{st} :: \text{closing}; \\
\text{set} :: & \text{tool } \text{st} :: \# \# \# \times \text{tool } \text{st} :: \text{saw} \& \text{tool } \text{st} :: \text{off} \rightarrow \text{tool } \text{st} :: \# \# \#; \\
\end{align*}
\]

The same restrictions are specified algebraically by

\[
\begin{align*}
\text{a} \_ \text{set} & v_1^9 \times v_3^{10} \& (v_3^9 \mid v_4^9) \rightarrow v_1^9; \\
\text{a} \_ \text{set} & v_1^9 \times v_4^{10} \& v_2^9 \rightarrow v_1^9; \\
\text{a} \_ \text{set} & v_1^9 \times v_5^{10} \& v_9^9 \rightarrow v_1^9; \\
\text{a} \_ \text{set} & v_2^9 \times v_4^{10} \& v_1^9 \rightarrow v_2^9; \\
\text{a} \_ \text{set} & v_3^9 \times v_3^{10} \& v_1^9 \rightarrow v_3^9; \\
\text{a} \_ \text{set} & v_4^9 \times v_3^{10} \& v_1^9 \rightarrow v_4^9; \\
\text{a} \_ \text{set} & v_5^9 \times v_2^{10} \& (v_6^9 \mid v_8^9) \rightarrow v_7^9; \\
\text{a} \_ \text{set} & v_6^9 \times v_2^{10} \& (v_5^9 \mid v_7^9) \rightarrow v_6^9; \\
\text{a} \_ \text{set} & v_9^9 \times v_5^{10} \& v_1^9 \rightarrow v_9^9;
\end{align*}
\]

4.5. Summary

This chapter demonstrates specification of the syntax and semantics for a relatively simple robotics domain. Specification of the syntax entails identifying properties, actions, and features to create the domain model. The informal domain
analysis performed does not effectively demonstrate the process of domain analysis, but the goal was to provide an example of the expected results.

The resulting domain model includes thirteen properties, forty-two property values (counting each enumeration or range as one value), and twenty-three actions and is given in Appendix A. Also, twelve features were identified for the product family. This example thus demonstrates the complexity of creating an interface for even relatively straightforward domains. For example, assuming that there are no restrictions on feature combinations, there are $2^{12} - 1$ possible product instances that can be generated from this specification where each instance has a unique software interface. Fortunately, this approach provides a systematic specification technique and an algebraic definition allowing consistent generation of an interface for a product instance within the domain. The domain expert produces an EBNF specification of the domain in a well-defined manner and then the interface generator infers an algebraic specification for reduction and optimization of the interface.

Typically, programming interfaces are developed with much less structure and the design is, therefore, much more tedious and subjective. Particularly, when two systems within a domain are independently designed, it is difficult to provide a consistent interface. So, while the approach defined in this report may not be the final answer, this robotics example demonstrates the complexity that this research is attempting to reduce. Following a well-defined systematic and algebraic approach to interface specification should reduce the effort required and decrease the subjectivity of interface creation for individual products and, to an even greater extent, multiple products within a given domain.
Chapter Five

5. Interface Generation and Representation

This chapter presents initial examples of using a domain interface specification to automatically generate the interface for a product instance within the product-line. The methods presented in this chapter have not been formalized and tested and are, therefore, presented as possible approaches that should be evaluated in future research. All of the methods are implemented manually for this example. The first section of this chapter presents interface generation as the selection of features and their combination into an optimizable specification of the interface to a particular product. The second section presents interface translation into desired representations. Examples are given using the robotic product-line specification of Chapter 4.

5.1. Interface Generation

Interface generation includes several steps to produce a complete EBNF specification of the interface to a particular product. The steps are: feature selection, feature addition, interface reduction, and interface optimization. Each of these steps is addressed in the following subsections.

5.1.1. Feature Selection

With the interface components of a product-line specified, the first step of product interface generation is to select the features desired for a product instance. To this point, the specifications have been developed by a domain expert but the rest of
the steps are intended to be performed automatically or by a user with less knowledge of the product-line. Ultimately, an EBNF specification of the interface to the product is desired, but the user need not manually provide the specification. For instance, a Graphical User Interface (GUI) for feature selection could be provided by the interface generator because the specification of the product-line is known. The GUI might look like Figure 5.1.

![Feature Selection GUI](image)

Figure 5.1: Feature Selection GUI

This GUI is a conceptual design and does not necessarily represent a working software implementation. The GUI gives a selectable list of features (from the robotics product-line in the previous chapter) obtained from the product-line specification, and it provides the interface details of the selected feature. The user can then analyze the contents of each feature and select the desired features. The five features selected in the GUI (joint position, end-effector position, error checking, tool
control, and collision detection) are used in the examples given in this chapter for
demonstration of interface synthesis. The value motion feature is included with a
grayed check box because it is implicitly included in the interface by other features.
This selection is given by the representations of Table 5.1.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>robot_interface = j_pos_feature, ee_pos_feature, error_check_feature, tool_feature, coll_detect_feature;</td>
<td>( I = {f_2, f_6, f_9, f_{11}, f_{12}} )</td>
</tr>
</tbody>
</table>

Table 5.1: Product Interface Specification

The EBNF specification is all that the user must specify. In this case, it would
be easy to do it manually, but more complex systems might use a GUI interface to
generate the EBNF interface as discussed earlier. The algebraic interface specification
is inferred by the interface generation system based on the EBNF specification. The
rest of the interface generation steps presented are performed automatically by the
interface generation system based on information provided.

5.1.2. Feature Interface Addition

A simple approach to interface generation would simply add together the
EBNF interface specifications of the selected features. This would result in a large
interface with many redundancies and would be extremely inefficient and difficult to
translate into various representations. The approach presented in this report begins
with this type of interface addition and a starting point for the reduction and
optimization steps. Interface addition is part of the automatic sequence of interface
synthesis and is thus performed in the algebraic specification only.

Interface addition is a simple step that entails combining individual interfaces
for each of the features that comprise the product instance. The process starts with the
algebraic specification of Table 5.1 and appends the specification of each feature. In
this example, the joint position feature is the first part to be added resulting in
\[ I = \{ f_2, f_6, f_9, f_{11}, f_{12} \} \]

\[ f_2 = (p^2, A^2) \cup f_1 \]

Features are defined in terms of the domain model but an interface is an independent entity so the properties and actions making up each feature must also be included. Adding this information for the joint position feature gives

\[ I = \{ f_2, f_6, f_9, f_{11}, f_{12} \} \]

\[ f_2 = (p^2, A^2) \cup f_1 \]

\[ p^V = (R, R, R, R, R, R) \]

\[ A^V = \{ set, get \} \]

\[ a_{set}^V = p^V \]

\[ a_{get}^V = \emptyset \]

Because the joint position feature is defined in terms of the value motion feature, it is the next component added. Its specification, given in Equation (5.3) is simply appended to the end of Equation (5.2).

\[ f_1 = (\hat{p}^I, A^I) \]

\[ \hat{p}_V = p_I - \{ v_{17}, v_{18}, v_{19}, v_{20}, v_{21} \} \]

\[ p^I = \{ \text{inactive, enabled, idle, shutdown, estop, homing, tool, val, hold, cancel, resume, error } \_S \#, \text{error } \_L \#, \text{error } \_E \#, \text{error } \_T \#, \text{error } \_C \#, \text{error } \_clear \} \]

\[ A^I = \{ \text{set, get} \} \]

\[ a_{set}^I = p^I \]

\[ a_{get}^I = \emptyset \]

Each feature is added to the interface in this manner until the interface is completely defined as a self contained specification. The entire interface specification for this robotics product as an addition of all its features is given in Appendix B.1.
The following two steps take this interface specification and reduce and then optimize it. To demonstrate the effectiveness of these steps, the following metrics are used.

- **Algebraic statements** – The algebraic specification consists of interface, feature, property, and action definition statements. Reduction removes redundant statements from the interface.

- **Action combinations** – The interface consists of a set of actions that can be included in a communication. Many action combinations are not allowed by the semantic restrictions. Optimization removes invalid action combinations.

After feature addition, the interface contains one interface definition statement, six feature definition statements, fourteen property definition statements, and twenty-eight action definition statements giving a total of forty-nine algebraic statements. The following sections discuss reduction and optimization of this interface with the results summarized in Table 5.2.

<table>
<thead>
<tr>
<th>Step</th>
<th>Algebraic Statements</th>
<th>Action Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature interface addition</td>
<td>49</td>
<td>4096</td>
</tr>
<tr>
<td>Interface reduction</td>
<td>28</td>
<td>4096</td>
</tr>
<tr>
<td>Interface optimization</td>
<td>28</td>
<td>1152</td>
</tr>
</tbody>
</table>

**Table 5.2: Interface Metrics**

### 5.1.3. Interface Reduction

The interface reduction step takes the specification produced by the previous step and simplifies the interface as much as possible. Features are only relevant as a means for selection of interface components. As such, one of the goals of interface reduction is to remove the references to features and instead produce the interface as a collection of property-action tuples as in

$$I = \{(p_1, A^1), (p^2, A^2), (p^6, A^6), (p^9, A^9), (p^{10}, A^{10}), (p^{12}, A^{12}), (p^{13}, A^{13})\} \quad (5.4)$$

This action was performed by simply collecting all of the property-action tuples of the interface. Next, redundancies are removed. There are two types of
redundancy that are removed. The first deals with multiple specifications of the same entity. These are easily identified because they occur when two statements in the interface have the same value on the left side of the equals sign.

Four of the features used in this example include a subset of the system state property and its actions. Therefore the interface includes four slightly different definitions of $\hat{p}_V^i$. To remove this redundancy, a union is performed between all of the instances of $\hat{p}_V^i$. This is valid because they all refer to the same property and if two features include different subsets of the property values, the interface should include the combination of the features’ subsets. The four instances of $\hat{p}_V^i$ are given for features $f_1$, $f_9$, $f_{11}$, and $f_{12}$, respectively as

$$\hat{p}_V^i = p_v - \{v_7, v_{13}, v_{14}, v_{15}, v_{16}\}$$

$$\hat{p}_V^i = p_v - \{v_7, v_6, v_{14}, v_{16}\}$$

$$\hat{p}_V^i = p_v - \{v_8, v_{13}, v_{14}, v_{16}\}$$

$$\hat{p}_V^i = p_v - \{v_7, v_6, v_{13}, v_{14}, v_{15}\}\quad (5.5)$$

In this case, when the union is performed, $\hat{p}_V^i$ becomes equivalent to $p_v^i$ as

$$\hat{p}_V^i = p_v^i\quad (5.6)$$

Also, the interface includes four identical instances of $p_v^i$, $A^i$, $a_{set}^i$, and $a_{get}^i$. Again, a union is performed resulting in the elimination of excess specifications. The other type of redundancy that is removed deals with feature specific property and action specifications, such as $\hat{p}_V^i$, as they are related to the domain level property. Since features are no longer relevant, feature specific properties and actions (none in this example) are reduced into the property itself. The system state property is redefined as

$$p_v^i = \hat{p}_V^i \cap p_v^i\quad (5.7)$$
The \( \cap \) is the intersection symbol and defines that the property is redefined as the intersection between the subset of the domain property defined by the selected features and the overall domain property. Because \( \hat{p}_v^1 \) is defined as a subset of \( p_v^1 \), Equation (5.7) is equivalent to

\[
p_v^1 = \hat{p}_v^1
\]  

(5.8)

The feature specific property reference in the interface specification is thus replaced resulting in

\[
I = \{(p^1, A^1), (p^2, A^2), (p^6, A^6), (p^9, A^9), (p^{10}, A^{10}), (p^{12}, A^{12}), (p^{13}, A^{13})\} \quad (5.9)
\]

The resulting reduced interface is given in Appendix B.2. It has been reduced to one interface definition statement, zero feature definition statements, eight property definition statements, and nineteen action definition statements resulting in twenty-eight total algebraic statements. This represents a 43% reduction in the interface specification.

### 5.1.4. Interface Optimization

After interface reduction, the product interface is an algebraic representation of the desired features. The interface optimization step serves two purposes. The interface is converted back into the human readable, EBNF syntax and the defined semantic relationships are simultaneously applied to optimize the interface. Optimization is the process of making a system as effective and efficient as possible. The reduced interface is a valid interface for the product, but it can be improved on. Semantic restrictions have been defined that could be enforced at runtime, but it would be more effective and efficient if they are enforced in the syntax of the interface. Not all semantic restrictions can be applied to the syntax, particularly conditional restrictions, but the optimization process applies all that can be.

The algebraic specification cannot represent semantic restrictions because it was formed solely for specification of properties, actions, and features. It only
describes what commands are allowed and cannot address the conditions under which they are allowed. As such, the interface optimization simultaneously translates the reduced algebraic specification into EBNF and applies the semantic relationships. Ultimately, the semantic restrictions that can be applied to the syntax specify conflicts between actions. The conflicts can be removed by modifying the syntax such that the conflicting actions cannot be included in a single communication. Interface optimization has not been thoroughly explored in this research and is a topic to consider for future work. However, for completeness, potential applications of physical, time-based, and conditional semantic restrictions are demonstrated in this section.

5.1.4.1. Conditional Semantic Relationship Analysis

Conditional semantic relationships provide information about the runtime performance of the system. As such, conditional relationship specifications are structured in a manner that makes this efficient. At runtime, the system simply searches the conditional relationship specifications for the incoming action and verifies that the system is in one of the states that makes it valid. For optimization of the interface syntax, however, this specification syntax is not very useful. The first step in optimization is, therefore, to extract a more concrete definition of syntactic restrictions from the conditional relationship specifications.

Two types of restrictions can be extracted from the conditional semantics. The first type of restriction specifies whether particular actions are ever valid in a communication. Properties will often have values that the user is not allowed to directly command the system into. For instance, in the robotics domain, the system state property contains a value called `inactive` which is only entered when the system is powered on. So, while it is a legitimate value to receive in response to a `get` action, it will never be the argument of a `set` action. This is clear in the semantic specification of the system state actions given by (as defined in Section 4.4.3.1)
These semantic relationships fully define the validity of system state actions and they do not include \( a_{\text{set}}^{1} \), \( a_{\text{set}}^{3} \), \( a_{\text{set}}^{7} \), or \( a_{\text{set}}^{8} \) which correspond to set::sys::inactive, set::sys::idle, set::sys::tool, and set::sys::val. Therefore, these actions are never allowed in a communication as defined in the extracted restriction of Table 5.3.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\text{set :: sys :: inactive;})</td>
<td>(-a_{\text{set}}^{1} );</td>
</tr>
<tr>
<td>(-\text{set :: sys :: idle;})</td>
<td>(-a_{\text{set}}^{3} );</td>
</tr>
<tr>
<td>(-\text{set :: sys :: tool;})</td>
<td>(-a_{\text{set}}^{7} );</td>
</tr>
<tr>
<td>(-\text{set :: sys :: val;})</td>
<td>(-a_{\text{set}}^{8} );</td>
</tr>
</tbody>
</table>

Table 5.3: System State Action Restrictions

An identical analysis on the tool state property actions reveals the restrictions in Table 5.4.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\text{set :: tool :: st :: opening;})</td>
<td>(-a_{\text{set}}^{9} );</td>
</tr>
<tr>
<td>(-\text{set :: tool :: st :: closing;})</td>
<td>(-a_{\text{set}}^{9} );</td>
</tr>
</tbody>
</table>

Table 5.4: Tool State Action Restrictions
The other type of restriction that can be extracted from the conditional semantics specification is based on the interrelationship between some properties. Some actions on one property require another property to be in a particular state. Obviously, it would not be acceptable in this case to allow both properties to change state simultaneously because the action may not be valid in the new state. In the robotics domain, many actions cause a change in the system state property. Also, the validity of tool state actions is dependent on the current tool, so the tool state and tool properties cannot change simultaneously. These semantic specifications are given by

\[
\begin{align*}
(a_{\text{set}}^2 | a_{\text{set}}^3 | a_{\text{set}}^4 | a_{\text{set}}^5) &\times v_3^1 \rightarrow v_8^1; \\
(a_{\text{set}}^6 | a_{\text{set}}^7 | a_{\text{set}}^8) &\times v_3^1 \rightarrow v_8^1; \\
a_{\text{set}}^{10} &\times v_3^1 \rightarrow v_8^1; \\
\end{align*}
\]

(5.11)

\[
a_{\text{set}}^9 \times v_3^{10} \land (v_3^9 | v_4^9) \rightarrow v_8^1;
\]

The extracted restrictions are given in Table 5.5.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle \text{set} :: \text{j} _ \text{pos} \mid \text{set} :: \text{j} _ \text{vel} \rangle - \text{set} :: \text{sys};)</td>
<td>(\langle a_{\text{set}}^2 \mid a_{\text{set}}^3 \rangle - a_{\text{set}}^1;)</td>
</tr>
<tr>
<td>(\langle \text{set} :: \text{j} _ \text{torq} \mid \text{set} :: \text{j} _ \text{curr} \rangle - \text{set} :: \text{sys};)</td>
<td>(\langle a_{\text{set}}^4 \mid a_{\text{set}}^5 \rangle - a_{\text{set}}^1;)</td>
</tr>
<tr>
<td>(\langle \text{set} :: \text{ee} _ \text{pos} \mid \text{set} :: \text{ee} _ \text{vel} \rangle - \text{set} :: \text{sys};)</td>
<td>(\langle a_{\text{set}}^6 \mid a_{\text{set}}^7 \mid a_{\text{set}}^8 \rangle - a_{\text{set}}^1;)</td>
</tr>
<tr>
<td>(\text{set} :: \text{tool} - \text{set} :: \text{sys};)</td>
<td>(a_{\text{set}}^{10} - a_{\text{set}}^1;)</td>
</tr>
<tr>
<td>(\text{set} :: \text{tool} _ \text{st} - \text{set} :: \text{tool};)</td>
<td>(a_{\text{set}}^9 - a_{\text{set}}^{10};)</td>
</tr>
</tbody>
</table>

Table 5.5: Conditional Restrictions

However, these are not all of the restrictions that can be extracted from the conditional semantics. Clearly, when a property’s *set* action causes changes in another properties state, the two cannot be included together. But, this is also true for two different properties’ *set* actions that cause a change in a third property. For instance, a joint position *set* action causes the system state property to change and so does a tool property *set* action. The two actions cannot be included together because
they might cause conflicting changes in the system state property. The additional restrictions because of this type of relationship are given in Table 5.6.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{set :: } j _ \text{pos}</td>
<td>\text{set :: } j _ \text{vel} )</td>
</tr>
<tr>
<td>( \text{set :: } j _ \text{torq}</td>
<td>\text{set :: } j _ \text{curr} )</td>
</tr>
<tr>
<td>( \text{set :: } ee _ \text{pos}</td>
<td>\text{set :: } ee _ \text{vel} )</td>
</tr>
<tr>
<td>( \text{set :: } ee _ \text{ft} )</td>
<td>( \text{a _set}^9 ) - ( \text{a _set}^{10} );</td>
</tr>
</tbody>
</table>

**Table 5.6: Additional Conditional Restrictions**

With the conditional semantic specification redefined in a manner that is conducive to optimization, the process of forming the optimized EBNF specification can begin.

### 5.1.4.2. Optimized EBNF Specification

The core components of the reduced interface are given by

\[
I = \{(p^1, A^1), (p^2, A^2), (p^6, A^6), (p^9, A^9), (p^{10}, A^{10}), (p^{12}, A^{12}), (p^{13}, A^{13})\}
\]

\[
P = \{\text{sys, } j \_ \text{pos, } ee \_ \text{pos, } tool \_ \text{st, } tool, \text{coll} \_ \text{dist, } ob \_ \text{ft}\} \quad (5.12)
\]

The semantic restrictions have no effect on the specification of the properties included in the interface, so the first step is to translate the property specifications into EBNF. With the details left out because of space considerations, the result is

\[
\begin{align*}
\text{sys \_ property} &= \"\text{sys:}\", \text{sys \_ t}\; \\
\text{j \_ pos \_ property} &= \"j \_ pos:}(\", \text{real \_ val}, 5^*({', \text{real \_ val}, }), (\text{\"deg\" | \"rad\")}) \\
\text{ee \_ pos \_ property} &= \"ee \_ pos:}[", \text{vals.} (\text{\"in\" | \"cm\")}, ', ' \\
& \quad ((\text{\"fixed \_ euler\")}, (\text{\"deg\" | \"rad\") | \text{\"mat\")}); \\
\text{tool \_ st \_ property} &= \"tool \_ pow:}\", (\text{grip | drill | paint \_ gun | saw}); \\
\text{tool \_ property} &= \"tool:\", (\text{\"none\" | \"gripper\" | \"drill\" | \"paint \_ gun\" | \"saw\")}; \\
\text{coll \_ dist \_ property} &= \"coll \_ dist:\", \text{pos \_ real \_ val,}(\text{\"in\" | \"cm\")}; \\
\text{ob \_ ft \_ property} &= \"ob \_ ft:}[", \text{vals,} \text{\"N\", \text{\"fixed\" | \text{\"N-m\")}]; \\
\end{align*}
\]
With the properties specified, the actions are translated into EBNF. Each action is individually placed into the interface syntax after searching the semantic restrictions for restrictions on that action. Each action will be discussed in this section in the order it is included in the interface. The first set of actions for the interface is the system state actions and the set action is addressed first. Because there are no other actions added to the interface yet, the system state set action can have no conflicts, so it is simply inserted as

\[
\text{robot\_interface} = [\text{sys\_set}]; \tag{5.14}
\]

Next, the specification of the \text{sys\_set} action is added. As presented in the previous section, there are four arguments to the system state set action that are never valid and are, therefore, removed from the interface. As such, the specification becomes

\[
\text{robot\_interface} = [\text{sys\_set}]; \\
\text{sys\_set} = "set:\", \text{sys\_property} = ("sys:\", ("inactive" | "idle" | "tool" | "val")); \tag{5.15}
\]

Next, the system state get action and its specification are added to the interface. None of the get actions have any restrictions defined, so the get action will not be discussed for the other properties. The result of the addition of the get action is

\[
\text{robot\_interface} = [\text{sys\_set}], [\text{sys\_get}]; \\
\text{sys\_set} = "set:\", \text{sys\_property} = ("sys:\", ("inactive" | "idle" | "tool" | "val")); \\
\text{sys\_get} = "get::sys"; \tag{5.16}
\]

The next actions to be added to the interface are those for the joint position property. As discussed in the previous section, a set action on the joint position causes a change in the state of the system state property. As such, the set actions for the two properties must be used independently. This is implemented in EBNF by making them options in a list resulting in (only the specification of the components being discussed include details)
The end-effector position actions are very similar to the joint position actions. In fact one semantic restriction defines that both the end-effector position set and joint position set must be independent from the system state set. At this point, however, the added action must address all actions already in the interface. A separate semantic restriction defines that joint position set and end-effector position set cannot be used simultaneously. The semantic restrictions that affect end-effector position are given by

\[
\text{robot} \_ \text{interface} = [\text{sys} \_ \text{set} \mid \text{j} \_ \text{pos} \_ \text{set}], [\text{sys} \_ \text{get}], [\text{j} \_ \text{pos} \_ \text{get}];
\]

\[
\text{j} \_ \text{pos} \_ \text{set} = \text{"set::","j} \_ \text{pos} \_ \text{property};
\]

\[
\text{j} \_ \text{pos} \_ \text{get} = \text{"get::j} \_ \text{pos"};
\]

Since none of the three set actions thus far can be included in the same communication, they are all grouped into an option list by

\[
\text{robot} \_ \text{interface} = [\text{sys} \_ \text{set} \mid \text{j} \_ \text{pos} \_ \text{set} \mid \text{ee} \_ \text{pos} \_ \text{set}],
\]

\[
[\text{sys} \_ \text{get}], [\text{j} \_ \text{pos} \_ \text{get}], [\text{ee} \_ \text{pos} \_ \text{get}];
\]

\[
\text{ee} \_ \text{pos} \_ \text{set} = \text{"set::","ee} \_ \text{pos} \_ \text{property};
\]

\[
\text{ee} \_ \text{pos} \_ \text{get} = \text{"get::ee} \_ \text{pos"};
\]

The tool state actions are the next to be translated into EBNF. In relation to the already discussed actions, the tool state set action has no restrictions so it can be used with any of the actions in the choice. However, as discussed in the previous section, two of the tool state set action arguments are restricted from the interface. The tool state actions are thus added by
robot_interface = [sys_set | j_pos_set | ee_pos_set], [tool_st_set]
[sys_get], [j_pos_get], [ee_pos_get], [tool_st_get];
tool_st_set = "set::", tool_st_property =  
("tool_st::", ("opening" | "closing"));
tool_st_get = "get::tool_st";

The last property that has a set action is the tool property. The previous section defined that the tool property set action must be used independently of all the set actions that have been added to the interface. This is accomplished by simply forcing a decision between the tool property set action and the other set actions in the interface. This is given by (the two other properties’ actions are also added)

robot_interface =
[[sys_set | j_pos_set | ee_pos_set], [tool_st_set] | tool_set]
[sys_get], [j_pos_get], [ee_pos_get], [tool_st_get],
[tool_get], [coll_dist_get], [ob_ft_get];
tool_set = "set::", tool_property;
tool_get = "get::tool";
coll_dist_get = "get::coll_dist";
ob_ft_get = "get::ob_ft";

(5.20)

(5.21)

With this step, the interface optimization is complete. The entire interface has been specified in EBNF and the interface has been made as effective as possible based on the semantics specification. The entire optimized interface specification is given in Appendix B.3. Because interface optimization is not intended to make the syntax more concise, like interface reduction, a different metric must be used to evaluate the effectiveness. Interface optimization attempts to remove conflicting actions from the interface. Without optimization, the interface contains twelve actions that can be combined in any fashion in a single communication resulting in $2^{12}$ (or 4096) possible communications (not considering different action arguments). An analysis of the optimized interface reveals that there are only 1152 possible
communications meaning that 2944 invalid action combinations have been removed from the syntax of the interface. The next section addresses translation of the interface into various representations.

5.2. Interface Translation

Interface translation takes the optimized EBNF specification and allows translation into various representations. The EBNF specification is a consistent specification of the interface as a collection of terminals and special EBNF symbols that are strung together through non-terminals that have no effect on the specification. The non-terminals have no effect on the specification because they represent an abstraction of EBNF that provides increased human readability. Therefore, the EBNF specification could be written as a single sentence consisting solely of terminals and EBNF symbols. This would result in a difficult to read specification but translation is easier to present when the interface specification is thought of in this manner. The beginning of the example interface specification would appear as

\[
\begin{align*}
\text{robot interface} = & \left[ ("set::sys::", \"inactive\", \ldots) | ("set::j_pos::", \ldots) | \ldots, \ldots \right] \ldots
\end{align*}
\]  

Clearly, this would not produce a very pleasant interface, but the reason translation is approached in this manner is the naming and granularity of non-terminals cannot be predicted. In this form, some details are known about the appearance of the specification. When the interface specification is given in this way, the translation need only specify how to handle EBNF special symbols and the terminals defined for the particular product-line interface. Two methods of translating the specification are given here. Neither has been formalized, and they are given only as possible approaches. They include parsing the specification in order and parsing hierarchically.

5.2.1. Sequential Parsing

One method of parsing the specification is to analyze each symbol in order, searching for patterns for which translations have been specified. The patterns might
be specified in a table and could consist of any combination of individual symbols, letters, or domain terminals. A good example of a complex pattern is the `real_val` non-terminal. The translator would look for the complex pattern and translate it into the desired representation. An example using C++ is given in Table 5.7.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>C++ Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[+</code>,<code>-</code>],{<code>0</code></td>
<td><code>1</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>void set_j_pos</code></td>
</tr>
<tr>
<td><code>enum angle{deg,rad};</code></td>
<td></td>
</tr>
<tr>
<td><code>set_j_pos(3.2,1.2,4.5,4.3,6.3,9.7,deg);</code></td>
<td><code>set_j_pos</code></td>
</tr>
</tbody>
</table>

### Table 5.7: Representation of `real_val` in C++

In this manner, additional ordered symbols could be specified as in Table 5.8 to define the interface to the joint position `set` action.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>C++ Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;set::j_pos::(&quot;</td>
<td><code>void set_j_pos(</code></td>
</tr>
<tr>
<td>`','</td>
<td><code>,</code> <code>enum angle{deg,rad};</code></td>
</tr>
<tr>
<td>`&quot;deg&quot;</td>
<td>&quot;rad&quot;)`</td>
</tr>
</tbody>
</table>

### Table 5.8: Representation of `j_pos_set` Components in C++

C++ is a complex language, so this is a simplified example. It is assumed that the placement of the C++ statements is known; however, actual application of these concepts would require some type of C++ specification beyond just the text to insert. Also, this simply represents the interface to the C++ code in the form of a function prototype. The actual implementation of the function is beyond the scope of this report. From Table 5.7 and Table 5.8, the parser would recognize the EBNF specification of the joint position `set` action and produce the representation in C++ as in Table 5.9.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;set::j_pos::(&quot; ,real_val,5*(&quot; ,real_val ),&quot;)' , (&quot;deg&quot;</td>
<td>&quot;rad&quot;)</td>
</tr>
<tr>
<td>Usage</td>
<td><code>set_j_pos(3.2,1.2,4.5,4.3,6.3,9.7,deg);</code></td>
</tr>
</tbody>
</table>

### Table 5.9: Representation of `j_pos_set` in C++
For space consideration, \textit{real_val} is left as a non-terminal; whereas, in application the translator would recognize the EBNF specification as discussed above. It might be better to specify the six float values as a vector and use structs instead of enumerations, but this example shows the process of building a C++ interface from the interface specification. For completeness, Appendix C.1 includes a full C++ interface for the product, as well as, an example of its usage.

5.2.2. Hierarchical Parsing

Another approach to translation would be to provide a hierarchical definition. While non-terminals specify an easily noticeable hierarchical structure to the specification, their removal does not remove the hierarchical structure of EBNF. The defined symbols for EBNF can be used to easily specify the hierarchy. This section uses an XML representation to present this approach.

First, the single sentence specification will have a single semicolon representing the interface specification. Therefore, no matter what is included in the specification, an encompassing representation can be provided as in Table 5.10.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx;</td>
<td><code>&lt;?xml version=&quot;1.0&quot; encoding=&quot;utf-8&quot; ?&gt;\n  &lt;xs:schema targetNamespace=&quot;robot&quot;\n    elementFormDefault=&quot;qualified&quot; xmlns=&quot;robot&quot;\n    xmlns:xs=&quot;http://www.w3.org/2001/XMLSchema&quot;&gt;\n    &lt;xs:element name=&quot;robot&quot;&gt;\n      xxx\n    &lt;/xs:element&gt;\n  &lt;/xs:schema&gt;</code></td>
</tr>
</tbody>
</table>

Table 5.10: Representation of Basic Interface in XML Schema

With a shell defined for the interface representation, various patterns and groups are addressed. Taking a hierarchical look at the optimized product interface specification of Appendix B.3, the first pattern found is the separation of components by commas. This specifies the separation of components of the interface that are

\[^4\] The notations xxx, yyy, ..., etc. are used to represent as yet unidentified components in the specification. With a hierarchical approach, the big picture is seen first and then the translator delves into the details of the specification.
placed within an *all* element in XML, which means they may appear in any order.
The translation is shown in Table 5.11.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx, yyy,...</td>
<td><code>&lt;xs:complexType&gt;</code>&lt;br&gt;<code>&lt;xs:all&gt;</code>&lt;br&gt;<code>xxx</code>&lt;br&gt;<code>yyy</code>&lt;br&gt;<code>...</code>&lt;br&gt;<code>&lt;/xs:all&gt;</code>&lt;br&gt;<code>&lt;/xs:complexType&gt;</code></td>
</tr>
</tbody>
</table>

Table 5.11: Representation of Comma Separated Components in XML Schema

The next hierarchical component found is the bracket symbols defined for EBNF. Since the bracket symbols represent an optional component, the XML specification simply adds a *minOccurs* attribute to the component, allowing it to not appear at all as in Table 5.12.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[xxx]</td>
<td><code>minOccurs=&quot;0&quot;</code></td>
</tr>
</tbody>
</table>

Table 5.12: Representation of Bracket Symbols in XML Schema

Much like comma separated values, options in a list are specified as in Table 5.13 using a *choice* block.

```xml
<xs:complexType>
  <xs:choice>
    xxx
    yyy
    ...
  </xs:choice>
</xs:complexType>
```

Table 5.13: Representation of List Options in XML Schema

The previous translations are very general to EBNF but it is unlikely that such generalities will never have exceptions. In fact, when analyzing the grouping of *set* actions in the interface specification a pattern arises that is a special case. It is given by
Based on the previously defined patterns, the tendency of the translator would be to translate this into

```
<xs:complexType>
  <xs:choice>
    xxx
    <xs:complexType>
      <xs:all>
        yyy
        zzz
      </xs:all>
    </xs:complexType>
  </xs:choice>
</xs:complexType>
```

This is not a valid XML schema component, however, because an `all` element is not allowed to be a child of a `choice` element. When problems like this arise, special cases can be identified. This case would be solved with the translation of Table 5.14.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
</table>
| `xxx | yyy, zzz,...`          | `<xs:complexType>
  <xs:choice>
    xxx
    <xs:sequence>
      yyy
      zzz
      ...
    </xs:sequence>
  </xs:choice>
</xs:complexType>` |

Table 5.14: Representation of Option Special Case in XML Schema

The translation specification continues with terminals because they are the next level in the hierarchy. In this case, actions and properties can be treated the same in an XML schema representation, and their translation is given in Table 5.15.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&quot;xxx::&quot;</code></td>
<td><code>&lt;xs:element name=&quot;xxx&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/xs:element&gt;</code></td>
</tr>
</tbody>
</table>

Table 5.15: Representation of Actions and Properties in XML Schema
Finally, property values are addressed. Each type of property value has a different representation, each of which must be specified. For brevity, only enumerated state values are described here, but the representation chosen for each type can be seen in the complete XML schema representation of Appendix C.2.1. Enumerated states are translated with the pattern given in Table 5.16.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;xxx&quot;</td>
<td>&quot;yyy&quot;</td>
</tr>
</tbody>
</table>

Table 5.16: Representation of Enumerated States in XML Schema

The combination of these translations and the others not listed can be seen in Appendix C.2.1 along with examples of the usage of the components in Appendix C.2.2. While this interface was assembled manually, the combination was performed in a way that adhered to the specifications given. The result, therefore, is not well written XML but it is functional. Future work should add additional description to the translation allowing more control of the outcome.

5.3. Summary

This chapter discussed the process of synthesizing a product interface from the properties, actions, and features defined for a domain. Features are chosen from those available for the product-line. From this selection, the algebraic specification of the interface is formed by addition of the selected features. For the robotics example, this resulted in forty-nine algebraic statements defining the interface. The interface is then reduced to remove redundancies from the specification and to simplify the overall interface. This produced a 43% reduction in the robotics interface to twenty-eight algebraic statements. Next, the reduced algebraic interface is simultaneously converted back into an EBNF specification and optimized using the semantic
specification. For the robotics domain, the reduced interface presented 4096 possible communications of which 2944 (72%) were invalid regardless of system state. Interface optimization removed all of these invalid action combinations resulting in an interface syntax allowing 1152 possible communications.

With the EBNF specification of the interface completed, it is possible to translate the interface into various interface representations. This process has not been fully formalized, but this chapter demonstrates its feasibility. Two conceptual approaches to interface translation were presented including a sequential approach and a hierarchical approach. The effectiveness of these approaches has not been fully evaluated, but it does appear that the hierarchical approach has benefits in greater structural control and reuse among different product-line specifications. Future work should further address the use of the specification of properties and actions to generate a product interface.
Chapter Six

6. Summary and Conclusions

Software interfaces are often considered as the physical interface through which a user interacts with an application. This report requires a distinction to be made between the software interface and the representation of the interface. The software interface defines the inputs and outputs of a system in a well-defined algebraic specification, whereas the physical interface is just one representation of the software interface. A system consists of a single software interface but may have many distinct interface representations. This research develops an approach to interface specification based on Feature Oriented Programming (FOP) that allows algebraic reduction, optimization, and translation into desired representations.

FOP is a software engineering concept in which features are composed to build software systems. This report extends the concept to software interfaces by defining the software interface to features of a system in terms of properties and actions. This is done through domain analysis and product-line analysis by a domain expert. With the domain specified, interface generation is automatically performed after features are selected. Then the interface is translated into any desired representation such as C++ or eXtensible Markup Language (XML) schemas. All of these steps are detailed in the following sections.
6.1. Approach

The primary goal of this report is to provide an approach for software interface generation as an impetus for a research thread can begin that may ultimately produce a complete interface generation scheme. The approach set forth in this report consists of four primary steps. They are: domain analysis, product-line analysis, interface generation, and interface translation. The following sections discuss each of these steps.

6.1.1. Domain Analysis and Product-Line Analysis

Fundamentally, domain analysis and product-line analysis are two separate steps. Practically, however, they are very similar and may be discussed together. Domain analysis is intended to provide a general look at the domain for which product interfaces are to be synthesized. In other words, if a company produces several product-lines within a single domain, it would be useful to perform a domain analysis first to discover aspects that are common among the entire domain. The detail that can be specified at the domain level, however, is likely not to be sufficient for any particular product-line. For this reason, the product-line analysis is considered separately. The goal of both the domain analysis and product-line analysis is to specify the interface syntax and semantics. From these, interfaces can be generated and translated into desired representations.

6.1.1.1. Interface Syntax

Interface syntax defines the structure of an interface. It gives the valid inputs and outputs of the system. In order to formalize this process, properties, actions, and features are defined. Properties are components of the domain that consist of a single value at any given time that can be monitored or modified by a user. Actions are defined as the mechanisms by which the user controls (or monitors) the property. Features are the building blocks for a product line and their interface is specified in terms of properties and actions.
This report defines a human-readable specification for interface syntax based on Extended Backus-Naur Form (EBNF) for programming language specification. Using this well-defined method, a domain expert can specify the properties of an interface like the alarm state property given in Equation (6.1) that specifies the possible states of a simple alarm clock.

\[
\text{alarm\_state\_property} = \text{"alarm\_state::", } \left( \text{"on"|"off"|"sounding"|"snooze"} \right);
\]  

Similarly, the actions can be specified for the property. Actions are defined in a general way so that a domain expert may customize without limits the actions defined for a property. However, it is believed that a set action and a get action will typically be sufficient to specify the interface for modification and monitoring of the property. As such, the actions for the alarm state property are given by

\[
\begin{align*}
\text{alarm\_state\_actions} &= [\text{alarm\_state\_set}, \text{alarm\_state\_get}] ; \\
\text{alarm\_state\_set} &= \text{"set::", alarm\_state\_property} ; \\
\text{alarm\_state\_get} &= \text{"get::alarm\_state"}; 
\end{align*}
\]  

The last portion of the interface syntax is the specification of features’ interfaces. Features’ interfaces are simply specified as collections of properties and actions representing tangible features of the product-line. An obvious feature of the alarm clock domain is an alarm feature providing the ability to set the alarm time and control the state of the alarm. Also, the alarm feature includes the time feature which provides the ability to set the time of the clock itself. This is given by Equation (6.3); for details, see Chapter 3.

\[
\begin{align*}
\text{alarm\_feature} &= \text{time\_feature, alarm\_time\_actions, alarm\_state\_actions}; \\
\text{alarm\_time\_actions} &= ; \\
\text{alarm\_state\_actions} &= ; \\
\text{alarm\_feature} &= \text{time\_feature, alarm\_time\_actions, alarm\_state\_actions}; \\
\end{align*}
\]  

The EBNF syntax provides a straightforward mechanism for a domain expert to specify the interface syntax. It is also structured enough for machine interpretation. However, for interface generation, reduction, and optimization, an algebraic
specification that is more conducive to these processes was also developed. This algebraic specification can be generated from the EBNF specification and the interface synthesizer can perform all of its operations behind the scenes so that the domain expert and product user need not bother with the details of the algebraic specification. This is fortunate because the algebraic specification does not contain the same level of human-readability but is appropriate for mathematical reasoning.

The alarm state property discussed above is specified algebraically by

\[ P = \{ \text{time, alarm\_time, alarm\_state} \} \]
\[ p^3_v = \{ \text{on, off, sounding, snooze} \} \] (6.4)

Similarly, the actions are given by

\[ A^3 = \{ \text{set, get} \} \]
\[ a_{\text{set}}^3_v = p^3_v \]
\[ a_{\text{set}}^3_v = \emptyset \] (6.5)

Finally, the alarm feature discussed is given by

\[ F = \{ \text{time, alarm, snooze} \} \]
\[ f_2 = f_1 \cup \left( p^2, A^3 \right) \cup \left( \hat{p}^3, \hat{A}^3 \right) \] (6.6)

The alarm clock example is used throughout Chapter 3 in the definition of the concepts of properties, actions, and features. As a more robust example of the application of these principles, a robotics domain analysis is performed in Chapter 4. This analysis resulted in the specification of thirteen properties with a combined total of forty-two property values and twenty-three actions. Also, twelve features were defined. For reference, the properties identified are given in the following list and the entire interface specification (EBNF and algebraic) can be found in Appendix A.
A. System State  
B. Joint Position  
C. Joint Velocity  
D. Joint Torque  
E. Joint Current  
F. End-Effector Position  
G. End-Effector Velocity  
H. End-Effector Force/Torque  
I. Tool State  
J. Tool Change  
K. Force/Torque Sensor  
L. Collision Distance  
M. Obstacle Artificial Force/Torque

Clearly, many more properties could be defined for the robotics domain in general, but this example shows that a relatively complex system can be modeled with a reasonable number of properties, actions, and features.

6.1.1.2. Interface Semantics

Interface semantics provides context to the interface syntax. Within an interface, there are two major contextual considerations for the communication. The first deals with conflicts in concurrent commands to the system and the second addresses validity of commands based on the current system state. In order to identify these restrictions, this report considers three different types of semantic analysis.

Physical semantics helps to identify conflicts among simultaneous commands by identifying the physical relationships between properties. Within the robotics domain, for instance, end-effector velocity \( (v_{\text{eff}}) \) is directly related to joint velocity \( (\dot{\theta}) \) through the geometry based Jacobian matrix \( (J(\theta)) \) as

\[
v_{\text{eff}} = J(\theta) \cdot \dot{\theta}
\]

While exceptions may occur, in general this relationship restricts the joint velocity and end-effector velocity of a robot from being set concurrently. As such, a restriction is defined between the two properties’ set actions by

\[
\text{set :: ee\_vel} - \text{set :: j\_vel};
\]

Algebraically, the same restriction is given by
The full syntax for this type of restriction is given by

\[
\text{restriction} = \left[ \text{or}_1, ' + ' | ' - ', \left( \text{or}_2 \right) \right] ; \left( \text{or}_3 \right) ;
\]

\[
\text{or}_1 = \text{and}_1, ' | ', \left( \text{and}_2 \right) ;
\]

\[
\text{and}_1 = \text{group}, ' & ', \left( \text{group} \right) ;
\]

\[
\text{group} = \left[ - \right], \left( \text{action} | ('', \text{or}_4, ', ') \right) ;
\]

These restrictions help to define the behavior of the syntax for the optimizer. As such, the syntactic specification can be altered so that this conflict is not ever allowed. This is called optimization of the interface because the interface is viable without this process being executed, but it provides improved efficiency and effectiveness to the interface specification. Because each property is specified independently, it is not possible for the domain expert to make these accommodations manually. Instead, it must be done after the interface features are combined through this well-defined specification of the restrictions.

Time-based semantic relationships are similar to physical relationships but the restriction is only necessary when a time component is included in the interface. For instance, joint velocity is the time derivative of joint-position ($\theta$)

\[
\dot{\theta} = \frac{d\theta}{dt}
\]  

(6.11)

When no time component is included in the interface, it is valid to set both joint position and joint velocity as long as the joint velocity values result in motion in the direction of the desired joint positions. In this case, the system could move at the specified velocities with each joint stopping when it reaches the goal position. However, if a time is given for the action to take place, the position and velocity become linked and cannot be set independently. The syntax of the restriction is virtually identical to that of physical relationships.

The last type of semantic restriction is based on conditional relationships. Conditional relationship analysis can also provide restrictions to modify the syntax,
but the primary purpose is to provide runtime data about the validity of commands. Properties are defined as sets of values which the user can monitor or modify. When the set of values is finite, the property can typically be modeled as a state machine. With this type of representation, it becomes relatively simple to define the conditional restrictions. A simple example within the robotics domain is the power state for a three state drill given in Figure 6.1.

In this diagram, the drill can be off, on in forward motion, or on in reverse motion. The tool is restricted so that the user cannot switch directly from forward to reverse or vice versa. This restriction cannot be defined syntactically so it is defined for runtime enforcement. A different syntax is used for conditional restrictions that is given by

\[
\text{restriction} = \text{action or list}, 'x', \left[ \text{state or list} \right], ';', '\to', \text{state or list}, ';';
\]

\[
\text{action or list} = \text{action and list}, '\mid', \text{action and list};
\]

\[
\text{action and list} = \text{action group}, '(', \text{action group};
\]

\[
\text{action group} = \left[ \right], \left( \text{action} \mid (', \text{action or list}, ') \right)
\]

\[
\text{state or list} = \text{state and list}, '\mid', \text{state and list};
\]

\[
\text{state and list} = \text{state group}, '(', \text{state group};
\]

\[
\text{state group} = \left[ \right], \left( \text{state} \mid (', \text{state or list}, ') \right)
\]

The restrictions for the tool state example are given by

\[
\begin{align*}
\text{set :: tool st :: off} & \times \text{tool :: drill} \& (\text{tool st :: fwd} \mid \text{tool st :: rev}) \rightarrow \\
& \text{tool st :: off}; \\
\text{set :: tool st :: fwd} & \times \text{tool :: drill} \& \text{tool st :: off} \rightarrow \text{tool st :: fwd}; \\
\text{set :: tool st :: rev} & \times \text{tool :: drill} \& \text{tool st :: off} \rightarrow \text{tool st :: rev};
\end{align*}
\]

Similarly, the same restrictions are given algebraically by
The format chosen for the specification of this runtime information is conducive to rapid analysis of incoming data based on the current state. The interface simply searches the conditional semantic specification for the incoming command. When found, the allowed states are immediately available, so the interface simply verifies that the system is currently in one of the valid states before executing the command. With the semantic analysis complete, the domain expert has fully specified the interface components and interface generation can begin.

### 6.1.2. Interface Generation

Interface generation combines the interfaces to the features selected for a product within the product-line into an optimized specification. This report does not formalize interface generation because it requires significant research beyond what was afforded at this time. Instead, this report discusses possible approaches for using the formal interface specifications defined to produce a final interface to a product. The ultimate goal of interface generation is to combine selected features into an EBNF specification that can be translated automatically into various representations.

This process has been presented in four major steps: feature selection, interface addition, interface reduction, and interface optimization. Each of these is discussed in the following sections.

#### 6.1.2.1. Feature Selection

The domain and product-line analyses serve to produce a list of features for the product-line. To begin the interface generation process, a set of these features must be selected as those desired for the product. This selection is fundamentally specified in EBNF as in

\[
\begin{align*}
\text{a_set}_{v_1}^9 \times v_3^{10} \land \left( v_3^9 \mid v_4^9 \right) & \rightarrow v_1^9; \\
\text{a_set}_{v_3}^9 \times v_3^{10} \land v_1^9 & \rightarrow v_3^9; \\
\text{a_set}_{v_4}^9 \times v_3^{10} \land v_1^9 & \rightarrow v_4^9;
\end{align*}
\] (6.14)
\[\text{robot\_interface} = j\_\text{pos\_feature}, ee\_\text{pos\_feature},
\text{error\_check\_feature}, tool\_\text{feature}, coll\_\text{detect\_feature};\] (6.15)

This is the interface specification for the robotics example given in Chapter 5. As discussed, this EBNF specification might not be developed manually. During the domain and product-line analyses, the assumption has been made that a domain expert is specifying the interface. At this point, it is likely that a less informed person might make the decisions about included features. As such, it is likely that a more user-friendly interface might be given. It is simple to conceive of a Graphical User Interface (GUI) that reads the EBNF specification of the interface syntax and provides information about features to aid in selection. Such a GUI might appear like Figure 6.2.

![Figure 6.2: Feature Selection GUI](image-url)
This interface provides in an understandable manner all of the information included in the interface syntax specification. However the features are selected, the algebraic equivalent is required before continuing to the next steps. This could be generated from the EBNF specification and, for this example, is given by

\[ I = \{ f_2, f_6, f_9, f_{11}, f_{12} \} \]  \hspace{1cm} (6.16)

Next, the combination of these features is addressed followed by interface reduction and then optimization. It will be shown that feature interface addition results in a large interface with many redundancies. Reduction will remove these redundancies and greatly simplify the interface. Optimization will then remove a large number of invalid action combinations making the interface more efficient. The metrics used to measure the effectiveness are:

- Algebraic statements – The algebraic specification consists of interface, feature, property, and action definition statements. Reduction removes redundant statements from the interface.
- Action combinations – The interface consists of a set of actions that can be included in a communication. Many action combinations are not allowed by the semantic restrictions. Optimization removes invalid action combinations.

The results of interface reduction and optimization for the robotics example are summarized in Table 6.1.

<table>
<thead>
<tr>
<th>Step</th>
<th>Algebraic Statements</th>
<th>Action Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature interface addition</td>
<td>49</td>
<td>4096</td>
</tr>
<tr>
<td>Interface reduction</td>
<td>28</td>
<td>4096</td>
</tr>
<tr>
<td>Interface optimization</td>
<td>28</td>
<td>1152</td>
</tr>
</tbody>
</table>

Table 6.1: Interface Metrics

6.1.2.2. Feature Interface Addition

Feature addition is an extremely simple combination of the features. On paper, it really serves little purpose but to put all of the information in one place. For
implementation, it represents a critical step of collecting the data into whatever data structure is deemed appropriate. Basically, the interface generation approach presented in this report begins with this simple addition as a starting point for reduction and optimization. The resulting interface, if given in its EBNF form, is a viable interface but it is bloated with redundancies and contains no consideration for optimized performance.

Feature addition is an intermediate step in the interface generation process and is therefore performed using the algebraic interface specification. The process starts with the product interface definition given in Equation (6.16) and appends the specification of each feature. Because the interface is considered an independent entity, the feature specification includes all of the necessary information including property and action specifications. After addressing the first feature of the interface, the robotics example interface is given by

\[
\begin{align*}
I &= \{f_2, f_6, f_9, f_{11}, f_{12}\} \\
f_2 &= (p^2, A^2) \cup f_1 \\
p^2 &= (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) \\
A^2 &= \{\text{set, get}\} \\
a_\text{set}^2 &= p^2 \\
a_\text{get}^2 &= \emptyset
\end{align*}
\]

(6.17)

Clearly, this is a trivial exercise of data collection. Appendix B contains the full robotics interface specification after addition as well as the specification after reduction and optimization. At this point, the robotics interface consists of forty-nine algebraic statements, many of which will be removed by reduction in the next step.

6.1.2.3. Interface Reduction

Interface reduction is designed to simplify the interface as much as possible. The first simplification that can be made is the elimination of feature references
because features are only intended as a means for selecting desired interface components. The important aspect of the interface is the collection of properties and actions that it represents. As such, the interface is reduced into a form that does not include unnecessary abstraction as

\[
I = \{ (\hat{p}^1, A^1), (p^2, A^2), (p^6, A^6), (p^9, A^9), (p^{10}, A^{10}), (p^{12}, A^{12}), (p^{13}, A^{13}) \} \tag{6.18}
\]

The next simplification to be addressed is the removal of redundancies in the interface. The two types of redundancy that occur in the interface are redeclaration of the same entity and declaration of a feature specific instance of a domain property. In the case of redeclaration, the redundancy is simply removed by performing the union between the multiple declarations. If two features include different instances of the same entity then the overall interface must include all of the possibilities. Four of the features used in the robotics example include a subset of the system state property and its actions. The declarations for features \( f_1, f_9, f_{11}, \) and \( f_{12} \) are given by

\[
\hat{p}^i_V = p^i_V - \{ v^i_7, v^i_8, v^i_{13}, v^i_{14}, v^i_{15}, v^i_{16} \}
\]

\[
\hat{p}^i_V = p^i_V - \{ v^i_7, v^i_8, v^i_{13}, v^i_{14}, v^i_{15}, v^i_{16} \}
\]

\[
\hat{p}^i_V = p^i_V - \{ v^i_7, v^i_8, v^i_{13}, v^i_{14}, v^i_{15}, v^i_{16} \}
\]

\[
\hat{p}^i_V = p^i_V - \{ v^i_7, v^i_8, v^i_{13}, v^i_{14}, v^i_{15}, v^i_{16} \}
\]

\[
\hat{p}^i_V = p^i_V - \{ v^i_7, v^i_8, v^i_{13}, v^i_{14}, v^i_{15}, v^i_{16} \}
\]

In this case, the union results in

\[
\hat{p}^i_V = p^i_V \tag{6.20}
\]

This example also represents the other type of redundancy that must be removed. \( \hat{p}^i_V \) represents a feature specific version of \( p^i_V \). Since features are no longer relevant, the relative definition of \( \hat{p}^i_V \) is also no longer relevant. Thus, the property is redefined by as the intersection (\( \cap \)) of the feature specific definition and the domain level definition by
This completes the reduction of the interface and the complete results can be seen in Appendix B.2. The total number of algebraic statements in the reduced interface is twenty-eight. This represents a 43% reduction from the original interface which included forty-nine statements.

6.1.2.4. Interface Optimization

Optimization is the process of applying criteria to an existing solution to make the effectiveness and efficiency as good as possible. The reduced interface is a valid interface that could be transformed into EBNF and used; however, it does not take into account the interface semantics that has been identified for the domain. Interface optimization converts the algebraic specification of the reduced interface back into the human readable, EBNF syntax and simultaneously applies the interface semantics to optimize the interface. Not all semantic restrictions can be enforced by the syntax and the others are left for runtime implementation. However, the interface is optimized as much as possible using the semantic specification.

The reason that optimization occurs in parallel to translation into EBNF is that the algebraic specification is based on set algebra and contains no structural information, like the order in which commands may be included. EBNF, however, contains constructs for this type of specification.

First, the properties are translated into EBNF. The property list is not affected by the interface semantics because it only deals with relationships between actions. As such, the properties are simply translated in their entirety. With the details left out because of space considerations, the result is

\[ p^1_v = \hat{p}^1_v \cap p^1_v \] (6.21)
Next, the actions are translated into EBNF. During this process, the semantic restrictions are applied. Each action is merged into the interface after searching the semantic specification for restrictions between it and already inserted actions. The first actions inserted are the system state set and get actions with their specifications. As the first actions inserted, they obviously have no conflicts with already inserted actions. The set action specification, however, is modified by the optimization process because the semantic specification implies that some arguments are not valid for the set action. As such, the specification is given by

```
robot_interface = [sys_set],[sys_get];
sys_set = "set::",sys_property = ("sys::",("inactive"|"idle"|"tool"|"val"));
sys_get = "get::sys";
```

Next, the joint position property actions are added. Because of a conflict between the joint position set action and the system state set action, the two must not be included in the same communication. As such, they are indicated as options in a list. This is given by

```
sys_property = "sys::",sys_t;
j_pos_property = "j_pos::("real_val,5*(',real_val'),"(\deg"|\"rad")

ee_pos_property = "ee_pos::[",vals,\("in"|"cm")",',

((\fixed\ euler\ ).\("deg"|\"rad")|\mat\]);

tool_st_property = "tool_pow::",(\grip\ |\drill\ |\paint\_gun\ |\saw\);

tool_property = "tool::",("none"|"gripper"|"drill"|"paint\_gun"|"saw");

coll_dist_property = "coll_dist::.pos\_real_val,\("in"|"cm")

ob_ft_property = "ob	::[",vals,\"N",\fixed,\"N-m\];
```
This process of translating each action while simultaneously applying semantic rules is carried on throughout the interface specification. Ultimately, the interface is given by

\[
\text{robot\_interface} = [\text{sys\_set}\ |\ j\_pos\_set], [\text{sys\_get}], [j\_pos\_get];
\]

\[j\_pos\_set = \text{"set::", } j\_pos\_property;\]  
\[j\_pos\_get = \text{"get::j\_pos";}\]

This demonstrates the relationships between actions. Some of these properties also have restrictions on the arguments that can be used. For the complete optimized interface specification, see Appendix B.3. With interface optimization complete, the interface specification can be translated into desired representations. Before optimization, the twelve actions provided \(2^{12} (4096)\) possible action combinations. After optimization, only 1152 possible action combination remain meaning that 2944 invalid combinations have been removed.

### 6.1.3. Interface Translation

Interface translation is a highly structured process of converting the optimized EBNF interface specification into desired representations. Two approaches are explored in this report. They are a sequential parsing approach and a hierarchical approach. In each case, the specification of the translation would be given by a domain expert as a lookup table. When a particular pattern is found that matches an entry in the lookup table, its translation is placed into the interface representation. These methods are discussed briefly in this section and extensively in Section 5.2.

The sequential method simply searches the interface specification for defined patterns. The patterns are defined as ordered sets of EBNF special symbols and
interface terminals. For instance, the complex pattern used to specify a real number is given by

\[
(\text{'+','-','}{0}'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'});\]

\[
(\text{'.'},\{{}'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9}'\})\]

(6.26)

Whenever this pattern is found in the interface specification, it would be replaced in the interface representation by an equivalent representation. In C++, this would likely just be the float keyword. Similarly, other ordered symbols could be specified to identify interface specific components and their representations as in Table 6.2.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>C++ Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;set::j_pos::(&quot;</td>
<td>void set_j_pos(</td>
</tr>
<tr>
<td>', '</td>
<td>,</td>
</tr>
<tr>
<td>&quot;)',(&quot;deg</td>
<td>&quot;rad&quot;)</td>
</tr>
<tr>
<td></td>
<td>,angle);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Representation of j_pos_set Components in C++

From these specifications, the parser could recognize the EBNF specification of the joint position set action and produce the C++ representation given in Table 6.3.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>Interface</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;set::j_pos::(&quot;real_val,5*{', real_val},')',(&quot;deg</td>
<td>&quot;rad&quot;)</td>
<td>enum angle{deg,rad};</td>
</tr>
<tr>
<td></td>
<td>void set_j_pos(float,float,float,float,float,float,angle);</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Representation of j_pos_set in C++

Clearly, this discussion of sequential parsing has not presented enough detail to actually generate the representation in Table 6.3. Some type of structure would have to be given to specify that the enum must come before the function. Also, it seems that the number of patterns necessary to describe the translation could become overwhelming, particularly when the translation table is developed for all possible combinations from the interface domain specification. Lastly, it is likely that conflicts could occur that require more specification. For instance, a comma may have
different representations based on its context. All of this is beyond the scope of this report as sequential parsing is only presented as an unexplored concept. With this said, Appendix C.1 includes a possible C++ interface for the product and an example of its usage.

The other approach presented in this report is hierarchical. Again, the hierarchical approach has not been evaluated, but it appears to provide some promising advantages. EBNF special symbols have particular meanings that can be interpreted in a hierarchical manner. For instance, anything included within square brackets is optional regardless of the complexity of the contents. Similarly, other symbols have particular meanings and are evaluated with a particular precedence regardless of the specific data. This fact likely will lead to a smaller number of table entries in the translation table. Also, translations might be reusable because the specific data is not important at higher levels of the hierarchy. Lastly, the hierarchical approach automatically contains some of the structural information that might be necessary for representations.

All of these details are beyond the scope of this report, but a brief discussion of a possible approach is given for completeness. The examples are in XML because it adequately demonstrates some of the structural requirements. First, the interface as a whole may require some initialization as in Table 6.4.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx;</td>
<td><code>&lt;?xml version=&quot;1.0&quot; encoding=&quot;utf-8&quot; ?&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;xs:schema targetNamespace=&quot;robot&quot;</code></td>
</tr>
<tr>
<td></td>
<td><code>    elementFormDefault=&quot;qualified&quot; xmlns=&quot;robot&quot;</code></td>
</tr>
<tr>
<td></td>
<td><code>    xmlns:xs=&quot;http://www.w3.org/2001/XMLSchema&quot;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;xs:element name=&quot;robot&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>    xxx</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/xs:element&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/xs:schema&gt;</code></td>
</tr>
</tbody>
</table>

Table 6.4: Representation of Basic Interface in XML Schema
Because of the hierarchical nature, the notations xxx, yyy, ..., etc. are used to identify unidentified components meaning that the translation is the same regardless of what data is given where these notations appear. Additional translations could be defined in the same manner. A few of these are given in Table 6.5. For explanations of these representations, see Section 5.2.2. Also, a possible XML representation for the entire interface is given in Appendix C.2.

<table>
<thead>
<tr>
<th>EBNF</th>
<th>XML Schema Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx, yyy,...</td>
<td><code>&lt;xs:complexType&gt;</code>&lt;br&gt;<code>&lt;xs:all&gt;</code>&lt;br&gt;<code>xxx</code>&lt;br&gt;<code>yyy</code>&lt;br&gt;...&lt;br&gt;<code>&lt;/xs:all&gt;</code>&lt;br&gt;<code>&lt;/xs:complexType&gt;</code></td>
</tr>
<tr>
<td>[xxx]</td>
<td><code>minOccurs=&quot;0&quot;</code>&lt;br&gt;<code>&lt;/xs:complexType&gt;</code></td>
</tr>
<tr>
<td>xxx</td>
<td>yyy</td>
</tr>
<tr>
<td>&quot;xxx::&quot;</td>
<td><code>&lt;xs:element name=&quot;xxx&quot;&gt;</code>&lt;br&gt;<code>&lt;/xs:element&gt;</code></td>
</tr>
<tr>
<td>(&quot;xxx&quot;</td>
<td>yyy</td>
</tr>
</tbody>
</table>

Table 6.5: Representations of Various EBNF Components in XML Schema

All of the interface examples given in this report were assembled manually, but wherever possible, the specifications given were adhered to in a realistic way. The result, therefore, is not necessarily well written C++ or XML but it is functional for the purposes of this report.
6.2. Future Work

This report has defined a new approach to software interface development. The steps were defined as domain analysis, product-line analysis, interface generation, and interface translation. Each of these is a complex concept that will likely require devoted research to fully understand and define. This report outlines the analysis and discusses each of these topics with enough detail to explain the concepts. Also, the analysis steps are thoroughly investigated as far as the results required to enable interface generation and translation. However, because of the scope of the interface generation problem, all of the steps require significant future work for the interface synthesis process to be implemented. This section presents concepts for future work on the approach itself and the individual steps.

6.2.1. Approach

Fundamentally, the process of analysis, generation, and then translation is sound. Analysis, however, is currently addressed as two separate analysis processes of domain analysis and product-line analysis. This decision was based on the possibility that certain aspects of a domain might be specified globally from which product-lines could be defined. This could provide reusability of the interface specifications within a domain. This report has not, however, successfully presented this argument because the examples given do not take advantage of this separation of the analysis. While it is likely that keeping these two analyses separate is beneficial to the process, it is up to future work to make a good argument for this and provide examples.

Additionally, as interface translation becomes better understood, it might become useful to add a translation analysis to the analysis performed by the domain expert at the beginning of the process. Translation analysis would ideally cover all possible combinations of interface components and thus would be performed at component specification instead of after the interface has been generated. In this
report, translation analysis was performed after the interface was generated. Moving
the step into the analysis performed early in the generation approach would make it
more difficult because all combinations must be addressed, but it would aid in
reusability and extensibility of the specification because translation would not be
dependant on the generated interface. It also might allow some reusability between
unrelated domains if logical and general translations exist for the various EBNF
components without consideration for the details. Based on the hierarchical approach
to interface translation that was presented in this report, this is a distinct possibility
that should be evaluated.

Research on the approach will be necessary in the future to evaluate these
possibilities as well as to discover new concepts. However, the majority of future
work will address the individual steps in the process. The following sections discuss
possible research topics related to the individual steps. Since this report makes little
distinction between domain analysis and product-line analysis, these steps will be
addressed together in the next section.

6.2.2. Analysis

This report focuses on the analysis steps of the interface synthesis process. Specifically an Extended Backus-Naur Form (EBNF) and an algebraic form have been created for specifying the properties, actions, and features that make up a domain. Similarly, a syntax for defining semantic relationships has been created. These things characterize the way that an interface should be specified so that interface generation can be performed, however, this report has not attempted to present a proper analysis approach.

This report took the approach of identifying features first and then properties
and actions required to define the features. From the properties and actions, features
are redefined formally. It was known from the beginning that this is not the optimal
approach; however, it is important to fully understand the desired results of analysis
before a proper approach can be given. This report has defined the desired results of the analysis and it is up to future work to research the optimal approach for discovering properties, actions, and features within a system.

Also, the effectiveness of the specification must be evaluated with respect to changes in the domain. Currently, features are defined in terms of domain properties and actions where a feature property may be some subset of the domain property. With this type of specification, a change in the domain property requires each feature that includes the property to be reevaluated. It is feasible that a better way of specifying the components exists that would eliminate this concern. More likely, however, it will be determined that this is an acceptable condition of this approach because the process is intended for well-understood domains. In general, this means that the domain is relatively stable. As such, it is not unreasonable to expect that interfaces for products within a domain must be reassessed if the domain changes.

Some of these details would become clearer with additional examples of interface specification. The examples used in this report do not stress the approach taken for interface specification because the alarm clock example was intentional kept simple for explanation purposes and the approach was developed with the robotics domain in mind so it is well suited for robotics domain specification. The examples in this report, therefore, do not necessarily demonstrate that the approach is applicable to a wide variety of domains. Future work should attempt to develop interface specifications of product-lines in diverse domains to assess the robustness of the approach presented in this report.

Experience from this research implies that the defined approach is extremely effective for domains where the properties can be defined as finite state machines. These systems have well structured input and output. However, it may be difficult or impossible to apply the property definition to less structured interface systems. For instance, a common software interface system is a database, but this approach may not be applicable to database management software. Functions like sorting and
searching within a list do not have readily apparent representations as states and may be difficult to implement as actions on a property. This is not to say that the approach presented is fundamentally flawed. It may be found, though, that it needs to be extended for some systems or the type of system to which it will be applicable should be defined.

Extending this work to other domains will also help to define the effectiveness of the syntax defined for interface specification. For instance, one flaw discovered in the syntax specification is the requirement that the \textit{set} action arguments be defined as a subset of the property values. There is good reason for this requirement in that it structures the interface consistently; however, in the robotics example, pseudo-states were required for various actions like returning from an error state. These pseudo-states had to be added to the property definition but they will never be returned in response to a \textit{get} action so the property values are somewhat misleading. Future work should evaluate this and similar concerns and determine whether changes should be made to the interface specification syntax.

Another concern that may arise is the specification of the interface as a passive system. The interface specification is defined such that a \textit{get} action is required to receive any information about the state of the system. This might be a problem in event driven systems. For instance, within the robotics domain, the system can enter an error state with no input from the user. The user would be unaware of the error state without a call to the system state \textit{get} action. Practically, in the robotics domain, this is only a small problem because robot safety is maintained at the controller level so the user interface could simply poll the system at a fixed rate to determine the status. In other domains, this may be unacceptable, so future research should attempt to identify a specification for regular and event-based feedback from the system. This does not appear to present significant challenges but this report has not presented a solution.
Variables and units, also, present significant complications in the specification of interfaces. Particularly in robotics, representation of variables and units is vital to the success of an interface. The approach taken in this report requires the domain expert to provide an EBNF representation of the variables and units required. While this is not an impossible task, their may be an approach that improves the specification. Common programming languages like C++ have a set of variables from which some type of standard could be defined for interface specification. Similarly, XML defines many variables including lists, which could be used for vectors. It would be wise for future researchers to entertain the possibility of defining some type of variable specification standard for the EBNF representation of interfaces to simplify this part of the specification. Similarly, units might be standardized. It is less difficult to define the units for a variable in EBNF so it might not be necessary to define such a standard, but it should be evaluated in future work.

A syntax for semantic relationship specification was given in this report to support interface optimization and to provide some runtime information about the interface. The process for identifying these relationships was not thoroughly explored in this report beyond describing several types of semantic relationship. From the descriptions, a domain expert might be able to easily identify relationships, but it is likely that a more formal approach could be developed.

Three types of semantic relationship were defined, but these definitions will probably require extension. Particularly, temporal behavior has only been addressed as the effect of a time component on certain relationships among properties. There are certainly many other types of temporal behavior that may have profound effects on system interfaces. Within the robotics domain, several system states are entered by a user command but are not exited until some action is completed. It might be possible to analyze the temporal behavior of these actions to provide information as to time requirements. For instance, homing might have a fixed time requirement and other actions like joint moves may have times that can be calculated. Future work should
assess the feasibility of this type of analysis and the likelihood that this information will prove useful to the interface. Similarly, other types of semantic relationships might be discovered. Also, the semantic relationship specification syntax should be evaluated. It has been laid out in a general way and should be able to support extension of interface semantics concepts but flaws may be found.

6.2.3. Interface Generation

Interface generation is a complex concept that has been introduced in this report. Chapter 5 presents a conceptual approach to interface generation with each step performed manually. The process includes feature selection, feature interface addition, interface reduction, and interface optimization. None of these steps has been researched thoroughly or been acceptably defined. Future work will be required to investigate the approach presented in this report and to implement the approach in a software suite.

It is not reasonable to present all of the concerns that future work must address for interface generation because it is a purely conceptual design. Future work will be required to evaluate every aspect of the approach to determine if it is logical and can be successfully implemented in software. The interface generation software will need to be completely general so that the process can be performed on products in any domain. This section will briefly identify the process and highlight some difficulties. Chapter 5 contains a detailed discussion of the conceptual design, assumptions, and challenges.

Assuming that the EBNF specification and its algebraic equivalent have been thoroughly investigated to the point that the generator can be implemented, the software must import an EBNF specification of a domain interface and generate the algebraic specification. This is conceptually fairly simple but will likely be relatively difficult in actual implementation because of the many unknowns that exist in the specification.
Once the domain is known to the generator, feature selection occurs. Feature selection is a relatively simple step where the features for a particular product are selected from the list of features defined for the domain. This could be imported to the generator in EBNF, but it is logical to develop a Graphical User Interface (GUI) for this step. A conceptual GUI was described in Chapter 5 that analyses the domain specification and presents a detailed list of features from which to select. Future work should develop this concept into a working application.

Interface addition is then performed algebraically. On paper, this is a relatively simple step analogous to cutting and pasting the individual feature specifications onto a single sheet. However, in software, complex data structures will probably be required to organize the specifications into a single interface. Similarly, interface reduction is, on paper, the simple process of removing redundancies that result from interface addition. The equivalent software algorithm will depend on the data structures chosen and will likely be somewhat difficult to specify.

Lastly, interface optimization is addressed. This report has presented some concepts for interface optimization but it is believed that this step will require significant future research. This report presents optimization as the process of applying semantic relationship specifications to the reduced interface in order to produce an efficient and effective interface specification. Again, it is likely that implementing this process in software will present significant difficulties. Also, the optimization approach given might not be sufficient. The additional temporal relationships that might be identified for an interface will likely provide less straightforward information that must be used in interface optimization. Also, it is feasible that other domains will not provide such manageable specifications. Because, interface optimization is not well understood, it will require additional research to develop an understanding before application can be investigated further.
6.2.4. **Interface Translation**

Interface translation is similar to interface optimization in the level of understanding that currently exists. This report has presented two possible approaches, but they are only initial guesses and the problem itself must be better understood before implementation can be explored. This report discusses interface translation with respect to a fully specified interface for a particular robotics product. While interface translation occurs after the product interface has been specified, the analysis must occur earlier and thus presents a much more difficult problem. Future research will need to identify a process for interface translation analysis that will result in successful translation at the end of interface generation. It is believed that EBNF is sufficiently structured to allow a well-defined approach to be developed for interface translation analysis. At this point, however, it is purely conceptual. See Section 5.2 for the conceptual approaches presented in this report.

6.3. **Conclusion**

This report has presented an approach to interface synthesis for products within well-understood domains. The primary accomplishment is formalization of an approach and the interface syntax and semantics that are required for interface generation. Hopefully, this research and the results presented will spark a new research thread into interface synthesis. While every effort has been made to produce an approach and specification that can act as a basis, it is likely that future research will be required to refine the results. However, though the results may not be flawless, this report has demonstrated the feasibility of interface generation and translation. For success, the approach must be based on a systematic approach and an algebraic representation of the components as demonstrated in this report.
Appendix A

A. Robotics Domain Specification

This appendix contains the full specification of the robotics domain. Each component is included in its entirety without any simplification or reference to previously defined non-terminals. The first listing is in the EBNF domain model specification given by the domain expert (in the specification order of Chapter 4). The second is the algebraic representation of the domain model that could ultimately be extracted by the system. Third is the EBNF feature specification and fourth is the algebraic representation of the feature specification as might be extracted from the EBNF specification. Lastly, the semantic restrictions for the domain are given in the EBNF syntax and then the algebraic.

A.1. EBNF Domain Model Specification

```
sys_property = "sys::", sys_t;
sys_t = init_t | end_t | move_t | hold_t | err_t;
init_t = "inactive" | "enabled" | "idle";
end_t = "shutdown" | "estop";
move_t = "homing" | "tool" | "val";
hold_t = "hold" | "cancel" | "resume";
err_t = "error", (("S" | "L" | "E" | "T" | "C"), digit | "clear");
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
```
sys_actions = [sys_set], [sys_get];
sys_set = "set::", sys_property;
sys_get = "get::sys";

j_pos_property = "j_pos::(", real_val, 5*(", real_val), "), ("deg" | "rad");
real_val = ['+', '-', ''], {digit}, ['.', {digit}] =;
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

j_pos_actions = [j_pos_set], [j_pos_get];
j_pos_set = "set::", j_pos_property;
j_pos_get = "get::j_pos";

j_vel_property = "j_vel::(", real_val, 5*(", real_val), "), ("deg/s" | "rad/s");
real_val = ['+', '-', ''], {digit}, ['.', {digit}] =;
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

j_vel_actions = [j_vel_set], [j_vel_get];
j_vel_set = "set::", j_vel_property;
j_vel_get = "get::j_vel";

j_torq_property = "j_torq::(" real_val, 5*(", real_val), "), "N-m";
real_val = ['+', '-', ''], {digit}, ['.', {digit}] =;
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

j_torq_actions = [j_torq_set], [j_torq_get];
j_torq_set = "set::", j_torq_property;
j_torq_get = "get::j_torq";
\( j_{\text{curr}} \text{ property} = "j_{\text{curr}}::(\"real\_val,5*(\',real\_val),\")","amp"; \)
\( \text{real\_val} = [\'+','-'],\{\text{digit}\},[',\{\text{digit}\}]\);  
\( \text{digit} = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'; \)

\( j_{\text{curr}} \text{ actions} = [j_{\text{curr}} \text{ set}],[j_{\text{curr}} \text{ get}]; \)
\( j_{\text{curr}} \text{ set} = \"set::\",j_{\text{curr}} \text{ property}; \)
\( j_{\text{curr}} \text{ get} = \"get::j_{\text{curr}}\"; \)

\( \text{ee\_pos \_property} = "\text{ee\_pos}::[\"vals,(\"in\"|\"cm\")\",\", \)
\( \quad ((\text{fixed} | \text{euler}),(\"deg\"|\"rad\")|\text{mat}),\]); \)
\( \text{vals} = (\',\text{real\_val},2*(\',\text{real\_val})); \)
\( \text{real\_val} = [\'+','-'],\{\text{digit}\},[',\{\text{digit}\}]\);  
\( \text{digit} = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'; \)
\( \text{fixed} = \"fixed:\",\text{vals}; \)
\( \text{euler} = \"euler:\",\text{vals}; \)
\( \text{mat} = \"mat::(\",\text{real\_val},8*(\',\text{real\_val})); \)

\( \text{ee\_pos \_actions} = [\text{ee\_pos \_set}],[\text{ee\_pos \_get}]; \)
\( \text{ee\_pos \_set} = \"set::\",\text{ee\_pos \_property}; \)
\( \text{ee\_pos \_get} = \"get::ee\_pos\"; \)

\( \text{ee\_vel \_property} = "\text{ee\_vel}::[\"vals,(\"in/s\"|\"cm/s\")\",\", \)
\( \quad (\text{fixed} | \text{euler}),\{"deg/s\"|\"rad/s\")|\text{mat}),(\]); \)
\( \text{vals} = (\',\text{real\_val},2*(\',\text{real\_val})); \)
\( \text{real\_val} = [\'+','-'],\{\text{digit}\},[',\{\text{digit}\}]\);  
\( \text{digit} = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'; \)
\( \text{fixed} = \"fixed:\",\text{vals}; \)

\( \text{ee\_vel \_actions} = [\text{ee\_vel \_set}],[\text{ee\_vel \_get}]; \)
\( \text{ee\_vel \_set} = \"set::\",\text{ee\_vel \_property}; \)
\( \text{ee\_vel \_get} = \"get::ee\_vel\"; \)
ee_ft_property = "ee_ft::["vals","N","fixed","N-m"];
vals = ('real_val',2*(',',real_val))';
real_val = ['+',']',{digit}, ['+',']',{digit}]};
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
fixed = "fixed:";vals;

ee_ft_actions = [ee_ft_set], [ee_ft_get];
ee_ft_set = "set::", ee_ft_property;
ee_ft_get = "get::ee_ft";

tool_st_property = "tool_st::", (grip | drill | paint-gun | saw);
grip = "open" | "closed" | "opening" | "closing";
drill = "fwd" | "rev" | "off";
paint_gun = "on" | "off";
saw = (['0']|['',']',{digit}])|'1'|['',']',{0}]|"off";
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

tool_st_actions = [tool_st_set], [tool_st_get];
tool_st_set = "set::", tool_st_property;
 tool_st_get = "get::tool_st";

tool_property = "tool::", ("none" | "gripper" | "drill" | "paint_gun" | "saw");

tool_actions = [tool_set], [tool_get];
tool_set = "set::", tool_property;
tool_get = "get::tool";

154
A.2. Algebraic Domain Model Specification

\[ P = \{ \text{sys, j_pos, j_vel, j_torq, j_curr, ee_pos, ee_vel,} \}
\]

\[ P_f = \{ \text{inactive, enabled, idle, shutdown, estop, homing, tool, val, hold, cancel, resume,} \}
\]

\[ P_f = \{ \text{error_S#, error_L#, error_E#, error_T#, error_C#, error_clear} \}
\]
\[ A^1 = \{ \text{set, get} \} \]
\[ a_{\text{set}}^1 = p^1_v \]
\[ a_{\text{get}}^1 = \emptyset \]

\[ p^2_v = (R, R, R, R, R, R) \]

\[ A^2 = \{ \text{set, get} \} \]
\[ a_{\text{set}}^2 = p^2_v \]
\[ a_{\text{get}}^2 = \emptyset \]

\[ p^3_v = (R, R, R, R, R, R) \]

\[ A^3 = \{ \text{set, get} \} \]
\[ a_{\text{set}}^3 = p^3_v \]
\[ a_{\text{get}}^3 = \emptyset \]

\[ p^4_v = (R, R, R, R, R, R) \]

\[ A^4 = \{ \text{set, get} \} \]
\[ a_{\text{set}}^4 = p^4_v \]
\[ a_{\text{get}}^4 = \emptyset \]

\[ p^5_v = (R, R, R, R, R, R) \]

\[ A^5 = \{ \text{set, get} \} \]
\[ a_{\text{set}}^5 = p^5_v \]
\[ a_{\text{get}}^5 = \emptyset \]

$A^6 = \{set, get\}$
$a_set^6 = p_6^v$
$a_get^6 = \emptyset$

$p_7^v = \{(R,R,R),(R,R,R)\}$

$A^7 = \{set, get\}$
$a_set^7 = p_7^v$
$a_get^7 = \emptyset$

$p_8^v = \{(R,R,R),(R,R,R)\}$

$A^8 = \{set, get\}$
$a_set^8 = p_8^v$
$a_get^8 = \emptyset$

$p_9^v = \{off, on, fwd, rev, open, closed, opening, closing, 0−1\}$

$A^9 = \{set, get\}$
$a_set^9 = p_9^v$
$a_get^9 = \emptyset$

$p_{10}^v = \{none, gripper, drill, paint _gun, saw\}$

$A^{10} = \{set, get\}$
$a_set^{10} = p_{10}^v$
$a_get^{10} = \emptyset$

157
\[ p_{i1}^{11} = \left[ (R, R, R), (R, R, R) \right] \]

\[ A^{i1} = \{ \text{get} \} \]
\[ a_{\text{get}}^{i1} = \emptyset \]

\[ p_{i2}^{12} = \mathbb{R} \]

\[ A^{i2} = \{ \text{get} \} \]
\[ a_{\text{get}}^{i2} = \emptyset \]

\[ p_{i3}^{13} = \left[ (R, R, R), (R, R, R) \right] \]

\[ A^{i3} = \{ \text{get} \} \]
\[ a_{\text{get}}^{i3} = \emptyset \]

### A.3. EBNF Feature Specification

val_move_feature = sys_actions;
move_t = "homing" | "val";
err_t = "error_", ('S', digit | "clear");

j_pos_feature = j_pos_actions, val_move_feature;

j_vel_feature = j_vel_actions, val_move_feature;

j_torq_feature = j_torq_actions, val_move_feature;

j_curr_feature = j_curr_actions, val_move_feature;

ee_pos_feature = ee_pos_actions, val_move_feature;
ee_vel_feature = ee_vel_actions, val_move_feature;

ee_ft_feature = ee_ft_actions, val_move_feature;

error_check_feature = sys_actions;
movet_t = "homing";
err_t = "error", ('S'|'L'|'E'), digit |"clear";

grip_feature = tool_st_actions;
tool_st_property = "tool_st::", grip;

tool_feature = tool_st_actions, tool_actions, sys_actions;
movet_t = "homing"|"tool";
err_t = "error", ('S','T'), digit |"clear";

coll_detect_feature = coll_dist_actions, ob_ft_actions, sys_actions;
movet_t = "homing";
err_t = "error", ('S','C'), digit |"clear";

A.4. Algebraic Feature Specification

\[
F = \begin{cases} 
val_move, j_pos, j_vel, j_torq, j.curr, ee_pos, \\
\{ee_vel, ee_ft, error_check, grip, tool, coll_detect\} 
\end{cases}
\]

\[
f_1 = (\hat{p}^1, A^1)
\]

\[
\hat{p}_v^1 = p_v^1 - \{v_7^1, v_1^3, v_1^4, v_1^5, v_1^6\}
\]

\[
f_2 = (p^2, A^2) \cup f_1
\]

\[
f_3 = (p^3, A^3) \cup f_1
\]
\[ f_4 = (p^4, A^4) \cup f_1 \]

\[ f_5 = (p^5, A^5) \cup f_1 \]

\[ f_6 = (p^6, A^6) \cup f_1 \]

\[ f_7 = (\dot{p}^7, A^7) \cup f_1 \]

\[ f_8 = (\dot{p}^8, A^8) \cup f_1 \]

\[ f_9 = (\dot{p}^9, A^9) \]

\[ \dot{p}_v^i = p_v^i - \{v_7, v_8, v_{15}, v_{16}\} \]

\[ f_{10} = (\dot{p}^9, A^9) \]

\[ \dot{p}_v^g = p_v^g - \{v_1, v_2, v_3, v_4, v_5\} \]

\[ f_{11} = (p^9, A^9) \cup (p^{10}, A^{10}) \cup (\dot{p}^7, A^7) \]

\[ \dot{p}_v^j = p_v^j - \{v_8, v_{13}, v_{14}, v_{16}\} \]

\[ f_{12} = (p^{12}, A^{12}) \cup (p^{13}, A^{13}) \cup (\dot{p}^1, A^1) \]

\[ \dot{p}_v^j = p_v^j - \{v_7, v_8, v_{13}, v_{14}, v_{15}\} \]

### A.5. EBNF Semantic Restrictions

- set : ee pos – set : j_pos;
- set : ee vel – set : j_vel;
- set : ee ft – set : j_torq;
- set : j_curr – set : j_torq;
set :: j_vel - set :: j_pos;
set :: j_torq - set :: j_vel;
set :: j_torq - set :: j pos;
set :: ee_vel - set :: ee_pos;
set :: ee_pos - set :: ee_vel;
set :: ee_pos - set :: ee_pos;

(set :: j_curr | set :: ee_vel | set :: ee_vel) - set :: j_pos;
(set :: j_curr | set :: ee_pos | set :: ee_vel) - set :: j_vel;
(set :: ee_pos | set :: ee_vel) - set :: j_torq;
(set :: ee_pos | set :: ee_vel | set :: ee_pos) - set :: j_curr;

set :: sys :: enabled × (sys :: inactive | sys :: estop) → sys :: enabled;
set :: sys :: enabled × sys :: shutdown → sys :: idle;
set :: sys :: shutdown × (sys :: enabled | sys :: idle) → sys :: shutdown;
set :: sys :: estop × → sys :: estop;
set :: sys :: homing × sys :: enabled → sys :: homing;
set :: sys :: hold × (sys :: homing | sys :: tool | sys :: val) → sys :: hold;
set :: sys :: cancel × (sys :: hold | sys :: homing | sys :: tool | sys :: val) →
(sys :: enabled | sys :: idle);
set :: sys :: resume × sys :: hold → (sys :: homing | sys :: tool | sys :: val);
set :: sys :: error _## × sys :: estop → sys :: error _##;
set :: sys :: error _clear × sys :: error _## → ¬ sys :: estop;

set :: (j_pos | j_vel | j_torq | j_curr) × sys :: idle → sys :: val;
set :: (ee_pos | ee_vel | ee_pos) × sys :: idle → sys :: val;
set :: tool × sys :: idle → sys :: tool;
set :: tool_st::off × tool::drill & (tool_st::fwd | tool_st::rev) → tool_st::off;
set :: tool_st::off × tool::paint_gun & tool_st::on → tool_st::off;
set :: tool_st::off × tool::saw & tool_st::### → tool_st::off;
set :: tool_st::on × tool::paint_gun & tool_st::off → tool_st::on;
set :: tool_st::fwd × tool::drill & tool_st::off → tool_st::fwd;
set :: tool_st::rev × tool::drill & tool_st::off → tool_st::rev;
set :: tool_st::open × tool::gripper & (tool_st::closed | tool_st::closing) → tool_st::opening;
set :: tool_st::closed × tool::gripper & (tool_st::open | tool_st::opening) → tool_st::closing;
set :: tool_st::### × tool::saw & tool_st::off → tool_st::###;

A.6. Algebraic Semantic Restrictions

\[ a_{set}^6 = a_{set}^2; \]
\[ a_{set}^7 = a_{set}^3; \]
\[ a_{set}^8 = a_{set}^4; \]
\[ a_{set}^9 = a_{set}^4; \]
\[ a_{set}^3 - a_{set}^2; \]
\[ a_{set}^4 - a_{set}^3; \]
\[ a_{set}^5 - a_{set}^7; \]
\[ a_{set}^5 - a_{set}^6; \]
\[ a_{set}^5 - a_{set}^6; \]
\[ (a_{set}^5 | a_{set}^7 | a_{set}^8) - a_{set}^2; \]
\[ (a_{set}^5 | a_{set}^6 | a_{set}^8) - a_{set}^3; \]
\[ (a_{set}^6 | a_{set}^7) - a_{set}^4; \]
\[ (a_{set}^6 | a_{set}^7 | a_{set}^8) - a_{set}^5; \]
\begin{align*}
a_{\text{set}}^1_{12} & \times (v^1_i \mid v^5_i) \rightarrow v^1_2; \\
a_{\text{set}}^1_{12} & \times v^1_4 \rightarrow v^1_3; \\
a_{\text{set}}^1_{13} & \times (v^1_2 \mid v^5_3) \rightarrow v^1_4; \\
a_{\text{set}}^1_{14} & \times v^1_5 \rightarrow v^1_6; \\
a_{\text{set}}^1_{15} & \times v^1_6 \rightarrow v^1_7; \\
a_{\text{set}}^1_{16} & \times v^1_7 \rightarrow v^1_8; \\
(a_{\text{set}}^1_{12} \mid a_{\text{set}}^1_{13} \mid a_{\text{set}}^1_{14} \mid a_{\text{set}}^1_{15} \mid a_{\text{set}}^1_{16}) & \times \neg v^1_5 \rightarrow (v^1_{12} \mid v^1_{13} \mid v^1_{14} \mid v^1_{15} \mid v^1_{16}); \\
a_{\text{set}}^1_{17} & \times (v^1_{12} \mid v^1_{13} \mid v^1_{14} \mid v^1_{15} \mid v^1_{16}) \rightarrow \neg v^1_5; \\
(a_{\text{set}}^2 \mid a_{\text{set}}^3 \mid a_{\text{set}}^4 \mid a_{\text{set}}^5) & \times v^1_3 \rightarrow v^1_8; \\
(a_{\text{set}}^6 \mid a_{\text{set}}^7 \mid a_{\text{set}}^8) & \times v^1_3 \rightarrow v^1_8; \\
a_{\text{set}}^{10} & \times v^1_4 \rightarrow v^1_7; \\
(a_{\text{set}}^9_1 \mid v^1_3 \mid v^9_4) & \rightarrow v^1_9; \\
a_{\text{set}}^9_1 & \times v^1_4 \& v^9_2 \rightarrow v^1_9; \\
a_{\text{set}}^9_1 & \times v^1_5 \& v^9_5 \rightarrow v^1_9; \\
a_{\text{set}}^9_2 & \times v^1_4 \& v^9_5 \rightarrow v^1_9; \\
a_{\text{set}}^9_3 & \times v^1_3 \& v^9_1 \rightarrow v^1_9; \\
a_{\text{set}}^9_4 & \times v^1_3 \& v^9_1 \rightarrow v^1_9; \\
a_{\text{set}}^9_5 & \times v^1_6 \& v^9_8 \rightarrow v^1_9; \\
a_{\text{set}}^9_6 & \times v^1_2 \& v^9_1 \rightarrow v^1_9; \\
a_{\text{set}}^9_7 & \times v^1_5 \& v^9_1 \rightarrow v^1_9; \\
a_{\text{set}}^9_8 & \times v^1_5 \& v^9_1 \rightarrow v^1_9;
\end{align*}
Appendix B

B. Robot Interface

This appendix contains the full specification of the robot interface containing the joint position, end-effector position, error checking, tool control, and collision detection features. The interface specification includes various steps of the process.

B.1. Feature Addition

\[ I = \{ f_2, f_6, f_9, f_{11}, f_{12} \} \]

\[ f_2 = (p_2^2, A^2) \cup f_1 \]

\[ p_2^2 = (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) \]

\[ A^2 = \{ \text{set, get} \} \]

\[ a\_set_2 = p_2^2 \]

\[ a\_get_2 = \emptyset \]

\[ f_1 = (\hat{p}_1, A^1) \]

\[ \hat{p}_1 = p_1^1 \setminus \{ v_{13}', v_{14}', v_{15}', v_{16}' \} \]

\[ p_1^1 = \{ \text{inactive, enabled, idle, shutdown, estop, homing, tool, val, hold, cancel, resume,} \]

\[ \text{error\_S\#, error\_L\#, error\_E\#, error\_T\#, error\_C\#, error\_clear} \}

\[ A^1 = \{ \text{set, get} \} \]

\[ a\_set_1 = \hat{p}_1 \]

\[ a\_get_1 = \emptyset \]
\[ f_6 = \left( p^6, A^6 \right) \cup f_1 \]
\[
p^6_r = \left\{ \left( R, R, R \right), \left( R, R, R \right), \left( R, R, R, R, R, R, R, R \right) \right\}
\]
\[ A^6 = \{ \text{set, get} \} \]
\[ a_{set}^6 = p^6_r \]
\[ a_{get}^6 = \emptyset \]

\[ f_9 = \left( \hat{p}^i, A^i \right) \]
\[
\hat{p}^i_r = p^i_r \setminus \{ v^i_7, v^i_8, v^i_15, v^i_16 \}
\]
\[ p^i_r = \{ \text{inactive, enabled, idle, shutdown, estop, homing, tool, val, hold, cancel, resume,} \}
\]
\[ A^i = \{ \text{set, get} \} \]
\[ a_{set}^i = \hat{p}^i_r \]
\[ a_{get}^i = \emptyset \]

\[ f_{11} = \left( p^9, A^9 \right) \cup \left( p^{10}, A^{10} \right) \cup \left( \hat{p}^i, A^i \right) \]
\[ p^9_r = \{ \text{off, on, fwd, rev, open, closed, opening, closing, 0–1} \} \]
\[ A^9 = \{ \text{set, get} \} \]
\[ a_{set}^9 = p^9_r \]
\[ a_{get}^9 = \emptyset \]
\[ p^{10}_r = \{ \text{none, gripper, drill, paint gun, saw} \} \]
\[ A^{10} = \{ \text{set, get} \} \]
\[ a_{set}^{10} = p^{10}_r \]
\[ a_{get}^{10} = \emptyset \]
\[ \hat{p}^i_r = p^i_r \setminus \{ v^i_8, v^i_{13}, v^i_{14}, v^i_{16} \} \]
\[ p^i_r = \{ \text{inactive, enabled, idle, shutdown, estop, homing, tool, val, hold, cancel, resume,} \}
\]
\[ A^i = \{ \text{set, get} \} \]
\[ a_{set}^i = \hat{p}^i_r \]
\[ a_{get}^i = \emptyset \]
\[ f_{12} = \left( p^{12}, A^{12} \right) \cup \left( p^{13}, A^{13} \right) \cup \left( \hat{p}^1, A^1 \right) \]

\[ p^{12}_v = \mathbb{R} \]

\[ A^{12} = \{ \text{get} \} \]

\[ a_{\text{get}}^2 = \emptyset \]

\[ p^{13}_v = \left[ (\mathbb{R}, \mathbb{R}, \mathbb{R}), (\mathbb{R}, \mathbb{R}, \mathbb{R}) \right] \]

\[ A^{13} = \{ \text{get} \} \]

\[ a_{\text{get}}^3 = \emptyset \]

\[ \hat{p}^1_v = p^1_v - \{ v^1_7, v^1_8, v^1_{13}, v^1_{14}, v^1_{15} \} \]

\[ p^1_v = \{ \text{inactive, enabled, idle, shutdown, estop, homing, tool, val, hold, cancel, resume} \} \]

\[ p^1_v = \{ \text{error } S\#, \text{error } L\#, \text{error } E\#, \text{error } T\#, \text{error } C\#, \text{error } \text{clear} \} \]

\[ A^1 = \{ \text{set, get} \} \]

\[ a_{\text{set}}^1 = \hat{p}^1_v \]

\[ a_{\text{get}}^1 = \emptyset \]

### B.2. Interface Reduction

\[ I = \left\{ (p^1, A^1), (p^2, A^2), (p^6, A^6), (p^9, A^9), (p^{10}, A^{10}), (p^{12}, A^{12}), (p^{13}, A^{13}) \right\} \]

\[ P = \{ \text{sys, j_pos, ee_pos, tool_st, tool, coll_dist, ob_ft} \} \]

\[ p^1_v = \{ \text{inactive, enabled, idle, shutdown, estop, homing, tool, val, hold, cancel, resume} \} \]

\[ p^1_v = \{ \text{error } S\#, \text{error } L\#, \text{error } E\#, \text{error } T\#, \text{error } C\#, \text{error } \text{clear} \} \]

\[ A^1 = \{ \text{set, get} \} \]

\[ a_{\text{set}}^1 = p^1_v \]

\[ a_{\text{get}}^1 = \emptyset \]

\[ p^2_v = (\mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R}) \]

\[ A^2 = \{ \text{set, get} \} \]

\[ a_{\text{set}}^2 = p^2_v \]

\[ a_{\text{get}}^2 = \emptyset \]

\( A^6 = \{ \text{set, get} \} \)

\( a\_\text{set}^6_v = p^6_v \)

\( a\_\text{get}^6_v = \emptyset \)

\( p^{9}_v = \{ \text{off, on, fwd, rev, open, closed, opening, closing, 0–1} \} \)

\( A^9 = \{ \text{set, get} \} \)

\( a\_\text{set}^9_v = p^9_v \)

\( a\_\text{get}^9_v = \emptyset \)

\( p^{10}_v = \{ \text{none, gripper, drill, paint\_gun, saw} \} \)

\( A^{10} = \{ \text{set, get} \} \)

\( a\_\text{set}^{10}_v = p^{10}_v \)

\( a\_\text{get}^{10}_v = \emptyset \)

\( p^{12}_v = \mathbb{R} \)

\( A^{12} = \{ \text{get} \} \)

\( a\_\text{get}^{12}_v = \emptyset \)

\( p^{13}_v = \{ [ (R, R, R), (R, R, R) ] \} \)

\( A^{13} = \{ \text{get} \} \)

\( a\_\text{get}^{13}_v = \emptyset \)
B.3. Optimized Interface in EBNF

\[\text{robot\_interface} = \left[ \begin{array}{c}
\text{sys\_set}, \text{j\_pos\_set}, \text{ee\_pos\_set}, \text{tool\_st\_set}, \text{tool\_set} \\
\text{sys\_get}, \text{j\_pos\_get}, \text{ee\_pos\_get}, \text{tool\_st\_get}, \text{tool\_get}
\end{array} \right] \]

\[\text{sys\_set} = \text{"set::"}, \text{sys\_property} = \left( \text{"sys::"}, \left( \text{"inactive" | "idle" | "tool" | "val"} \right) \right)\]

\[\text{sys\_get} = \text{"get::sys"}\]

\[\text{j\_pos\_set} = \text{"set::"}, \text{j\_pos\_property}\]

\[\text{j\_pos\_get} = \text{"get::j\_pos"}\]

\[\text{ee\_pos\_set} = \text{"set::"}, \text{ee\_pos\_property}\]

\[\text{ee\_pos\_get} = \text{"get::ee\_pos"}\]

\[\text{tool\_st\_set} = \text{"set::"}, \text{tool\_st\_property} = \left( \text{"tool\_st::"}, \left( \text{"opening" | "closing"} \right) \right)\]

\[\text{tool\_st\_get} = \text{"get::tool\_st"}\]

\[\text{tool\_set} = \text{"set::"}, \text{tool\_property}\]

\[\text{tool\_get} = \text{"get::tool"}\]

\[\text{coll\_dist\_get} = \text{"get::coll\_dist"}\]

\[\text{ob\_ft\_get} = \text{"get::ob\_ft"}\]

\[\text{sys\_property} = \text{"sys::"}, \text{sys\_t}\]

\[\text{sys\_t} = \text{init\_t | end\_t | move\_t | hold\_t | err\_t}\]

\[\text{init\_t} = \text{"inactive" | "enabled" | "idle"}\]

\[\text{end\_t} = \text{"shutdown" | "estop"}\]

\[\text{move\_t} = \text{"homing" | "tool" | "val"}\]

\[\text{hold\_t} = \text{"hold" | "cancel" | "resume"}\]

\[\text{err\_t} = \text{"error\_", \text{digit} | "clear"}\]

\[\text{digit} = 0' | 1' | 2' | 3' | 4' | 5' | 6' | 7' | 8' | 9'\]

\[\text{j\_pos\_property} = \text{"j\_pos::"}, \text{real\_val}, 5*'(', real\_val),'), ('\text{"deg" | "rad"}\]

\[\text{real\_val} = \text{[+'-', \\
\{digit\}, \text{[', ;', digit\}]}\]

\[\text{ee\_pos\_property} = \text{"ee\_pos::"}, \text{vals}, ('\text{"in" | "cm"}, ', \\
\text{\{fixed | euler\}, ('\text{"deg" | "rad" | mat\}]}\]

\[\text{vals} = '(' , real\_val, 2*(', real\_val)')\]
`fixed = "fixed:", vals;
euler = "euler:", vals;
mat = "mat:('', real_val, 8*(',', real_val))';
tool_st_property = "tool_pow::", (grip | drill | paint_gun | saw);
grip = "open" | "closed" | "opening" | "closing";
drill = "fwd" | "rev" | "off";
paint_gun = "on" | "off";
saw = ([0], [',', {digit}]) | [1], [',', {'0'}] | "off";
tool_property = "tool::", ("none" | "gripper" | "drill" | "paint_gun" | "saw");
coll_dist_property = "coll_dist::", pos_real_val, ("in" | "cm");
pos_real_val = ({digit}, [',', {digit}]);
ob_ft_property = "ob_ft::["vals,"N", fixed,"N-m"]";`
Appendix C

C. Interface Samples

C.1. C++

C.1.1. Interface

```cpp
enum tool_property{none, gripper, drill, paint_gun, saw};
enum tool_st_property{off, on, fwd, rev, open, closed, opening, closing};
enum sys_property{inactive, enabled, idle, shutdown, estop, homing, tool,
                  val, hold, cancel, resume, error_S, error_L, error_E,
                  error_T, error_C, error_clear};
enum angle{deg, rad};
enum dist{in, cm};
enum orientation{fixed, euler};
enum force{N};
enum torque{Nm};

class robot{
  public:
    void set_tool(tool_property);
    void set_tool_st(tool_st_property);
    void set_tool_st(float);
    void set_j_pos(float, float, float, float, float, float, angle);
    void set_ee_pos(float, float, float, dist, orientation,
                    float, float, float, angle);
    void set_ee_pos(float, float, float, dist,
                    float, float, float,
                    float, float, float);
    void set_sys(sys_property);
    void set_sys(sys_property, unsigned int);

    void get_j_pos(float&, float&, float&, float&, float&, float&, angle);
    void get_sys(sys_property&);
    void get_sys(sys_property&, unsigned int&);
};
```
void get_ee_pos(float&, float&, float&, dist, orientation, float&, float&, float&, angle);
void get_ee_pos(float&, float&, float&, dist, float&, float&, float&, float&, float&, float&);
void get_tool_st(tool_pow_property&);
void get_tool_st(float&);
void get_tool(tool_property&);
void get_coll_dist(float&, dist);
void get_ob_ft(float&, float&, float&, force, orientation, float&, float&, float&, torque);
};

C.1.2. Usage

int main()
{
  robot r;
  r.set_tool(none);
  r.set_tool_st(off);
  r.set_tool_st(0.363);
  r.set_j_pos(3.2, 1.2, 4.5, 4.3, 6.3, 9.7, deg);
  r.set_j_pos(3.2, 1.2, 4.5, in, fixed, 4.3, 6.3, 9.7, deg);
  r.set_j_pos(3.2, 1.2, 4.5, in, 1., 0., 0., 0., 0., 0., 0., 1.);
  r.set_sys(enabled);
  r.set_sys(error_S, 4);

  float f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12;
  sys_property sp;
  unsigned int ui;
  tool_st_property ts;
  tool_property t;

  r.get_j_pos(f1, f2, f3, f4, f5, f6, deg);
  r.get_sys(sp);
  r.get_sys(sp, ui);
  r.get_ee_pos(f1, f2, f3, in, fixed, f4, f5, f6, deg);
  r.get_ee_pos(f1, f2, f3, in, f4, f5, f6, f7, f8, f9, f10, f11, f12);
  r.get_tool_st(ts);
  r.get_tool_st(f1);
  r.get_tool(t);
  r.get_coll_dist(f1, in);
  r.get_ob_ft(f1, f2, f3, N, fixed, f4, f5, f6, Nm);

  return 0;
}
C.2. XML

C.2.1. Interface

```xml
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="robot" elementFormDefault="qualified"
xmlns="robot" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="robot">
    <xs:complexType>
      <xs:all>
        <xs:element name="set" minOccurs="0">
          <xs:complexType>
            <xs:choice minOccurs="0">
              <xs:element name="tool" minOccurs="0">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:enumeration value="none" />
                    <xs:enumeration value="gripper" />
                    <xs:enumeration value="drill" />
                    <xs:enumeration value="paint_gun" />
                    <xs:enumeration value="saw" />
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
              <xs:element name="tool_st" minOccurs="0">
                <xs:simpleType>
                  <xs:union>
                    <xs:simpleType>
                      <xs:restriction base="xs:float">
                        <xs:minInclusive value="0" />
                        <xs:maxInclusive value="1" />
                      </xs:restriction>
                    </xs:simpleType>
                    <xs:simpleType>
                      <xs:restriction base="xs:string">
                        <xs:enumeration value="off" />
                        <xs:enumeration value="on" />
                        <xs:enumeration value="fwd" />
                        <xs:enumeration value="rev" />
                        <xs:enumeration value="open" />
                        <xs:enumeration value="closed" />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:union>
                </xs:simpleType>
              </xs:element>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:element name="euler">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="floatList3">
        <xs:attribute name="unit">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="deg" />
              <xs:enumeration value="rad" />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<xs:element name="mat">
  <xs:simpleType>
    <xs:restriction>
      <xs:simpleType>
        <xs:list itemType="xs:float"/>
      </xs:simpleType>
      <xs:length value="9"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="sys">
  <xs:complexType>
    <xs:choice>
      <xs:sequence>
      </xs:sequence>
    </xs:choice>
  </xs:complexType>
</xs:element>
C.2.2. Usage

C.2.2.1. Set Tool

```xml
<robot xmlns="robot">
  <set>
    <tool>gripper</tool>
  </set>
</robot>
```

C.2.2.2. Set Tool State (State) and Set Joint Position

```xml
<robot xmlns="robot">
  <set>
    <tool_st>fwd</tool_st>
    <j_pos unit="deg">2.3 3.4 6.3 7.3 9.3 2.7</j_pos>
  </set>
</robot>
```

C.2.2.3. Set Tool State (Value) and Set End-Effector Position (Fixed)

```xml
<robot xmlns="robot">
  <set>
    <tool_st>0.342</tool_st>
    <ee_pos>
      <pos unit="in">3.2 -1.8 8.2</pos>
      <fixed unit="deg">1.2 -1.0 0.2</fixed>
    </ee_pos>
  </set>
</robot>
```

C.2.2.4. Set End-Effector Position (Matrix)

```xml
<robot xmlns="robot">
  <set>
    <ee_pos>
      <pos unit="in">3.2 -1.8 8.2</pos>
      <mat>1. 0. 0. 0. 1. 0. 0. 0. 1.</mat>
    </ee_pos>
  </set>
</robot>
```

C.2.2.5. Set System State

```xml
<robot xmlns="robot">
  <set>
  </set>
</robot>
```
C.2.2.6. Set System State (Error)
<robot xmlns="robot">
  <set>
    <sys>error_S4</sys>
  </set>
</robot>

C.2.2.7. Get
<robot xmlns="robot">
  <get>
    <j_pos />
    <sys />
    <ee_pos />
    <tool_st />
    <tool />
    <coll_dist />
    <ob_ft />
  </get>
</robot>
References


Vita

Ross Corey Taylor was born in Shreveport, Louisiana on December 19, 1978, the son of Susan Stone Taylor and John Richard Taylor. After his 1997 graduation from C. E. Byrd High School in Shreveport, Louisiana, he enrolled at Louisiana Tech University in Ruston, Louisiana. In May 2001, he received the degree of Bachelor of Science in Mechanical Engineering from Louisiana Tech University. In September 2001, he entered the Mechanical Engineering Graduate Program at The University of Texas in Austin, Texas.

Ross was married to Elizabeth Cole Mitchell Taylor on August 17, 2002, in Shreveport, Louisiana. They now live in Houston, Texas, where he is employed by Neptec International, Incorporated.

Publications:


Permanent Address: 237 Symphony Lane
Shreveport, Louisiana 71105

This thesis was typed by the author.