A Method for Mapping between ASMs and Implementation Language

by

David Cheng-Ping Wang

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2010

Author .....
Department of Aeronautics and Astronautics
January 29, 2010

Certified by .....
I. Kristina Lundqvist
Professor, Mälardalen University
Thesis Supervisor

Certified by .....
R. John Hansman
Professor, Massachusetts Institute of Technology
Thesis Supervisor

Accepted by .....
Eytan H. Modiano
Associate Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

A Method for Mapping between ASMs and Implementation Language

by

David Cheng-Ping Wang

Submitted to the Department of Aeronautics and Astronautics
on January 29, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

One of the challenges of model-based engineering is traceability: the ability to relate the set of models developed during the design stages to the implemented system. This thesis develops a language specific method for creating *bidirectional traceability*, a mapping between model and implementation, suitable for tracing requirements from model through implementation and vice versa. The mapping is created as a by-product of code generation and reverse engineering, and can be used to subsequently synchronize changes between the model and implementation.

The creation of the mapping is specifically demonstrated through generating Java code from an abstract state machine (ASM) based modeling language, called the Timed Abstract State Machine (TASM) language. This code generation process involves a series of three transformations. The first transformation creates a specialised System Dependency Graph (SDG) called a TASM SDG from a TASM specification. The second uses Triple Graph Grammars to transform the TASM SDG to a Java SDG (JSDG). The applied grammars are saved as the mapping information. The third transformation procedurally generates Java code. In order to make this methodology possible, this thesis introduces the TASM SDG, as well as a novel algorithm, generally applicable to ASM languages, that explicates state transitions.

The approach presented extends the bidirectional traceability capabilities inherent in the TASM language to Java. The code generation technique is demonstrated using an industrial case study from the automotive domain, an Electronic Throttle Controller (ETC).

Thesis Supervisor: I. Kristina Lundqvist
Title: Professor, Mälardalen University

Thesis Supervisor: R. John Hansman
Title: Professor, Massachusetts Institute of Technology

Acknowledgments

I would like to express my gratitude to all who have supported me in the completion of this thesis.

First, to my advisor Kristina Lundqvist: You have been more patient with me than I deserve and I could have ever asked for. Thank you for your continued faith in me and support through the years.

My thanks to Martin Ouimet for his care laying out TASM, the language on which this thesis is built and for his advice in both matters life and academic.

My thanks and love to my girlfriend of 6 years, Justine, who kept me company when I was burning the midnight oil. She helped keep me sane when I had to surmount the impossible and she reminds me of all the good things that will come when I pass through the tunnel of being a graduate student.

Foremost, I thank my parents, Priestley and Nancy Wang. You taught me of perseverance, of optimism, and instilled in me a love of learning that has led me to this point and will stay with me for the rest of my life.

Contents

1 Introduction **15**

1.1 Objective 15

1.2 Context 15

1.2.1 Modeling Language — TASM 16

1.2.2 Implementation Language — Java 16

1.3 Background 16

1.3.1 Correctness 17

1.3.2 Software Engineering 17

1.3.3 Traceability 19

1.3.4 Verification and Validation 19

1.3.5 Traceability in Avionics 20

1.4 Motivation 20

1.4.1 Decay of Traceability Information 20

1.4.2 Related Work - Automating Traceability 21

1.4.3 Related Work - Traceability Recovery 21

1.5 Approach 22

1.5.1 Functional Equivalence 23

1.5.2 Application 23

1.6 Contributions 25

1.7 Thesis Layout 26

2 Languages **27**

2.1 ASM 27

2.1.1 Origins . . . 27
2.1.2 Application . . . 28
2.1.3 Syntax & Semantics . . . 29
2.2 TASM . . . 31
2.2.1 Behavior . . . 32
2.2.2 Language Introduction . . . 33
2.2.3 Overview . . . 34
2.2.4 Syntax . . . 37
2.2.5 Semantics . . . 52
2.2.6 Supported TASM . . . 57
2.3 Java . . . 58
2.3.1 Supported Java . . . 58
2.4 Segue into Chapter 3 . . . 59

3 TASM System Dependency Graph **61**
3.1 Technical Motivation . . . 61
3.1.1 Related Work . . . 62
3.2 TASM System Dependence Graph . . . 62
3.2.1 Edge Types . . . 62
3.2.2 Vertex Classes . . . 63
3.2.3 Construction . . . 67
3.2.4 Example . . . 68
3.3 Extracting Transitions . . . 72
3.4 Implicit Transitions . . . 72
3.5 Variable Space Introduction . . . 74
3.6 Extracting Transitions . . . 74
3.6.1 Algorithm . . . 75
3.6.2 Example . . . 75
3.6.3 Soundness . . . 78
3.6.4 Termination . . . 78

4 Mapping from ASM to Java **79**

4.1 Triple Graph Grammars 79

4.1.1 Syntax 79

4.2 Code Generation 80

4.3 Reverse Engineering 80

4.4 Synchronization 81

5 Case Study: Electronic Throttle Controller **83**

5.1 Electronic Throttle Controller 83

5.2 Implementation 86

5.3 Results 87

5.3.1 Code Generation 87

6 Conclusion **89**

6.1 Future Work 89

6.1.1 Object Oriented Languages 90

6.1.2 Advanced Language Features 90

6.1.3 Reverse Engineering 90

6.1.4 Test Case Generation for Timing and Resources 91

A TASM Language Reference **93**

A.1 TASM Objects 93

A.1.1 Specification 93

A.1.2 Project 93

A.1.3 Environment 94

A.1.4 Main Machine Template 94

A.1.5 Function Machine 95

A.1.6 Sub Machine 95

A.1.7 Configuration 95

A.2 Syntax 95

A.2.1 Notational Conventions 95

A.2.2 Names . . . 96
A.2.3 Types . . . 96
A.2.4 Arithmetic Operators . . . 98
A.2.5 Logical Operators . . . 99
A.2.6 Context-Free Grammar . . . 99
A.3 Semantics . . . 106
A.3.1 Operator Precedence . . . 106
A.3.2 Calling Convention . . . 108
A.3.3 Types . . . 108
A.3.4 Relation to Abstract State Machines . . . 108
A.3.5 Sugaring/Desugaring . . . 108
A.3.6 Resource definitions . . . 109
A.3.7 Type definitions . . . 109
A.3.8 Variables . . . 109
A.3.9 Rules . . . 109
A.3.10 Execution Semantics . . . 111

B Electronic Throttle Controller — TASM Model 113
B.1 Environment . . . 116
B.2 Main Machines . . . 118
B.3 Function Machines . . . 125
B.4 Sub Machines . . . 130

C Electronic Throttle Controller — Java Model 141

List of Figures

1-1 Proposed Traceability Framework 23

1-2 Program A and B cannot be mapped using method based input-output equivalence 24

1-3 Code Generation, Reverse Engineering, and Synchronization via a tool 25

2-1 title-body nesting structure 34

2-2 Light Switch Example 36

2-3 Moore FSM of Light Switch Example 36

2-4 Example time line for a run of a single TASM 55

2-5 Example time line for a run of two TASMs 57

3-1 An example of the three types of data dependencies. 64

3-2 Light Switch Example Partial, Hierarchial TASM SDG 71

3-3 The global states of the light switch example. 72

3-4 The global states of the light switch example. 73

3-5 Simplified TASM Specification with 3 rules. 76

3-6 The charts represent the variable space, divided into regions in which each rule is true. A. The region defined by the initial values are outlined in black, and are shown to trigger (intersect the regions defined by) rules 1 & 3. B. The evaluation of rule 1s effect (y:=y-x) via the properties of interval arithmetic alters the shape of the region and triggers rules 1 & 2. Evaluation proceeds in a branching manner, with each intersection identifying an abstract state transition. 77

4-1 Code Generation 80
4-2 Reverse Engineering 81
4-3 Synchronization 81

5-1 High level Simulink model of the ETC 84
5-2 ETC modes 85
5-3 ETC tasks and scheduler 86

List of Tables

A.1 Reserved keywords 97
A.2 Operators 100
A.3 Operator precedence 107

B.1 List of machines used in the low level ETC model (part 1) 114
B.2 List of machines used in the low level ETC model (part 2) 115

C.1 List of Java files generated from the low level ETC model. 141

Chapter 1

Introduction

In this chapter we present an overview of the thesis. Section 1 states the objective. Section 2 presents the context in which that objective was achieved. Section 3 presents some background for the context. Section 4 uses that background to frame the motivations for this research. Sections 5 and 6, summarize the important points of the research and its innovations. Finally Section 7 lays out the remainder of the thesis.

1.1 Objective

The objective of this thesis is to describe a methodology for creating bidirectional traceability information between an Abstract State Machine (ASM) model of a system and its implementation. By *implementation*, we refer to a program written in a procedural manner (no classes/objects). The language in which the implementation is written, is referred to as the *implementation language*, which is required to execute sequentially and use variables. Such languages usually target a von Neumann architecture.

The framework used by the methodology should be generally applicable such that it can be used for the model-based software engineering processes of *code generation*, *reverse engineering*, and *synchronization*. Where: Code generation is, as its name implies, the automated creation of implementation code from a model[s]. Reverse

engineering is the dual of code generation, and involves the extraction of a model[s] from code. Synchronization is the process of merging changes from either the model[s], implementation code, or both, to produce up-to-date model[s] and implementation.

1.2 Context

The work is achieved in the context of the Hi-Five Framework, an integrated framework for the validation and verification of embedded real-time systems. The framework uses a model-based approach to software engineering, with a custom modeling language based on Abstract State Machine theory, called the Timed Abstract State Machine (TASM) language.

As a result of the context, the work was impacted in two important ways. First, our approach to achieving bidirectional traceability is restricted to two languages: the TASM modeling language and the Java implementation language. Second, one of the benefits of the Hi-Five Framework and, in general, model-based software engineering, is the ability to prove various correctness properties of a model prior to its implementation. In order to preserve that correctness from model to implementation, our approach will focus on using functional correctness preserving traceability relations.

1.2.1 Modeling Language — TASM

For a modeling language, the Timed Abstract State Machine (TASM) language was used. The TASM language allows for a text-based, instead of graphical, expression of ASM theory [23, 27] which explicitly captures timing and resource consumption. TASM is also capable of representing multiple concurrently operating hierarchial ASMs with synchronization primitives. This language forms the basis of a larger body of work called the Hi-Five framework, which leverages state-of-the-art model checkers for the validation and verification of embedded real-time systems represented with the TASM language [33]. This research extends the traceability already available in this framework between equivalent TASM models of different levels of detail to implementation code.

In order to support TASM, a GUI based IDE was developed called the *TASM Tool*. The TASM Tool serves as the implementation of the Hi-Five framework and provides a convenient interface to the associated analysis engines.

1.2.2 Implementation Language — Java

Java was chosen as the implementation language due to its increasing popularity and the increasing popularity of object oriented development. While this thesis only uses the functional aspect of Java (i.e. static methods and fields), the use of Java should make it easier to extend this research to incorporate objects in the future.

From hereafter, examples and statements dependent on the language choice will explicitly state TASM or Java. The terms "model" and "specification" will be used interchangeably to refer to a document written in an ASM language. The terms "code" or "implementation" will be used to refer to the document written in an implementation language. Chapter 2 will elaborate on ASMs, as well as what subsets of the TASM and Java language are supported.

1.3 Background

In this section, we provide a background on a variety topics important towards understanding the context. We specifically present: correctness, software and model-based software engineering, and traceability. We spend a large portion of the background presenting traceability, and its use in verification and validation activities to stress its importance in the field of software engineering.

1.3.1 Correctness

When we create a computer system, we want it to function 'correctly.' But, there are many ways to classify the correctness of a computer system. We most commonly wish a computer to exhibit *functional correctness*: given a certain input (stimulus), the computer should produce a certain output (response). But, there are also many forms

of *non-functional correctness*, including timing, power consumption, heat dissipation, and resource consumption, that we may want a system to satisfy.

To help understand the difference between the two, we present a system that requires both. A *real-time system* is a computer system that must exhibit both functional and timing or *temporal correctness*. A computer system is said to be temporally correct when it accomplishes a *task* within certain time bounds. This bound is often expressed as an upper limit, or *deadline*. A functionally correct system which produces an output after the deadline is said to have "missed the deadline." How a missed deadline impacts the correctness of the system depends on whether the system is a soft or hard real-time system. In a *soft* real-time system, a miss may be acceptable under certain circumstances. In a *hard* real-time system, they are not. For example, it may be acceptable for a DVD player to miss rendering frames when switching between menus (soft real-time), but unacceptable for a missile navigation system to miss updating its position (hard real-time). Since specifying an appropriate deadline for a task is highly dependent on the hardware on which the task is running, one way to help bound the worse case execution time is to design the software and hardware of a computer system together. Real-time systems designed in this manner are referred to as *real-time embedded systems*.

In this thesis, we will focus on using and preserving functional correctness only. However, this research was completed in the context of extending an existing body of research benefiting real-time systems development so an understanding of non-functional correctness is important to appreciate what aspects of the TASM language we will not support.

1.3.2 Software Engineering

Software engineering is the set of techniques, processes, and tools used to develop computer systems [43]. Traditionally, a software engineering project is divided into life cycle phases starting with requirements engineering, design, implementation, testing, and ending with maintenance [46]. While these phases are generally completed in order, the sequence may change depending on the process model employed. Projects

which employ this exact sequence are said to use the *waterfall-model.* Projects with many intermediate demonstrations may favor a *spiral-model*, which repeats the 4 steps of requirements engineering, design, implementation and testing for each increase in functionality. There are many such process models and the choice of which model to use is often driven by budgetary constraints, deadlines, software size, tradition, as well as the degree of correctness required. The techniques and tools used during the various phases can be equally varied, but is traditionally paper-oriented: and involve preparing a requirements document, architecture diagrams, and interface specifications prior to implementation [3, 1].

Model-based software engineering (MBSE), also known as model-driven software engineering (MDSE) and model-driven architecture (MDA), is an approach towards software engineering in which the techniques and tools used revolve around developing models. Not to be confused with process models, the word 'model' in MBSE refers to the different types or levels of abstraction of a system's desired behavior (like an input-output or timing model). Used primarily during the design phase, the gradations of detail possible with models make them ideal for bridging the discrete phases of requirements engineering and implementation. But, models can also be used to facilitate the other phases through generating code and identifying test cases. A key benefit of this approach is the ability to uncover defects early in the development process. A draw back is that the models can easily become outdated or disconnected, requiring an additional effort to propagate design choices between them.

How a model is specified depends on the *modeling language* used. Modeling languages can be graphical or text-based, formal (with precise syntax) or informal. Formal languages have the benefit of expressing behaviors verifiable by automated model-checkers and theorem provers, but can be difficult to understand and may be restrictive in what they can express. In general, certain languages can be better suited for representing certain behaviors, and each language comes with varying levels of tool support for automating engineering activities. The benefits of a model-based approach occur when a literate notation with formal semantics is used, so the models can serve the dual purpose of being documentation and an analysis mechanism [33].

Two popular examples of modeling languages come from MBSE standards proposed by two professional groups. The Object Management Group (OMG) founded the Unified Modeling Language (UML), a graphical language with subsets allowing the creation of a wide variety of diagrams capable of expressing structure, flow, use-case, and timing. As of the writing of this thesis, the most current version of UML is 2.0, which adds diagrams for expressing functional behavior. But, a lack of formal semantics limits the amount of automated analysis possible. The Architecture and Analysis Language (AADL) is a modeling language endorsed by the Society of Automotive Engineers (SAE). In contrast to UML, AADL has both textual and graphical formal semantics, allowing AADL models to be analyzed for reliability, data quality, timing, security, and resource consumption. However, AADL models are only suited to express high level component interaction, and lacks semantics for expressing component level behavior.

The modeling language used in this thesis is the Timed Abstract State Machine (TASM) language [33]. In comparison to UML and AADL, TASM's basis in ASM theory allows the language to overcome several of the afore mentioned shortcomings. Namely, it provides a single set of formal semantics capable of unifying several behaviors in the same model. And, it allows hierarchical composition, suitable for representing component behaviors. However, it was not intended to supplant either language: as it lacks a graphical representation and cannot explicitly represent data structures and time lines.

1.3.3 Traceability

Traceability, in the most general sense, is the ability to study out in detail or step by step. In software engineering, 'traceability' is a property of the software development process, it is the presence of information that relates one step of development to another. Note that this information relates to how software is made, and not how it functions (i.e. monitoring execution traces). For example, we can add traceability to a system by recording that a fragment of code was written to fulfill a requirement. However, this overarching notion of traceability can be decomposed: The develop-

ment process may involve creating architecture diagrams from the requirements, then inheritance models, and state-charts, prior to writing any code. Even within the code, abstractions are made to make the interface distinct from the underlying code or drivers. Software traceability can therefore consist of mapping requirements to a model, from a model to a refined model, model to code, one version of code to another, or between any abstraction level there-in.

Forward traceability refers to the addition of information to a higher level of abstraction that state how it relates to a lower, more detailed level of abstraction. Whereas *backward traceability* adds information to lower abstractions that state how it relates to a higher level. For a simple software development cycle where code is directly created from a set of requirements, forward traceability may take the form of adding lines beneath each requirement in a requirements document stating which section of code satisfies it, and backward traceability could be supplied in code comments stating which requirement a method satisfies.

But, maintaining forward and backward traceability information does not need to be so disparate. The term *bidirectional traceability* unifies these two notions to encompass the idea that traceability is more naturally described as a bidirectional mapping between a higher and lower level of abstraction. Instead of maintaining code comments, one popular way of achieving bidirectional traceability is through a *traceability matrix* which serves as the bidirectional map. In a requirements traceability matrix, the rows correspond to different requirements, the columns to different features, and the body of the matrix correlates features to requirements. For simplicity, the term *traceability* will be used hereafter to refer to bidirectional traceability.

1.3.4 Verification and Validation

Maintaining traceability information helps software engineers understand code through its relation to other models and artifacts. The importance of these traceability relations and the understanding they entail is formalized in a process called Verification and Validation.

A major challenge of software engineering is producing software with high-reliability:

high-confidence in its correctness, despite a trend in increasing software size and complexity. In mission critical software, where high-reliability is required, one way to establish confidence in the correctness of a system is by undertaking Verification and Validation (V & V) activities in parallel with the software development life cycle. *Verification* provides quality assurance through documentation of the development process. *Validation* provides quality control through testing of the end-product. Both activities are necessary. Since even small projects cannot be feasibly, exhaustively tested, verification helps assure correctness for those execution traces that were not validated. Verification is premised on the notion that software can be "correct by construction": that thoughtful adherence to a coding and documenting procedure can help assure reliability.

1.3.5 Traceability in Avionics

Traceability is especially important in large or complex software systems that have long life-spans requiring upgrade and maintenance. A particularly prevalent example of such a software system occurs in *avionics*. *Avionics*, a contraction of "aviation electronics", is an example of a real-time embedded system: the term collectively describes all the electronic and software components involved in a plane's control, communication, and navigation systems.

Traceability is a key aspect in compliance with DO-178B, "Software Considerations in Airborne Systems and Equipment Certification"[39]. Originally created by the Radio Technical Commission for Aeronautics (RTCA) and the European Organization for Civil Aviation Equipment (EUROCAE) in 1992, DO-178B is the accepted guideline for safety-critical software production required for certification of an avionics system by the Federal Aviation Administration (FAA), European Aviation Safety Agency (EASA), and Transport Canada.

DO-178B specifically calls for a large body of requirements and traceability documentation to be maintained in parallel with software development to prove that [28]: (1) The software completely satisfies all the specified system requirements; (2) Every single code instruction of the software is necessary and serves its intended purpose;

and (3) No unintended code exists in the software, and whatever non-essential code that may exist for portability, robustness or similar reasons will not detrimentally impact the software's reliability from a safety perspective.

1.4 Motivation

Traceability is important in the development process, V&V, and maintenance. Maintaining traceability is important not only during development to ensure the code satisfies the requirements, but also during debugging and maintenance to understand why certain code exists and ensure any changes still reflect the requirements.

1.4.1 Decay of Traceability Information

Unfortunately, the natural language documents which are traditionally used to capture requirements and design decisions, can be ambiguous, miss details, and be expensive in cost and time to maintain. Maintaining such traceability information is challenging, even in organizations with established software development processes. The difficulty arises from many factors, including: the development of software in different languages, the use of models at different levels of abstraction or expressing different concepts (i.e. behavioral vs. inheritance structure), processes that do not enforce the maintenance or creation of traceability links, and a lack of tool support [34]. Consequently, verifying software or comprehending existing software can require a lot of time and expense to recreate the connections between seemingly disparate documents, models, and code.

1.4.2 Related Work - Automating Traceability

In the 80's, computer-aided software engineering (CASE) tools were developed that could maintain traceability information through the use of a *traceability matrix*. However, such tools still relied on the manual entry of traceability relationships. If automated, the tool would create and store traceability information as a byproduct of

using the tool for an engineering activity, but this usually required a strict adherence to a particular modeling language or development process [41].

Since the 80's, the software development process has changed greatly. Spiral and other iterative development models are preferred over document dependent waterfall models. Procedural programming has been replaced with Object Oriented programming. And, personal computers can now display rich graphics in addition to textual summaries. These changes sparked development in the area of model-based CASE tools. Languages such as the Universal Modeling Language (UML) and its many supporting tools, such as UModel, Agile Platform, IBM's Rational XDE, etc... , allow the graphical modeling of behaviorial and structural aspects of software.

Today, tools such as UModel can not only maintain bidirectional traceability information between models, but during an expanded set of capabilities including code generation, reverse engineering, and synchronization. Traceability information relating to changes to or relationships between models usually take the form of saving the sequence of user interactions with the tool. However, code generation, reverse engineering, and synchronization produces code that is usually not modified through the tool, but an external IDE.

Without the ability to save user interactions as traceability information, and restricted by UML's lack of formal semantics, the resulting traceability information is primitive, relying on function signatures, class structures, and file names to relate model to code. Without careful forethought, a major reorganization of names can obviate the use of the tool and require manual reconciliation.

1.4.3 Related Work - Traceability Recovery

A lot of work has been done to recover traceability information from between model and code [2, 12, 1, 3]. These approaches to building traceability information require the analysis of the documentation and code to search for matching 'artifacts,' usually in the form of similar names. Unfortunately, since these techniques rely on recovery, the dictionary of traceability information they create is incomplete. Unlike a natural language document, a TASM language (and all ASM languages) have the formal

semantics required to make them more amenable to such strategies of traceability recovery. Nevertheless, the traceability information is still imprecise. To this author's knowledge, there have been no attempts at creating traceability information between an ASM language and its implementation.

1.5 Approach

In order to achieve bidirectional traceability between TASM and Java, we developed a mapping between the two languages that consists of a sequence of transformations. A *transformation* between languages is a set of rules that describe the conversion of one syntactic structure into another related syntactic structure. A transformation from one document to another, such as between an ASM specification and code, is the application of those rules. By storing the manner in which those rules are applied, we maintain traceability information.

Our specific approach uses a sequence of transformations that leads through 2 intermediate representations based on the System Dependency Graph (SDG) [26]. One of the representations is a specialized SDG for the TASM language introduced in this thesis, called the TASM SDG. The other is an existing SDG for the Java language called the Java SDG (JSDG). The original SDG was an extension of the Program Dependency Graph (PDG), a concise merging of a control-flow and data dependency graph that completely describes a program's functional behavior. An SDG adds to the PDG the ability to model multiple procedures and scope. These intermediate forms help distill language specific syntax to a set of semantics universal to both languages. SDGs will be discussed in more detail in chapter 3.

In general, two transformations can be defined between any two structures: a transformation and its inverse. But, the pairs we are concerned with include the transformations between a TASM specification and a TASM SDG, between a TASM SDG and a JSDG, and between a JSDG and Java code [Figure 1-1]. Instead of storing traceability information for all of those transformations, this thesis defines a procedural transformation between TASM and TASM SDG, and leverages existing

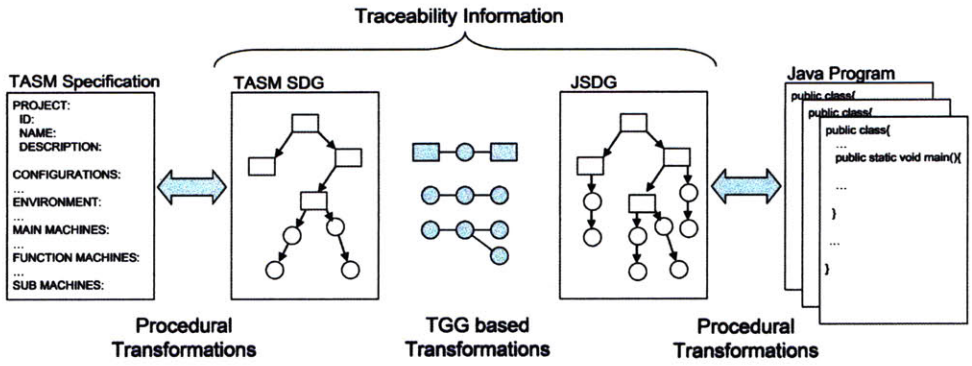

Figure 1-1: Proposed Traceability Framework

procedural transformations between Java and the JSDG. For the remaining pair of transformations between the SDG intermediaries, we use Triple Graph Grammars.

One of the problems when using a typical transformation is that the set of rules is unidirectional. As a result, the transformation from one document to another only uses one set of rules and thus only creates traceability in one direction. To overcome this problem, we use Triple Graph Grammars (TGG), a graph-based language to describe the transformations between the two SDGs. Each TGG grammar can express a rule and its inverse. Therefore, recording the transformation from one document to another using TGG grammars automatically stores both the forward and reverse traceability information. TGGs will be discussed in more detail in chapter 4.

The approach results in a robust mapping. By composing different TGG grammars, the transformations between the SDGs need not be rigid. SDGs have been shown to be an efficient graph structure for the code transformations used in compiler optimizations, and a large body of work exists describing such transformations. Instead of just applying a standard set of TGGs to create a JSDG from a TASM SDG or vice versa, other code transformations can be used to reorganize an SDG. Section 3.1 gives an example to illustrate the importance of robust mapping.

1.5.1 Functional Equivalence

One of the reasons the framework requires so many transformation sequences is a desire to robustly preserve functional equivalence. That is, to be able to describe transformations between two programs for all of the different ways each language can describe the same functional behavior. Unlike the UML tools which must crudely rely on superficial name and syntax based transformations because the UML language lacks formal semantics, the TASM modeling language has a well defined set of formal semantics for which functional behavior can be precisely described.

Functional equivalence is the notion of equivalence used in compiler optimizations and program slicing, and is the form of equivalence established by the Church-Turing thesis between recursion, a turing machine, and lambda-calculus. To be more specific, we will say that two programs are equivalent if for every input they produce the same output. Figure 1-2 illustrates two functionally equivalent programs that sequentially call methods, which themselves may not be functionally equivalent. This distinction between saying two programs are functionally equivalent at the top-level versus saying two programs are functionally equivalent because their underlying structures are functionally equivalent is important. If traceability information only maps structures with identical functionality, moving a single line of code from one method to another would break the mapping, resulting in the false conclusion that the programs are not equivalent.

1.5.2 Application

The transformation framework can be implemented in a tool and be used for code generation, reverse engineering, and synchronization [Figure 1-3]. Due to the design of the framework, any process that uses the framework automatically generates traceability information through the TGGs. The traceability information created aids in synchronization, but can also have many of its own applications.

At the simplest level, a tool could be built on top of this mapping that adds comments to TASM indicating which class, method or statement implements that

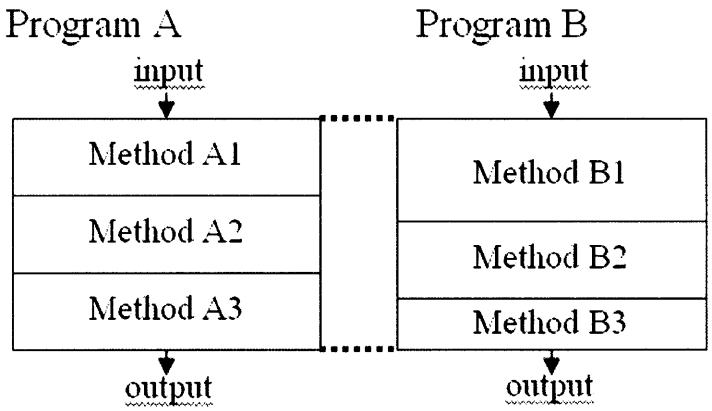

Figure 1-2: Program A and B cannot be mapped using method based input-output equivalence

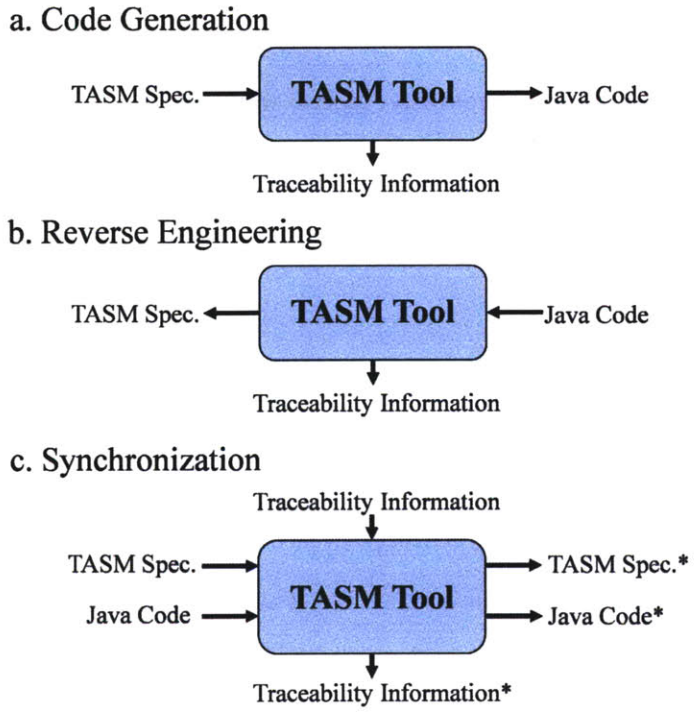

Figure 1-3: Code Generation, Reverse Engineering, and Synchronization via a tool

section of the model, and similar comments could be added to the code to indicate how it was generated. A more sophisticated tool might use the mapping explicitly to allow clicking on a section of Java to jump to the section of TASM it implements and vice versa. The traceability information could even be printed as a form of traceability documentation.

Instead of implementing all of these possibilities, we ground the approach by using the transformation framework in code generation. Generating Java code from a TASM specification, involves the sequence of transformations from TASM to TASM SDG to JSDG to Java. Chapter 4 describes this process as well as reverse engineering and synchronization.

1.6 Contributions

The approach described is innovative in two ways:

1. A new SDG structure is defined that describes the semantics of the TASM language.

2. In order to build the dependency subgraph of the TASM SDG, a novel algorithm based on geometry calculations in variable space is used. The algorithm is generally applicable to ASMs, and finds all the abstract state transitions of an ASM with bounded variables. This in contrast to many simulation based approaches which monitor traces of execution and can only provide probabilistic coverage of the transitions [20]. The algorithm is also shown to provably terminate.

1.7 Thesis Layout

The remainder of this thesis is organized as follows:

Chapter 2 discusses the languages used in this thesis. First, basic ASM theory is discussed. Then TASM and Java are presented with a summary on the subset of each language used.

Chapter 3 presents the two innovations: The syntax of the TASM SDG and the method for extracting the full set of transitions from an ASM. The algorithm is extended to help build the TASM SDG.

Chapter 4 presents Triple Graph Grammars (TGG) and what subset of the transformation information must be saved to preserve traceability information. The transformation framework is described in terms of its application in code generation, reverse engineering, and synchronization.

Chapter 5 presents a case study involving code generation from an electronic throttle controller. It starts by presenting the state of the current implementation. Then follows with a summary of the case study and its results.

Chapter 6 finishes the thesis with a summary of the work and possible future work.

Chapter 2

Languages

In order to appreciate the difficulty of mapping between two languages, it is important to understand that languages can express different notions of computing. This chapter presents the syntax and semantics of the Abstract State Machine (ASM), Timed Abstract State Machine (TASM), and Java language. Since TASM is based on Abstract State Machines (ASMs) and the research is more generally applicable to ASMs, the chapter starts with a description of the ASM language. Section 2.2 describes TASM and the subset that can be used for traceability information. Section 2.3 describes Java and its supported subset.

Sections 2.1 and 2.2 paraphrase Chapter 4 of [33], which presents the TASM language in detail.

2.1 ASM

2.1.1 Origins

Abstract State Machines (ASMs) were originally posited by Yuri Gurevich in the 1980's as a computing machine more generally applicable than a Turing Machine. He specifically wanted to create a machine capable of simulating "any algorithm, never mind how abstract"at the level of abstraction of the algorithm [23]. This goal differentiated his computing machine from Turing's in two important ways: First, his

machine would be able to represent any algorithm. And, second it would be able to represent the algorithm naturally.

Turing's original thesis can be paraphrased as "every effectively calculable function is computable by a Turing machine." Where:

- *machine* — is used here to describe a conceptualized mechanical or electrical device. For example, Turing envisioned a machine with a head that could read from and write to cells along an infinite length of tape. The machine is conceptual in that it is infeasible to actually produce an infinite length of tape.

- *effectively calculable* — is an intuitive idea that neither Church nor Turning formally defined. Something is 'effectively calculable' if it is capable of being calculated by anything or anyone.

- *computable* — capable of being produced by a machine.

But, an even stronger thesis proved to be true: every algorithm can be simulated by a Turing machine [35]. The distinction between these theses lies in the difference between a function and an algorithm. As used in Turing's thesis, a *function* refers to the mathematical notion of a function with input and output. Whereas an *algorithm* is a method for solving a problem using a finite sequence of instructions. For example, the greatest common divisor (GCD) of two numbers is a function, but there are many ways to find the GCD, including a Matrix method and several variations of the Euclidean algorithm. Savage showed that a Turing machine can not only compute any function, it can compute the function via any algorithm. Gurevich wanted his machine to have the same expressive power.

However, expressing any arbitrary algorithm as a Turing machine would require the translation of that algorithm to a table of operations involving the manipulation of 1's and 0's on a tape. The purpose of the resulting Turing Machine would be difficult to decipher. In contrast, Gurevich wanted a machine that could naturally simulate any algorithm.

The solution devised was the Abstract State Machine (ASM).

2.1.2 Application

The ability of an Abstract State Machine (ASM) to model any abstraction of an algorithm naturally and its basis in the formalisms of computing machinery make it an ideal candidate for system design. Practically, ASMs have been used successfully on a wide range of applications, ranging from hardware-software systems to high level system design [9, 10]. It has been used to specify APIs [7] and provide formal semantics for validating UML models [37]. It has even been used to generate C++ code for train scheduling [8].

Part of the success of ASMs can be attributed to the simplicity of the language used to specify its behavior. The original language and most variants have semantics that closely reflect the functional behavior of the machine. Coupled with a small grammar that reflects many existing implementation languages, there is enough evidence to believe that ASMs provide a literate specification language, this is, a language that is understandable and usable without extensive mathematical training [14]. The close relation between the semantics and functional behavior, mean properties of ASMs also carry over to the language: including the ability to specify an algorithm at any level of abstraction.

The well-defined formal semantics of the ASM language also make it suitable for formal verification. ASM specifications are independent of a specific verification method and can be verified either through manual proofs or through automated tools [45]. Furthermore, an integral part of ASM theory, is *refinement*, or the process of gradually adding details to a system design, which makes ASMs applicable at various levels of abstraction and allows for incremental design improvements.

2.1.3 Syntax & Semantics

2.1.3.1 Abstract State

An Abstract State Machine, like all state machines, operates by transitioning between states, where a state is a "full and instantaneous description of an algorithm" [23]. What makes an ASM unique is its use of structures as state.

In mathematics, a *structure* is a set, along with relations and functions over that set[1] For example, a graph can be considered a structure consisting of a set of vertices, with binary function *edge* that takes two vertices and returns *true* or $false$ if an edge connects them. But, a mathematical structure need not correspond to a physical realization. The set of all reals and the binary function '+' can also be considered a structure. The expressiveness of a structure stems from the versatility of functions. A function can be *static* or *dynamic*. A function whose value depends on the state, or can be assigned, is called dynamic (i.e. a variable or array). A function whose value depends only on its arguments is static, (i.e. *sine*, *cosine*, *abs*). A function that has 0-arity, takes no arguments, is called a *nullary-function*. A static nullary function is a *constant* (i.e. π, e, *true*, $false$). A function that returns *true*, $false$, or $undef$ is called *relational* (i.e. &, =).

This structure based state is a generalization of the state, as used in a Finite State Machine (FSM). In order to distinguish the two, the generalized state is called an *abstract state*. To understand the difference, take for example the game of tic-tac-toe. Assume whatever algorithm we wish to describe is completely dependent on the state of the board, that is the location of the X's and O's on a 3x3 grid. In a FSM, we might identify each state with a label such as: "one X, upper left, no O's". Even though one might understand the configuration described, for the purposes of specifying an FSM, a systematic numbering scheme for all 3^9 states would have worked just as well. In contrast, an ASM might represent the same state as "$cell(0,0) = X$ & $cell(1,0) = empty$ & ... & $cell(2,2) = empty$," where *cell* is a binary function that takes grid coordinates and returns the contents of the cell, X, O, or *empty*[2].

The use of functions (note that equality and boolean operators are also functions) in specifying state allow an ASM to operate on states in unique ways. For example,

[1]Formally, a structure consists of a triple: A base set or domain, which is an arbitrary non-empty set containing, for example, all the numbers, function names, and symbols to be used in representing the structure; Signatures, which associate a function name with an arity (the number of arguments the function takes); And an interpretation, which associates with each j-ary function signature, a mapping "f: (base set)$^j \rightarrow$ base set.

[2]The *vocabulary* (the superset of the base sets of all the states) of this ASM, would be a union of the relational boolean operators, equality, and $\{0, 1, 2, grid, X, O, empty\}$.

we can change the abstraction level of the algorithm by: changing the vocabulary used to describe states (use functions *row* and *column* instead of *grid*), or by using the functions in alternate expressions (instead of states corresponding to board configurations, the states could correspond to 'X winning', 'O winning', 'no winners', and 'undecided'). Additionally, state transitions no longer need to be explicitly named as moving from state to another. Instead, they can be specified implicitly by changing the value of a function used in identifying a state. In the above example, setting $cell(1, 0)$ to O would result in a state transition. Incidently, ASMs were originally called *evolving algebras* because of this ability to transition between states through affecting a function.

In the original thesis, abstract states were specifically first order structures, but higher-order structures could have also been used. Implementations have also diversified the possible state representations to use typed variables.

2.1.3.2 Behavior

The behavior of an ASM can be subdivided into a sequence of computing steps which modify the *global state*. The global state refers to the abstract state of the ASM at any time. It is called 'global' to emphasize the notion that this state must encompass all the information (not withstanding non-deterministic constructs such as 'choose') needed to decide the next state transition. Namely, the global state can include the abstract state of the algorithm, its environment, and the states of other agents/ASMs.

In each *step* of the ASM, the current global state is checked and accordingly, a group of atomic updates is applied to yield the next global state. An *update set* is the term used to describe the set of atomic updates that are associated with a single step. A *run* of an ASM is a sequence of global states, starting with the initial state, visited by an ASM as a result of executing a sequence of steps. Specifically, a run can be calculated by sequentially applying the update sets to the global state. In general, there is no formal requirement concerning what triggers the execution of a step or the rate at which the steps or executed. Depending on the level of abstraction being modeled, a step could correspond to a clock cycle, a machine operation, or a

statement execution in a high level programming language.

Concerning interactions with the environment or other agents, an ASM interacts with the world around it through either directly updating the global state, or indirectly through executing an *external function* [27, 23].

For the remainder of this thesis, we will call a 'global state' and 'abstract state' a 'state' when the intended meaning is clear.

2.1.3.3 Specification

The program an Abstract State Machine executes is commonly referred to as a *specification*. The use of the term *specification* stems from the foundations of ASMs as computing machinery, where it specifies how such a computing machine would need to be configured in order to simulate a particular algorithm. When applied, i.e. used in model-based software engineering, an ASM specification is also referred to as a *model*. We will use the terms interchangeably.

In general, a *specification* is a document that results from the process of writing down a system design. It may have its own structure (chapters and sections) and be composed of several languages, where a *language* is defined by *syntax* and *semantics*. In relation to ASMs, a specification is usually written in one language, where the syntax and semantics of the language have been designed to reflect the behavior of the ASM. As a result, the terminology loses some distinction: The structure of the specification is incorporated in the syntax of the language.

An ASM specification fundamentally consists of an initial state and a finite set of rules. Since the initial state is just an ASM's first abstract state, its specification depends on the type of structure and syntax used by the implementation language; We defer those details until we discuss a specific language, the TASM language, in the next section. The rules, however, are usually written in *canonical form* or *block form*. For an ASM that contains n rules, a machine in block form has the following structure:

$$
\begin{aligned}
R_1 &\equiv if\ G_1\ then\ E_1 \\
R_2 &\equiv if\ G_2\ then\ E_2 \\
&\vdots \\
R_n &\equiv if\ G_n\ then\ E_n
\end{aligned}
\tag{4.1}
$$

A *rule*, R_i, consists of a single guard-effect pair[3], which are usually written as an if-statement. The *guard*, G_i, is a boolean expression. The effect, E_i, can be zero or more assignment or update statements. When a guard evaluates to true, we say the containing rule is *enabled*, and *apply* the *effect* of the rule to the global state. For brevity, we also use the word enabled to lazily describe a guard or effect of an enabled rule.

Relative to the specification, a single *step* of the ASM has three parts: First, all the guards are evaluated simultaneously in the current state. Second, all enabled effects are evaluated in the current state to yield one *update set*. If there are conflicting updates, that is, differing assignments to the same term, the update set is empty. Finally, the update set is applied atomically to the current state to yield the next state. This process repeats until the step where no rules are enabled, or the update set is empty (all enabled rules have no effect statements). The consequence of using atomic actions is that guards cannot have side-effects that modify the state.

2.2 TASM

The TASM language is a modeling language based on ASM specifications, and one of the two languages between which we seek to provide a bidirectional mapping. This section starts by introducing the behavior of a Timed Abstract State Machine, then

[3]In [Sequential ASM Thesis], an 'update rule' or 'rule' is a statement, that when executed, alters the state. The parallel composition of rules (effect), or the presence of a precondition (guard-effect pair) are also rules. To help distinguish between rules, we adopt the naming convention used in [33] and refer only to a single guard-effect pair as a rule.

summarizes the syntax and semantics of the TASM language used to specify that behavior.

2.2.1 Behavior

The Timed Abstract State Machine (TASM) is an extension of the ASM, with the ability to simulate time and resource consumption.

2.2.1.1 Time

Time is introduced in TASM using the durative action paradigm. In the original ASM each step is instantaneous. In contrast, each step of a TASM can have an associated duration. The evaluation of the guards to determine which rule is enabled is instantaneous, and so it the creation of the update set, but the update set won't be applied until the duration has elapsed. During which time, the TASM is considered busy and can do no other work. This delay simulates an implementation where the actual effects take time to create and apply. The duration of one run of a TASM is therefore the sum of the duration of all of its steps.

2.2.1.2 Resource

Resources are primary used as a simulation and verification device to ensure a machine can execute with a fixed quantity of real-world resources. As a result, a resource does not interact with the rules of the ASM: A resource cannot be used to identify or compute a state. Additionally, resources cannot be destroyed or created, so the total amount of any resource is fixed.

A *resource* is a global quantity of finite size. The quantity is considered 'global' because it can be used by all TASMs operating in the same environment, and 'finite' because there is a limited amount available. A resource must also be reusable. If a quantity of resource is considered *used* when it is deducted from the global quantity, a *reusable* resource is one where any used quantity can also be added back to the global quantity. Memory and communication bandwidth are examples of a resource.

But, fuel, for example, which can be consumed, is not (fuel can be modeled in TASM by using a global variable as a decrement counter).

Each step of a TASM can use a fixed quantity of each resource for the duration of its execution. The quantity used is deducted from the global quantity for the duration of the step, and returned to the global quantity upon completion of the step. The evaluation of guards is assumed use no resources. If the global quantity of a resource ever falls below zero, the behavior is undefined.

2.2.1.3 Environment

Beyond the additions of time and resource, the behavior of a TASM also varies in the manner in which it interacts with the environment and other agents. There is no notion of an external function (a function that is defined outside of the ASM). Instead, a TASM interacts externally by *monitoring* (reading) and *controlling* (writing) values defined in a global scope, called the *environment.* Additional TASMs defined in the same environment can also monitor and control those values, thus taking the place of external functions. The advantage of this approach is that the behavior of all the agents and environment can be captured in the same formalism for simulation and verification. The disadvantage is that it can blur abstraction boundaries between agents. For example, when modeling a 2 agent system, consisting of a thermostat and air conditioning (AC) unit, instead of calling an external function $turnOnAC()$, the TASM modeling the thermostat might need to explicitly set the values for the AC's power and fan speed.

The TASM language mitigates this by introducing auxiliary machines and hierarchical composition, as used in XASM, which compartmentalizes code for reuse.

2.2.2 Language Introduction

The language used to specify the behavior of a TASM is called the *TASM language.* The TASM language was designed as the textual input language of the *TASM toolset*, an IDE with bundled simulation and state-of-the-art verification tools important for

model-based software engineering. The language itself was designed to be a formal and literate modeling language, making it suitable for simulation and to serve as a human-readable design specification. A document written in the TASM language is called a *TASM specification*, or specification, for short.

2.2.2.1 Features

At its heart, the TASM language is an extension of the Abstract State Machine (ASM) specification, with facilities to specify time and resource consumption. The subset of the ASM specification included in the TASM language is the same as explained in [45], which includes conditional statements and assignments, but excludes the *forall*, *choose*, and *import* construct.

The language also allows the expression of a TASM in component machines called *main machines*, *sub machines*, and *function machines*. Through the *parallel composition* of main machines, the language allows for more than one TASM to interact in the same environment. And through the *hierarchial composition* of machines, it facilitates readability and code reuse. To emphasize this terminology, the term *machine* will be used to refer to the syntactic components that compose a TASM.

To further facilitate readability, the language also allows a plain-language name and description to be paired with many language constructs.

2.2.2.2 Implementation and Limitations

In its current revision (1.0.0.74a), the TASM toolset is implemented in Java and provides a graphical form-interface to create the constructs of the language. The close tie the language has with its implementation is reflected in the syntax of the language, and makes manual editing difficult. Most evidently, a specification written in the language is stored in one *.tasm* file. Additionally, specific whitespace characters are used to delimit some constructs.

The language has also inherited traits from its implementation. Since it is interpreted during simulation, the data types in TASM reflect those primitive data types available in the Java language.

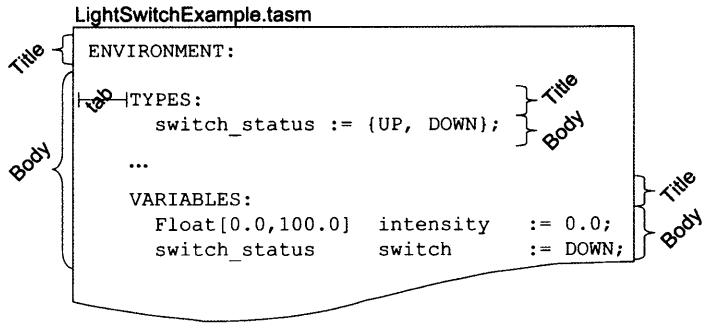

Figure 2-1: title-body nesting structure

2.2.2.3 Structure and Whitespace

The syntax of the TASM language creates a structure similar to that of an outline. The structure is composed of a recursive nesting of *title-body* pairs [Figure 2-1]. This format facilitates its use as a human-readable specification.

The *title* denotes the contents of the body that follows it. Syntactically, titles are a language defined sequence of characters that act as delimiters. A title can consist of capital letters, space, dash, and underscore, ends with a semicolon, and resides on its own line. When a title is nested in a body, it is indented by one tab character for each nesting.

The *body* can consist of nothing, a nested set of title-body pairs, or non-structural expressions. Syntactically, a body can contain any whitespace character and does not need to be indented to correspond to its title.

As mentioned before, the toolset uses a form-interface for creating a TASM specification, so almost all of the structural syntax used in the TASM language is generated automatically.

2.2.3 Overview

The remainder of this section presents a summary of the syntax and semantics of the TASM language. The next section (2.2.4), on syntax, presents the basic building blocks of the language: data types, variables, operators, the environment, rules,

machines, and specification. The subsequent section (2.2.5) presents the semantics of parallel and hierarchial composition. Those already familiar with the syntax of the language will find that it is sufficient to skim section 2.2.4 and just read the semantics.

A description of the *resource* modeling and *channel* language features is omitted, because they will not be preserved in the bi-directional mapping we seek to create. Other details of the language including whitespace usage, whether a named construct requires a functional name or plain-language name, and whether certain titles are required are also omitted for clarity. For a complete set of concrete syntax and semantics, please refer to Appendix A for the language reference.

2.2.3.1 Syntax Conventions

The syntax will be presented in two ways. It will be formally described using discrete mathematics (i.e. set theory) in an *abstract syntax*, and examples will be given in the *concrete syntax* implemented by the language's compiler. The abstract syntax presented will be equivalent to that presented in [33], but will vary slightly in vocabulary and organization. Namely, a set containing only elements, S, will be written in script, $\mathcal{S}$.

2.2.3.2 Light Switch Example

We will use a simple example to illustrate the syntax and semantics of the TASM language. The basic example will involve a light switch that controls a light bulb. When the switch is turned on, or is in the "UP" state, the bulb will increase in brightness from 0 to a maximum intensity of 100, at a rate of $+1$ intensity per second. At any time the switch can be turned off, or put in its "DOWN" state, which forces the light bulb to immediately go dark. Figure 2-2 illustrates this example as a *finite state machine* (FSM). Figure 2-3 formalizes the FSM as a Moore machine, where the arcs represent state transitions and are labeled with the triggering events and the states have associated entry actions.

The Moore machine formulation provides some insight into how TASMs interact with the environment. A Moore machine uses *entry actions*, which are executed upon

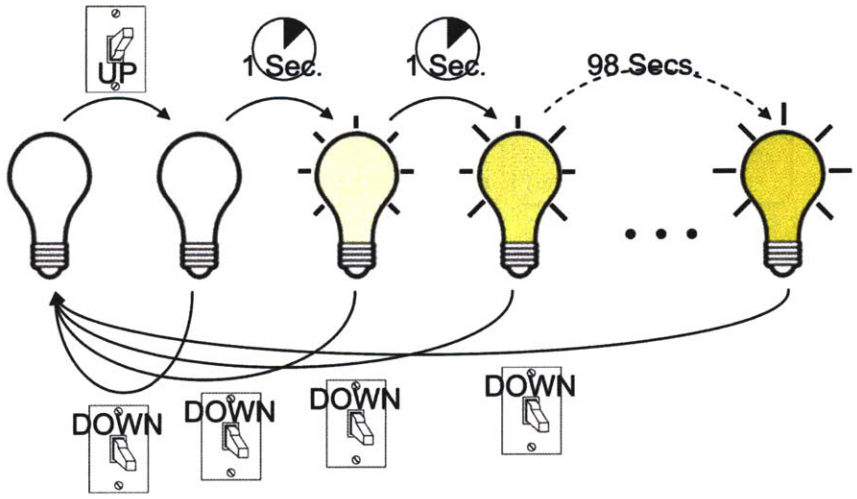

Figure 2-2: Light Switch Example

entering a state, to influence the environment and uses *events* to react to the environment. However, a TASM can only interact with the environment by monitoring and controlling variables. As a result, entry actions and events must be represented by variables. In this case, the `intensity()` entry action can be replaced by a variable, `intensity`, which models the intensity of the light. The `UP` and `DOWN` events can be replaced by a variable `status`, that models the status of the light switch.

To formulate this example, at its current level of abstraction in a TASM specification, two variables and two main machines (a main machine is a syntactic construct that specifies the behavior of one TASM) are needed. One TASM simulates control of the light's intensity by monitoring `status` and controlling `intensity`. The other simulates the environment by controlling `status`. The complete TASM specification that simulates this example are split across listings 2.5 and 2.6. The same specification will be presented in sections as language features are presented.

These examples extend upon those originally presented in [33] by replacing the binary on-off states with varying degrees of intensity. This modification was made to emphasize certain language features important to this thesis. The original examples and paper should be referred to for a more exhaustive presentation of the TASM language.

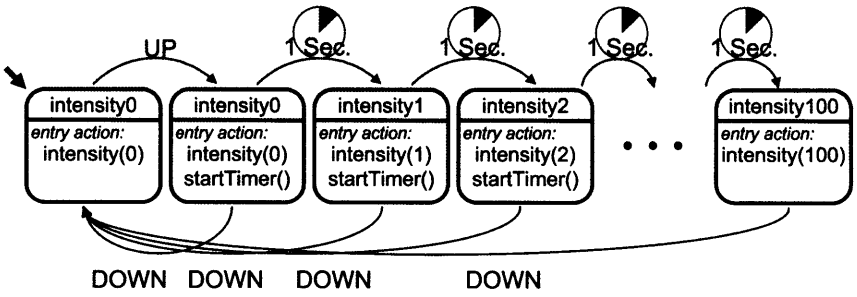

Figure 2-3: Moore FSM of Light Switch Example

2.2.4 Syntax

2.2.4.1 Data Types

The TASM language uses a simple type system: There are no data structures or arrays, and all variables are strongly typed. The language provides 3 primitive types and also allows the user to define their own type. Formally, the *Type Universe*, TU, in TASM is:

$$TU = \{Boolean, Integer, Real\} \cup \mathcal{UDN}$$

,

Where:

- *Boolean* names the value universe, $BVU = \{True, False\}$
- *Integer* names the value universe, $NVU = \{\ldots, -1, 0, 1, \ldots\}$
- *Real* names the value universe, $RVU = \mathbb{R}$
- $\mathcal{UDN}$ is the set of user-defined types names, UDN, each of which uniquely names a value universe of user-defined values.

The user-defined type is a set, where the user defines the name of the set and the elements of the set. It is similar to an enumeration, but the elements of the set are not assigned numbers and are not comparable.

Concrete Syntax In concrete syntax:

abstract syntax type name	**concrete syntax type name**	**concrete syntax example values (literals)**
Boolean	`Boolean`	`True, False`
Integer	`Integer`	`-1, 0, 1, 100`
Real	`Float`	`-1.0, 0.0, 1e2`
UDN	`<TASMName>`	`<TASMName>, UP, DOWN`

Numerical Accuracy If used as a descriptive modeling language, the precision, minimum, and maximum values of the numerical types (Integer & Real) can be written arbitrary. However, when used in simulation or for verification, the types are given concrete interpretations:

concrete syntax name	**min.**	**max.**	**precision**
`Integer`	-2,147,483,648	+2,147,483,647	1
`Float`	-1.8 e 308	+1.8 e 308	64 bit, IEEE 754 Floating-Point

During simulation, the parsing or computation of `Integer` literals that are non-representable numbers cause the simulation to halt and report and error. `Float` literals are rounded to their closest representable form, with warnings if rounding to $\pm$`inf` and halt with errors if resulting in `NaN`.

Type Conversion In its current revision the TASM language does not support widening (promotion) or narrowing (casting) conversions.

2.2.4.2 Variables

Formally, a variable, V, is the tuple:

$$V = \langle type, vn, val \rangle$$

Where:

- *type* is an element of the Type Universe, *TU*
- *vn* is the *variable name*
- *val* is the value of the variable, an element of the named set, *type*.

Variables do not have a default initial value, so they must be initialized before use. The language syntax enforces this by requiring all variables to be defined (initialized upon declaration). The only time a variable can be declared is in the signature of a function machine [section 2.2.4.8], where its initial value will be provided when the function machine is called.

Concrete Syntax Besides declaration and initialization, variables are used solely in rules. The expressions in which variables are used:

name	**concrete syntax**	**examples**
declaration	`<TypeName> <VarName>;`	`Integer[0,100] i;`
definition (decl. & init.)	`<TypeName> <VarName> := <Constant>;`	`Integer[0,100] i := 0;`
arithmetic exp. logical exp.	See Appendix A	`TURN_ON(switch)` `i <= 100 and switch = UP`
initialization	`<VarName> := <Constant>;`	`i := 1;`
effect exp.	`<VarName> := <(Arith|Logic)Expr>;`	`i := i + 1`

When declaring a variable, the type name can be written in two ways. If written alone, a variable has the same range as its type. If written using the bracket notation, as shown in the above table, the variable range is restricted to the inclusive interval. Variables of the same type, but restricted to different ranges, are still considered to be of the same type, and can be used in the same expressions.

Scope In general, the scope of variables is constrained by the construct in which they are declared. When declared in the environment construct, the variable is said

to have global scope, and is visible to all main and sub machines. However, individual ASMs must identify variables by name that they with to read from or write to, prior to use. When declared internal to a main machine [section 2.2.4.7], it is only useable by rules declared within that machine. Similarly, when declared as an input or output variable of a function machine, it can only be used by rules declared in that function machine. Subsequent abstract syntax will formalize variable use.

2.2.4.3 Operators

The TASM language supports a limited set of operators. Since there is no type conversion, expressions using these operators must use a uniform type.

- `Integer` and `Float`
  - the assignment operator (:=)
  - 4 binary arithmetic operators: addition (+), subtraction (−), multiplication (∗), division (/)
  - 6 binary comparison operators: equal (=), not-equal (!=), greater-than (>), greater-than-or-equal (>=), less-than (<), less-than-or-equal (>=)
- `Boolean`
  - the assignment operator (:=)
  - 1 unary comparison operator: logical-not (`not`)
  - 4 binary comparison operators: equal (=), not-equal (!=), logical-and (`and`), logical-or (`and`)

The operators can be used in compound expressions, where the order of operations is defined in Appendix A, but parenthesis should be used for clarity. Expressions that evaluate to an `Integer` or `Float` are referred to as *arithmetic expressions.* Expressions that evaluate to a `Boolean` are referred to as *logical expressions.*

2.2.4.4 Time

In the TASM language, the passage of time is only associated with the execution of a rule. Syntactically, that duration is specified by annotating each rule with a duration interval, within which the machine must complete the execution of that rule. Formally, the time annotation takes the following interval form:

$$TI = \langle tmin, tmax \rangle$$

Where:

- $tmin$ is the inclusive minimum duration, $tmin \in \{0, 1, 2, ..., \infty\}$
- $tmax$ is the inclusive maximum duration, $tmin \in \{0, 1, 2, ..., \infty\}$
- $tmin \leq tmax$

During rule execution, the exact duration of the rule, t, is sampled from the interval each time the rule is enabled, such that $tmin \leq t \leq tmax$. All time in the TASM language is represented by unit-less, non-negative integer values.

Concrete Syntax In concrete syntax the annotations can take several forms:

concrete syntax	abstract syntax	interpretation
`t := [<tmin>,<tmax>];`	$TI = \langle tmin, tmax \rangle$	execution duration sampled from interval
`t := <t>;`	$TI = \langle t, t \rangle$	execution duration of exactly time t
`t := 0;`	$TI = \langle 0, 0 \rangle$	instantaneous execution
In the absence of a time annotation: the interpretation varies based on whether the rule is used in a hierarchial composition (if it uses or is used by another rule). If used in a hierarchical composition, see section 2.2.5.2. If not, $TI = \langle 0, 0 \rangle$.		
`<no time annotation>`	$TI = \perp$	see above
When multiple machines are executing in parallel, the following annotation indicates the update set of this rule will be applied the next time an update set is applied by anther machine. If no other machines exist, the rule is executed instantaneously (see section 2.2.4.6).		
`t := next;`	$TI = \langle \tau, \tau \rangle$	see above

In the original ASM formulation, multiple rules could be enabled and their results merged to form a single step. To prevent any confusion over the merging of time annotations, the TASM language requires that only one rule be enabled per step per machine.

The initial state of the environment is assumed to occur at global time, $gt = 0$. While not explicitly declared as a variable, the global progression of time can be retrieved by using the reserved word `now` as a variable. When used in the guard or effect expressions of a rule, it returns the value of gt at the outset of a rule.

2.2.4.5 Environment

Each TASM specification is required to have exactly one *environment*. The environment is a language construct that acts as a global scope for the TASM specification: Any language element that can be used by all machines must be declared in the environment. Formally, the environment is a tuple:

$$E = \langle TU, EV, \mathcal{RS}, \mathcal{CH} \rangle$$

Where:

- TU is the *Type Universe*
- EV are the *Environment Variables*, a non-empty set of typed variables
- $\mathcal{RS}$ is a possibly empty set of *Resources*, $RS = \langle rn, ramin, ramax \rangle$. ($rn$ is a resource name, $ramin$ and $ramax$ are fixed, inclusive bounds representing the total amount of resource rn available at any given time.)
- $\mathcal{CH}$ is a possibly empty set of *Channels*, CH

Concrete Syntax In concrete syntax, the environment construct is where the user can add to the type universe by defining user-defined types, define variables, declare resources, and declare channels that will be used by the machines operating within its scope.

Listing 2.1 Light switch example, environment

```
1 ENVIRONMENT:
2     TYPES:
3       switch_status := {UP, DOWN};
4     RESOURCES:
5     CHANNELS:
6     VARIABLES:
7       Integer[0,100]  intensity   := 0;   // 0=OFF, 100=FULL ON
8       switch_status   switch      := DOWN;
```

Light Switch Example Listing 2.1 shows the environment for the light switch example. Line 1 contains the environment title and lines 2, 4, 5, and 6 contain the titles of the environment's components. Line 3 declares the only user-defined type, `switch_status`. Line 8 uses that type to define a variable named `switch`. Line 7 defines a variable named `intensity` that has a bounded integer range from 0-100. There are no resources or channels.

2.2.4.6 Rule

A rule, as described in Section 2.1.3.3, is the basic building block of an ASM. A rule, R, in the TASM language is extended to have a name, a time annotation, and resource annotations:

$$R = \langle n, TI, RI, G, E \rangle$$

Where:

- n is the plain-language name of the rule
- G is the guard, a logical expression
- E is the effect, a set of effect statements.
- TI is the time annotation.
- $\mathcal{RI}$ is a set of resource usage annotations, $RI = \langle rn, raumin, raumax \rangle$. ($rn$ is a resource name, $raumin$ and $raumax$ are inclusive bounds of the amount of resource rn this rule will use during its duration.)

Concrete Syntax In concrete syntax a rule takes the form:

```
<rule name>:
{
    <time annotation>;
    <resource annotation(s)>;
    if <logical expression> then
        <effect expression(s)>;
}
```

Else & Skip The keyword `else` is syntactic sugar and can take the place of an `if` statement. It creates a rule that is executed when no other guards in the same machine are enabled. That is, given a set of rules with guards: $G_1, G_2, ..., G_n$, it creates a rule with the guard *not* (G_1 *or* G_1 *or ... or* G_n).

When a rule has no effect statement, the keyword `skip` must be used in its place.

The keywords `else` and `skip` are commonly used together when there are machines executing in parallel. Since a machine that has no enabled rules or no effect statements (a.k.a. no update set) is said to have terminated, the `else-skip` pairing creates a rule to keep the enclosing machine 'alive' while waiting for another parallel machine to affect the environment. Furthermore, to keep the enclosing machine reactive, the `t:=next` time annotation is usually paired with `skip` to ensure the machine checks the environment as soon as another machine modifies it. Any other time annotation, other than `t:=0` or `t:=[0,0]` is also acceptable.

```
<rule name>:
{
    t := next;
    <resource annotation(s)>;
    else then
        skip;
}
```

One Rule Per Step Unlike an ASM, where any number of rules can be enabled and executed simultaneously, each TASM has the restriction that only one rule can be executed per step. This limitation arises from the presence of the time annotation, for which there is no straightforward composition when two rules of differing time annotations are enabled at the same time. When main machines are composed using parallel composition, each main machine can execute one rule per step.

Listing 2.2 Light switch example, rule

```
1 R1: Turn On
2   {
3     t := 1;
4     if intensity < 100 and switch = UP then
5        intensity := intensity + 1;
6   }
```

Light Switch Example Listing 2.2 shows one of the rules from the light switch example. Line 1 contains the name of the rule. Line 3 contains the time annotation,

indicating each execution of this rule should take exactly one unit of time. Line 4 contains the rule's guard or logical expression. Semantically, when line 4 evaluates to `True` (is enabled), the single effect expression on line 5 will be executed yielding a new value for the variable `intensity`. One second after the rule is enabled, the value of `intensity` will be changed.

2.2.4.7 Main Machine

To enable specification of multiple parallel activities in a system, the TASM language allows parallel composition of multiple abstract state machines. Parallel composition is enabled through the definition of multiple top-level machines, called *main* machines, analogous to multiple agents in [9]. A main machine fundamentally consists of a set of variables and rules. The variables are differentiated based on scope and read-write permissions. Formally:

$$MASM = \langle n, d, MV, CV, IV, \mathcal{R} \rangle$$

Where:

- n is the machine name
- d is a plain-language description
- MV is the set of *Monitored Variables* = $\{mv \mid mv \in EV$ and mv is read-only in $R\}$
- CV is the set of *Controlled Variables* = $\{cv \mid cv \in EV$ and cv is read-write in $R\}$
- IV is the set of *Internal Variables* = $\{iv \mid iv$ is a typed variable and $iv \notin EV$ and iv is read-write in $R\}$.
- $\mathcal{R}$ is the set of *Rules*, R.

Concrete Semantics In concrete semantics a main machine takes the form:

```
MAIN MACHINE:
    NAME:
      <machine name>
    DESCRIPTION:
      <plain-language description>
    MONITORED VARIABLES:
      <names of monitored variables>;
    CONTROLLED VARIABLES:
      <names of controlled variables>;
    INTERNAL VARIABLES:
      <define internal variables>;
    RULES:
      <rules>
```

Listing 2.3 Light switch example, machine

```
1  MAIN MACHINE:
2      NAME:
3        LIGHT_CONTROL
4      DESCRIPTION:
5      MONITORED VARIABLES:
6        switch;
7      CONTROLLED VARIABLES:
8        intensity;
9      INTERNAL VARIABLES:
10     RULES:
11       R1: Turn On
12       {
13         t := 1;
14         if intensity < 100 and switch = UP then
15            intensity := intensity + 1;
16       }
17       R2: Turn Off
18       {
19         if switch = DOWN then
20            intensity := 0;
21       }
22       R3: Wait
23       {
24         t:=next;
25         if intensity >= 100 and switch = UP then
26           skip;
27       }
```

Light Switch Example Listing 2.3 shows one of the main machines from the light switch example, which runs in the environment found in listing 2.1. The main machine models the lightbulb's physical controller by controlling the integer variable `intensity`. Lines 10-16 define rule 1: When the switch is up, the intensity will increase at a rate of 1 every 1 second. Lines 16-21 define rule 2, which sets the intensity of the light to 0, when the switch is down. Lines 22-27 define rule 3, which makes the light switch wait when the light bulb is at full intensity. Rule 3 could be replaced equivalently with an `else` rule, but the guard here is explicitly written out for clarity.

Even though the variable `intensity` has been declared to have an integer range from 0-100, inclusive, the guards on line 14 and 25 still check the full range of integer values. This is because the type declaration assists in verification and error checking, but does not enforce those bounds.

2.2.4.8 Auxiliary Machines

Auxiliary machines are a purely syntactic construct used to ease reuse and structuring of specifications. An auxiliary machine encapsulates rules so they can be easily nested in the definition of other rules. This manner of reuse is called *hierarchical composition.* When an auxiliary machine is used in a rule, it is referred to as a *call.* The calling rule is referred to as the *parent*, and the rules defined in the auxiliary machine are considered *children.*

Unlike main machines, auxiliary machines are not stand-alone, and must be used in the rule of a main machine to be executed. Like a main machine, auxiliary machines fundamentally consist of a set of variables and rules. There are two types of auxiliary machines: a *sub machine*, $SASM$, and a *function machine*, $FASM$.

Sub Machine

$$SASM = \langle n, d, MV, CV, \mathcal{R} \rangle$$

Where:

- n, d, MV, and CV are as defined before.

- $\mathcal{R}$ is a set of rules, R, that only operate on variables in MV or CV.

Sub machines can be used as a type of effect expression. The syntax for calling a sub machine is to use the machine's name followed by an empty pair of parenthesis. For example:

definition	**use**
SUB MACHINE: NAME: <machine name> DESCRIPTION: <plain-language description> MONITORED VARIABLES: <names of monitored variables>; CONTROLLED VARIABLES: <names of controlled variables>; RULES: <rules> // children rules	<rule name>: // parent rule { <time annotation> <resource annotation(s)> if <logical expression> then <effect expression(s)>; *<machine name>();* <effect expression(s)>; }

The semantics of hierarchial composition with a sub machine can be found in section 2.2.5.2.

Function Machine

$$FASM = \langle n, d, IV, OV, MV, R \rangle$$

Where:

- n, d, and MV, are as defined before.

- IV is a set of named inputs (ivn, it) where ivn is the input name, unique in IV and $it \in TU$ is its type.

- OV is a single pair (ovn, ot) specifying the output where ovn is the name of the output and $ot \in TU$ is its type.

- R is the set of rules with the same definition as previously stated, but with the restriction that it is read-write in IV and each rule must write to OV.

Usually a function only consists of inputs and outputs. The addition of the *monitored variables*, allows the function machine to monitor environment variables , without forcing the calling machine to also declare that it is monitoring the variable.

Function machines can be used as a type of value expression. Therefore, they can be called in the guard or effect of a rule. The syntax for calling a function machine is to use the machine's name followed by a pair of parenthesis. Within the parenthesis is a comma separated list of value expressions, whose evaluated values will be assigned to the input variables. The first value expression will be stored in the first declared input variable, the second value in the second declared input variable, and so on. When the function machine has finished execution, it will return the value of its output variable. The semantics of hierarchial composition with a function machine can be found in section 2.2.5.2.

definition	**use**
FUNCTION MACHINE: NAME: <machine name> DESCRIPTION: <plain-language description> INPUT VARIABLES: <names of input variables>; OUTPUT VARIABLES: <names of output variables>; MONITORED VARIABLES: <names of monitored variables>; RULES: <rules> // children rules	<rule name>: // parent rule { <time annotation> <resource annotation(s)> if *<machine name>(<args>)* then <effect expression(s)>; var:=*<machine name>(<args>);* <effect expression(s)>; }

Light Switch Example Listing 2.4 shows a trivial example of hierarchial composition, which is behaviorally equivalent to the main machine from listing 2.3. The main machine is composed of one sub and one function machine. The sub machine is responsible for turning on the light; and the function machine for turning it off.

The canonical `else-skip` pairing is used in both auxiliary machines to ensure their rules completely cover each machine's state space. This is not required, and is only done to make the set of rules in each machine complete. Namely, the else rule in the function machine could only be enabled if the function machine had an input "`switch=DOWN`". This is obviously impossible given that the parent rule, from which it is called, will only call this function if the switch is up.

2.2.4.9 Specification

A basic instance of a TASM specification only requires an environment and one main machine. However, a TASM specification, in general, consists of constructs described in previous sections: an environment, a set of main machines, a set of sub machines, and a set of function machines. Plus, two additional constructs that we will introduce in this section: a project, and a configuration.

Formally, the TASM language defines a TASM specification as a tuple:

$$TASMSPEC = \langle P, \mathcal{CF}, E, \mathcal{MASM}, \mathcal{SASM}, \mathcal{FASM} \rangle$$

Where:

- P is the *Project Header.*
- $\mathcal{CF}$ is a set of *Configurations*, CF.
- E is the *Environment.*
- $\mathcal{MASM}$ is the a set of *Main Machines*, $MASM$
- $\mathcal{SASM}$ is the a set of *Sub Machines*, $SASM$
- $\mathcal{FASM}$ is the a set of *Function Machines*, $FASM$

Project A project is a header identifying the contents of the specification. It consists of a unique ID, a plain-language name, and a plain-language description of the contents of the specification.

Listing 2.4 Light switch example 2 – hierarchial composition

```
MAIN MACHINE:
    NAME:
      LIGHT_CONTROL
    DESCRIPTION:
    MONITORED VARIABLES:
      switch;
    CONTROLLED VARIABLES:
      intensity;
    INTERNAL VARIABLES:
    RULES:
      R1: Turn On
      {
        if intensity < 100 and switch = UP then
           intensity := TURN_ON(switch,intensity); // uses function machine
      }
      R2: Turn Off
      {
        if switch = DOWN then
           TURN_OFF();                              // uses sub machine
      }
      R3: Wait
      {
        t:=next;
        if intensity >= 100 and switch = UP then
          skip;
      }
-------------------------------------------------------------------------------
FUNCTION MACHINE:                        |     SUB MACHINE:
    NAME:                                |          NAME:
      TURN_ON                            |            TURN_OFF
    DESCRIPTION:                         |          DESCRIPTION:
    INPUT VARIABLES:                     |          MONITORED VARIABLES:
      switch_status  switch;             |            switch;
      Integer[0,100] intensityIn;        |
    OUTPUT VARIABLES:                    |          CONTROLLED VARIABLES:
      Integer[0,100] intensityOut;       |            intensity;
    MONITORED VARIABLES:                 |
    RULES:                               |          RULES:
      R1: Turn On                        |            R1: Turn Off
      {                                  |            {
        if switch = UP then              |              if switch = DOWN then
           intensityOut :=               |                 intensity := 0;
                     intensityIn + 1;    |
      }                                  |            }
      R2: Else                           |            R2: Else
      {                                  |            {
        t := next;                       |              t := next;
        else then                        |              else then
           skip;                         |                 skip;
      }                                  |            }
```

Configuration A configuration is a language construct that facilitates easy reconfiguration of the specification to 1. model different initial conditions and 2. instantiate different sets of main machines to run in parallel. The execution of a TASM specification requires the selection of a specific configuration.

In order to accommodate this new construct, the definition of a main machine is expanded to include a constructor to initialize its internal variables. In the absence of a constructor, a *default constructor* is provided which provides no special initialization for the internal variables. Since the main machine definition must be initialized prior to use, it is also referred to as a *template main machine.*

A configuration is a tuple consisting of a plain-language name, plain-language description, a set of environment variable initializations, and main machine initializations. Formally:

$$CF = \langle n, d, \mathcal{VI}, \mathcal{MI} \rangle$$

Where:

- n is a plain-language name
- d is a plain-language description.
- $\mathcal{VI}$ is a set of variable initializations, $VI = (vn, val)$, such that vn names a variable in EV, and val is an element of its *type*.
- $\mathcal{MI}$ is a set of main machine initializations, which involve calling the constructor of a main machine with appropriate arguments.

Variables that occur in the Environment, but are not initialized by the configuration use the values they are defined with in the Environment. For more details concerning the concrete syntax of the main machine's constructor and how to call it, please refer to Appendix A.

2.2.4.10 Light Switch Example

Listings 2.5 and 2.6 shows the complete TASM specification for the light switch example, which requires the parallel composition of two main machines. One of the main machines (lines 31-60), which was presented in listing 2.3, models the controller for the light's intensity. The other main machine (lines 61-84), is new, and models one possible interaction with the light switch. After being switched up, it switches down after a random delay of 0-200 units. Both main machines now have an added constructor title, after which constructors can be defined. Even though a default constructor is sufficient for both main machines and is automatically provided, they are explicitly written in lines 41-42 and lines 70-71.

Lines 1-7 show the `Project` construct. Lines 10-19 show the single `Configuration` construct. Note the use of the constructor calls in line 18 and 19, which create an instance of each main machine.

Listing 2.5 Light switch example - complete TASM Specification, part 1

```
1  PROJECT:
2      ID:
3        7190e928-9081-4f9a-9241-1c8aaebc5a04
4      NAME:
5        Light Switch Example
6      DESCRIPTION:
7        This is the complete TASM specification for the light switch example.
8
9  CONFIGURATIONS:
10     CONFIGURATION:
11         NAME:
12           Parallel Composition
13         DESCRIPTION:
14           One main machine models the light intensity controller.
15           Another main machine models interactions with the light switch.
16         VARIABLE INITIALIZATIONS:
17         MACHINE INITIALIZATIONS:
18         light_control := new LIGHT_CONTROL();
19         light_switch_control = new LIGHT_SWITCH_CONTROL();
20
21 ENVIRONMENT:
22     TYPES:
23       switch_status := {UP, DOWN};
24     RESOURCES:
25     CHANNELS:
26     VARIABLES:
27       Integer[0,100]  intensity  := 0;      // 0=OFF, 100=FULL ON
28       switch_status   switch     := DOWN;
29
```

Listing 2.6 Light switch example - complete TASM Specification, part 2

```
MAIN MACHINES:
    MAIN MACHINE:
        NAME:
          LIGHT_CONTROL
        DESCRIPTION:
        MONITORED VARIABLES:
          switch;
        CONTROLLED VARIABLES:
          intensity;
        INTERNAL VARIABLES:
        CONSTRUCTOR:
          LIGHT_CONTROL(){
          }
        RULES:
          R1: Turn On
          {
            t := 1;
            if intensity < 100 and switch = UP then
               intensity := intensity + 1;
          }
          R2: Turn Off
          {
            if switch = DOWN then
               intensity := 0;
          }
          R3: Else
          {
            t:=next;
            else then
              skip;
          }
    MAIN MACHINE:
        NAME:
          LIGHT_SWITCH_CONTROL
        DESCRIPTION:
        MONITORED VARIABLES:
        CONTROLLED VARIABLES:
          switch;
        INTERNAL VARIABLES:
        CONSTRUCTOR:
          LIGHT_SWITCH_CONTROL(){
          }
        RULES:
          R1: Turn On
          {
            t := 0;
            if switch = DOWN then
               switch := UP;
          }
          R2: Turn Off
          {
            t := [0,200];
            if switch = UP then
               switch := DOWN;
          }
FUNCTION MACHINES:
SUB MACHINES:
```

2.2.5 Semantics

A *run* of a multi-agent system described by the TASM language consists of a sequence of *global states*. Where a global state is an instantaneous description of the state of the environment and all agents operating within that environment. In the case of a TASM, the global state must also incorporate time and resource usage. For brevity, we will refer to a global state as a state.

The state evolves by applying a sequence of update sets. Where each update set is created by one step of a TASM. For a TASM composed of a single main machine and no auxiliary machines, one step corresponds to the evaluation of a single rule, and therefore one update set.

This section explains the semantics of the TASM language starting by formalizing the definition of a state, update set, and run; then using those to describe hierarchial and parallel composition.

2.2.5.1 States & Update Sets

In this section we formalize the notion of a state, an update set, and a run.

State Formally, a *state*, S, is a set of variable-value pairs, (vn, val):

$$S = \{(vn_0, val_0), ..., (vn_n, val_n)\}$$

Where:

- vn is a variable name
- val is an element from the *type* of vn.
- A state must be *consistent*. For any pair $(vn_i, val_i) \in S$ and $(vn_j, val_j) \in S$, $vn_i \neq vn_j$. [4]

[4] We use a strong notion of consistency, which only allows each variable name to be used once in a

- Any variable name, vn_i, that occurs in a pair, must name a variable that is an element of the environment variables, EV, or an element of a [initialized] main machine's internal variables, IV.

To incorporate time into the state, a *timed state*, TS, is a pair consisting of a global-time value, gt, indicating the time at which the state occurred:

$$TS = (gt, S)$$

The initial timed state, TS_0, is the pair $(0, S_0)$. The initial global time is 0. And, the initial state, S_0, is the set of variable-value pairs created the definition of variables in the Environment (overwritten by any initializations in the chosen Configuration), and the definition of internal variables in initialized main machines.

Update Set In relation to the execution of a machine, the evaluation of the guards and effect expressions are instantaneous, so the update set is also generated instantaneously. The *time of generation*, tg, is the global time at which the update set is generated and corresponds to the start of a step. The *time of application*, ta, of an update set, is the global time when the update set is instantaneously applied to the global state and corresponds to the end of the step. The relation between tg and ta is: $ta = tg + t$, where t is the duration of the update set. Consequently, the ta of one step is equal to the tg of the following step. Between the global times, tg and ta, the machine is *busy*.

An update set, like a state, is also a set of variable-value pairs, (vn, val). For an update set generated from the evaluation of a single rule, the pairs represent the evaluation of the rule's effect statements, `<vn>:=<val>`. Formally, an update set, U, is:

---

pair. Another 'weaker' consistency is: if $vn_i = vn_j$ then $val_i = val_j$. However, this latter definition allows duplicate variable-value pairs. Accounting for these in later definitions is a distraction.

$$U = \{(vn_0, val_0), ..., (vn_n, val_n)\}$$

To incorporate time into an update set, a *timed update set*, TU is a triple consisting of the time of generation, tg, the duration, t, and the update set, U. For a single machine, t is a value in the interval, TI, of the same rule that yielded the update set.

$$TU = (tg, t, U)$$

Run In general, a *run* is a sequence of timed states, starting with the initial state, $TS_0 = (0, S_0)$. Formally:

$$TS_0,\ TS_1,\ TS_2,\ \ldots,\ TS_n$$

The timed states must be sorted in ascending order of the global time that occurs in each step and each global time must be unique (Formally, for any $TS_i = (gt_i, S_i)$ and $TS_j = (gt_j, S_j)$, if $gt_i < gt_j$ then TS_i must precede TS_j in the run, and $gt_i \neq gt_j$).

Given the initial state, TS_0, and a sequence of update sets, sorted in order of ascending tg (Formally, for any $TU_i = (tg_i, t_i, S_i)$ and $TU_j = (tg_j, t_j, S_j)$, if $tg_i < tg_j$ then TU_i must precede TU_j. To allow for parallel composition, there is **no** requirement that elements of the sequence must be unique).:

$$TU_1,\ TU_2,\ \ldots,\ TU_n$$

The remaining run, or state sequence, can be determined by applying the $\circ$ operator.

$$TS_i = TS_{i-1} \circ TU_i = (gt_{i-1}, S_{i-1}) \circ (tg_i, t_i, U_i) = (tg_i + t_i, S_{i-1} \circ U_i)$$

$$S_i = S_{i-1} \circ U_i$$
$$= (S_i - \{(vn, val) \mid (vn, val) \in S_i \; and \; (vn_i, val_i) \in U_i \; and \; vn = vn_i\}) \cup U_i$$

The *run of a single main machine*, t_{main_run}, defined as the duration when the machine first generates an update set to when the last update set is applied, is the sum of the durations of all of its update sets. For a single main machine (which may or may not be hierarchically composed), that executes n steps, and therefore applies n update sets:

$$t_{main\_run} = \sum_{i=1}^{n} t_i$$

Note that when there are multiple main machines running in parallel, this equation is still true. However the duration of the run of the entire system, t_{run}, may vary from this value if there are multiple main machines running in parallel (parallel composition). In that case, t_{run}, is equal to the global time, gt, of the last global state.

Example, one main machine For a **single** TASM (a single main machine, that may or may not be hierarchically composed), the chronological actions (for lack of a better word) that generate a run can be described as:

0. Start in the initial state.
1. All guards are *instantaneously* evaluated in the current state.
2. The effect expressions in the enabled rule are evaluated *instantaneously* to yield an update set.
3. The machine is 'busy' during the duration, t, of the update set. (How the duration is simulated can vary greatly. The machine could use a wall clock as global time and wait for duration, t, or it could increment global time by t and proceed immediately).

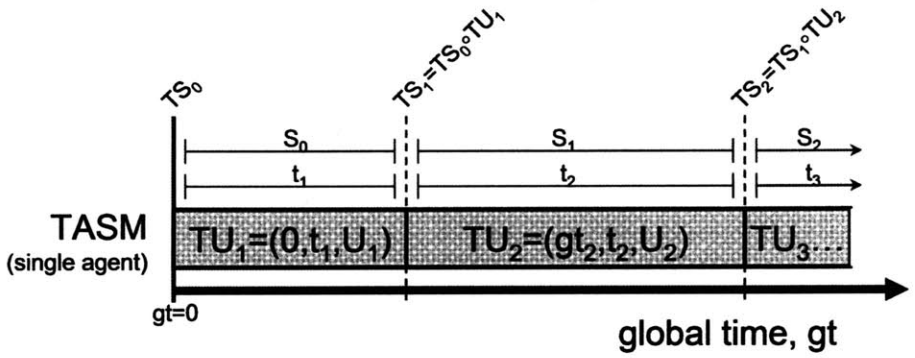

Figure 2-4: Example time line for a run of a single TASM

4. Use the ∘ operator to apply the update set to the current global state to create the new global state. Repeat from action 1.

Figure 2-4 depicts the relation of a states and update sets relative to the progression of time. Global time starts at 0, and progresses to the right. The solid color blocks correspond to the duration used by each update set. Spaces are included in the time and space intervals for clarity.

2.2.5.2 Hierarchial Composition

A hierarchial composition is when one or more rules are nested in another rule. Syntactically, this occurs when a sub or function machine call appears in a rule. There are two ways hierarchial composition can be explained. One way is to show how the update sets resulting from nested rules can be merged. The other is to show that a machine that is hierarchically composed is equivalent to a machine that is not.

Before continuing, we clarify the cases of hierarchial composition. Hierarchial composition can occur when there is: 1. a function machine in rule guard. 2. a function machine in effect expression. 3. a sub machine in effect expression. Or any combination of these cases (i.e. two function calls in the guard, one sub and one function machine call in the effect expressions).

Merging Update Sets The goal of merging update sets, is to produce one update set at the main machine level that merges the update sets produced by nested rules. The resulting update set can then use the $\circ$ operator as described in section 2.2.5.1.

When a function call appears in a guard (case 1), no composition of update sets is needed because the evaluation of a guard should take no time and resources. So any time or resource annotations appearing in the rules of the function machine are ignored.

When more than one function or sub machine call occurs in the effect expressions of a single rule (multiple cases 2 & 3), the $\otimes$ operator can be used to compose the update sets. The $\otimes$ operator is overloaded, commutative and associative. Given two update sets, TU_1 and TU_2, produced by evaluating a pair of sub or function machine calls from within the effect expressions of the same rule:

$$\begin{aligned} TU_1 \otimes TU_2 &= (tg_1, t_1, U_1) \otimes (tg_2, t_2, U_2) \\ &= (tg_1, t_1 \otimes t_2, U_1 \cup U_2) \\ &= (tg_2, t_1 \otimes t_2, U_1 \cup U_2) \end{aligned}$$

Where: tg_1 is always equal to tg_2, in a hierarchial composition.

$$t_1 \otimes t_2 = \begin{cases} t_1 & \text{if } t_2 = \bot \\ t_2 & \text{if } t_1 = \bot \\ 0 & \text{if } t_1 = \bot \text{ and } t_2 = \bot \\ max(t_1, t_2) & \text{otherwise} \end{cases}$$

When a function or sub machine appears in the effect expression of a rule (cases 2 & 3), the $\oplus$ operator can be used to compose the update sets. The $\oplus$ operator is overloaded, and is *not* commutative, but it is associative. Given two update sets, the

update set of the parent rule, TU_p, and the update set of the child rule (auxiliary machine that is called), TU_c:

$$\begin{aligned} TU_p \oplus TU_c &= (tg_p, t_p, U_p) \oplus (tg_c, t_c, U_c) \\ &= (tg_p, t_p \oplus t_c, U_p \cup U_c) \end{aligned}$$

Where: tg_c is always equal to tg_p, in a hierarchial composition.

$$t_p \oplus t_c = \begin{cases} t_c & \text{if } t_p = \bot \\ 0 & \text{if } t_p = \bot \text{ and } t_c = \bot \\ t_p & \text{otherwise} \end{cases}$$

The distribution of the $\oplus$ operator over the set of consumed resources is the same as for the $\otimes$ operator.

Equivalent Machines An equivalent machine can be formed by 'flattening' the hierarchy. The examples of this are not reproduced here. Please refer to Theorem 4.1 and Theorem 4.2, as found in section 4.3.7 of [33].

2.2.5.3 Parallel Composition

A parallel composition occurs when more than one main machine operates in the same environment. Syntactically, we say a parallel composition occurs in a particular Configuration, when more than one main machine is initialized in that Configuration.

The use of time annotations in TASM allows machines operating in parallel to run synchronously or asynchronously. When running in parallel, update sets generated by different main machines are applied to the global state ($\circ$ operator) in the order of their time of application, ta (For a timed update set, $TU = \langle tg, t, U \rangle$, $ta = tg + t$).

When two update sets have the same time of application, the update sets can be

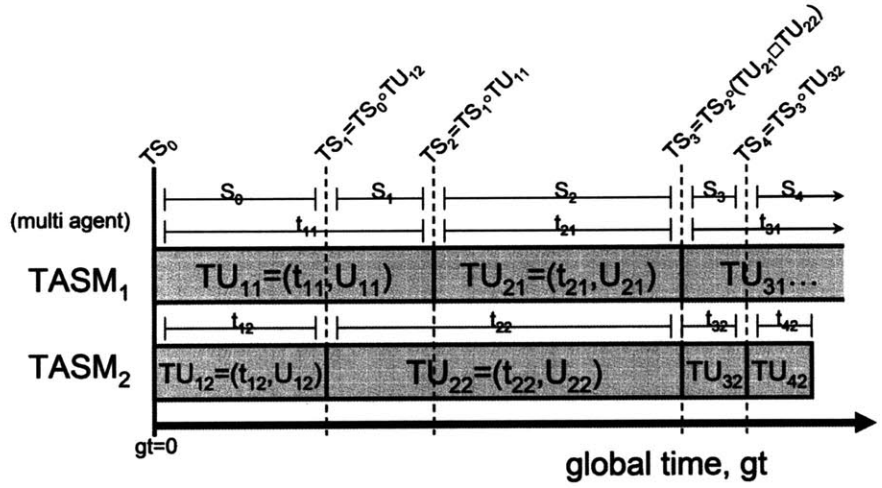

Figure 2-5: Example time line for a run of two TASMs

composed using the $\boxdot$ operator. The resulting update set can then be applied to the global state using the $\circ$ operator. For any two update sets $TU_1 = (tg_1, t_1, U_1)$ and $TU_2 = (tg_2, t_2, U_2)$ generated by two different main machines, where $tg_1 + t_1 = tg_2 + t_2$:

$$\begin{aligned} TU_1 \boxdot TU_2 = (tg_1, t_1, U_1) \boxdot (tg_2, t_2, U_2) &= (tg_1, t_1, U_1 \cup U_2) \\ &= (tg_2, t_2, U_1 \cup U_2) \end{aligned}$$

The choice of which resulting update set to use is not important.

Example Figure 2-5 depicts the relation of states and update sets relative to the progression of time for a parallel composition of two main machines. Global time starts at 0, and progresses to the right. The solid color blocks correspond to the duration of each update set. Spaces are included in the time and space intervals for clarity. Timed update set, TU_{ij} is the ith step of the jth machine. Timed update sets TU_{21} and TU_{22} both have the same time of application. The resulting timed update set is calculated with the $\boxdot$ operator,

2.2.6 Supported TASM

The primary goal of this thesis is to preserve functional behavior through transformations, so the time and resource annotations of the TASM language are largely ignored. In general, these annotations would be hard to preserve in an implementation language since they describe non-functional behaviors.

However, the time annotations also play a critical role in the TASM language of specifying the behavior of parallel machines. For example, the time annotations could be such that one machine's rule would always execute before another machine's rule. Unfortunately, this thesis does not specifically address the functional behavior resulting from the interleaving of main machines, so multiple main machines are also not supported.

In terms of what can and cannot be included in a TASM specification. All aspects of the TASM language, including time and resource annotations, channels, as well as multiple main machines can be used, and are syntactically represented in the transformation framework's intermediate TASM SDG. However, the semantics of those language features are not represented in a TASM SDG, and as a result will not be preserved in the transformations.

2.3 Java

The Java programming language is an object oriented language developed and maintained by Sun Microsystems. The first publicly released version, Java 1.0 was released in 1995. As of the writing of this thesis, the most current version is Java 1.6.14. Due to naming convention changes, this version is also called "Java 6 update 14".

The Java language uses a unique two-phase compilation process that gives the langauge its advertised flexibility of "Write Once, Run Anywhere" (WORA). The first compilation phase translates java code to java bytecode, a platform independent representation of the code that is quick to parse. The bytecode or ".class" files, serve as the compiled equivalent of ".o" files from the C language family. The second compilation process, referred to as just-in-time (JIT) compilation, occurs in the Java

Virtual Machine, a runtime environment tailored to each system, which handles the final translation to machine specific code. In recent years, the JIT compiler has been augmented with compiler optimizations. The culmination of the two-phase compilation process and optimizations makes Java code difficult to time statically.

2.3.1 Supported Java

The subset of the Java language supported by the transformation framework cannot be as clearly stated as the supported subset for TASM. In general, there are two limitations on what subsets are supported: the language features that are representable by the JSDG, one of the intermediate graph forms in the transformation framework, and the type of transformations available between the TASM SDG and the JSDG.

In general, a JSDG can represent Java methods, variables, arrays, objects, classes, and class inheritance [44]. The major restriction to the supported subset of the Java language is that multi-threading is not representable by the JSDG. Other SDG variants, such as the multi-threaded dependence graph [47] is capable of representing threading, but because that research is relatively new, there has not been extensive use of that structure in the behavior preserving transformations required by the framework. In addition, there was no reason to select a multi-threading language because the semantics of parallel machines is not fully describable in the TASM SDG. Since transformations between Java and a JSDG are procedural, it is not flexible to language features not supported by the JSDG. A Java program that uses features not represented in a JSDG breaks the transformation framework.

The second restriction on the useable subset of Java stems from what transformations are defined between a JSDG and TASM SDG. These transformations make it difficult to describe what features of Java of supported. For example, a loop in Java which uses an array may be transformed to a sequence of statements using primitive variables by applying a loop unrolling transformation. Therefore, in certain circumstances, an array in Java can be expressed in a TASM SDG (despite the lack of an explicit array type in either TASM or its SDG), and is therefore supported by the transformation framework. The use of TGGs for specifying the transformations be-

tween a JSDG and TASM SDG also allow the set of transformations to be expanded and combined in different ways. Thus, the framework contains the ability to add future support for Java features.

In general, we will say that a Java feature not expressible in a JSDG is strictly not supported.

2.4 Segue into Chapter 3

In this chapter we presented three languages: the basic ASM language, the modeling language, TASM, and the implementation language, Java. We presented the ASM theory to provide a background for the TASM language, and then presented, in detail, the TASM language. The thoroughness of that presentation will be useful in understanding the following chapters. Since TASM and Java constitute the ends of our transformation framework, we also discussed the supported subsets of either language and the reason for those restrictions to scope the framework's applicability.

The following chapter, Chapter 3, introduces the TASM System Dependency Graph (TASM SDG) that can be built from a specification written in the TASM language. It presents the structure of the TASM SDG, as well as the algorithms required to construct it.

Chapter 3

TASM System Dependency Graph

In this chapter we present the TASM System Dependency Graph (TASM SDG) and associated algorithms required to create it. Section 1 discusses the motivations for this approach. Section 2 presents the TASM SDG structure. And section 3 discusses in detail, the key algorithm required to generate the TASM SDG, which makes explicit the TASM state transitions.

3.1 Technical Motivation

In the light of the syntax and semantics of the TASM language, we return to qualitatively explain what it means for a transformation to be robust.

One way to judge the quality of a mapping is by whether or not it preserves the functional behavior of a program. Provided an ASM formulation is written in guard-effect form [Eq. 4.1] and the rules are consistent (that is only one rule can be enabled at a time), the transformation from ASM to implementation requires only 3 major steps: 1. For each rule, introduce new variables so the effect expressions can execute sequentially as assignment statements. 2. rewrite each rule as an if-else statement 3. Place all the if-else statements in one infinite loop. There are details concerning how nested rules should be flattened and how using an implementation language's syntax affects the scoping of variables and use of reserved words, but the general concept is simple. For the TASM language, all the provisions are true, and this transformation

does preserve functional behavior. Unfortunately, the resulting program would consist of many sequential if statements nested in a loop for each TASM. If the same set of rules are 'applied in reverse' to constitute the reverse transformation, it would only be applicable if the implementation program used the same structure. The mapping this pair of transformations describes is restricted to a subset of the implementation language with a prescribed structure.

Another way to judge the quality of a mapping is on how complete the transformation rules are, such that they can start from and end at any combination of syntax structures. In the previously described transformation, a fourth step could be added that would make the transformation more flexible by having complete knowledge of all the partial orders of all the expressions. Such knowledge would allow data dependency based transformations as implemented in most modern compilers [5, 24, 6, 42]. The resulting mapping could allow a more natural structuring of the implementation.

The proposed transformation framework requires that a transformation preserve behavior, and leverages compiler based code transformations in an attempt to be complete. In order to facilitate compiler based transformations, the intermediate graph structures used in the transformation framework are System Dependence Graphs, a graph structure commonly used in many restructuring/optimizing compilers.

3.1.1 Related Work

An ancestor to the System Dependence Graph, the Program Dependence Graph (PDG) was first introduced by Ottenstein & Ottenstein in 1984 [32]. Unlike an abstract syntax tree, which accurately describes a program's syntax, the PDG was a succinct way of representing a program's semantics. Only applicable to a monolithic program, with no procedures and no objects, the PDG merged the control-flow graph and the data dependency graph. The resulting structure was a functionally equivalent representation of a program [25]. Although originally proposed for compiler optimizations, it has subsequently been used for various software engineering activities including program slicing, debugging, testing, and complexity measurements [47].

The System Dependence Graph (SDG) was first described by Horwitz et. al. as

an extension of PDG to model a multi-procedure program [Interprocedural Slicing Using Dependence Graphs]. A large body of subsequent work has expanded SDGs to model classes, inheritance, polymorphism, and dynamic binding [13, 30, 26, 44, 38]. Other work has tailored SDGs to model specific features including threads [47].

3.2 TASM System Dependence Graph

The TASM SDG proposed uses a subset of the multi-threaded dependence graph presented in [47], but decomposes the statements that would usually form the nodes into expression trees, similar to those found in a PDG. Formally, a TASM SDG is an attributed directed graph with labeled edges. This means that while the vertices of the SDG can be grouped into classes, they also have have attributes in the form of *name, value* pairs that can distinguish two nodes of the same class. In addition, certain types of edges, which are defined to exist between a pair of vertices can have a set of labels. Where a label is just a name, with no value pairing. The purpose of the attributes and edge labels will be presented in the following sections.

In total, a TASM SDG consists of 13 classes of nodes and 5 types of directed edges.

3.2.1 Edge Types

The edges of a TASM SDG connect only two vertices and represent an ordering between the vertices they connect. The ordering is called a dependency. The head of each edge connects a vertex that is somehow dependent on the vertex at the tail.

Three of the edge types are used to model the dependencies between a method call and a method body. They are referred to as *call*, *linkage-entry*, and *linkage-exit* edges. Their use will be elucidated in the following section on vertex classes [Section 3.2.2].

The remaining two edges represent *control dependencies* and *data dependencies.* These edges were the original and only two edges used to define a PDG. A control dependency represents a semantic construct of the language where one execution

depends on another. For example, if we block a language into statements and expressions, a control dependency exists from the expression that guards a control structure to each of the statements nested directly inside that structure.

In contrast, a data dependency refers to how operators need to be ordered due to their use of variables. Specifically, that a rearranging of the order of the operations would violate functional equivalence. From a program slicing perspective, a data dependency is normally defined at a higher level, where one statement is dependent on another due to their use of variables. We specifically use a finer grained definition suitable for compiler optimizations, as originally described in [18]. A data dependency edge can be further classified as a *flow dependency*, *anti dependency*, or *output dependency* edge. A flow dependency, sometimes called a *true dependency*, exists between an assignment to a variable and its subsequent use. An anti dependency exists between a variable use and its subsequent assignment. An output dependency exists between two assignments to the same variable. All three types of data dependency edges can appear in a TASM SDG. Figure 3-1 illustrates these dependencies in both the normal block-statement form in which they are usually presented, and graphically as they would be in the TASM SDG. In the TASM SDG, variables always appear in assignment vertices: The shaded vertices therefore do not actually correspond to a class of vertices, and are only used in this figure for brevity.

Depending on their use, control edges can be labeled with 'T' or 'F' to indicate the that the subsequent vertex depends on the preceding operation yielding a True or False value, respectively. Data dependency edges can also be labeled with a set of numbers. The specialized labeling cases will be discussed in the section on TASM SDG construction.

3.2.2 Vertex Classes

A TASM SDG consists of 13 classes of vertices. They are: Entry, Specification, Environment, Configuration, Call, Formal In, Formal Out, Actual In, Actual Out, Operation, Constant, Assignment, and Region. The Entry vertex defines a starting point of execution on which subsequent operations can have a control dependency on.

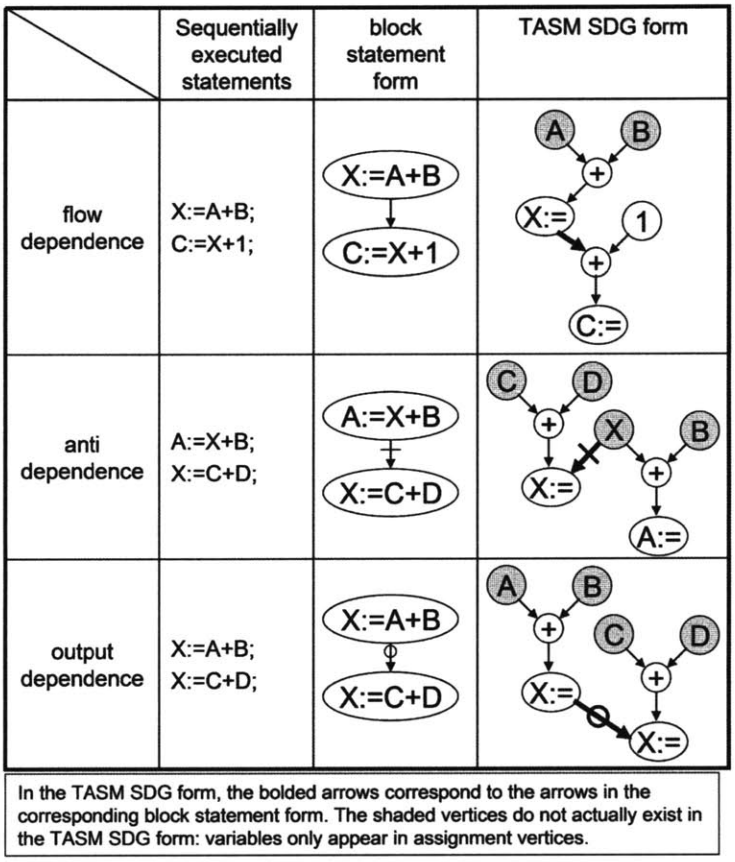

Figure 3-1: An example of the three types of data dependencies.

It is used to organize operations. The Specification, Environment, and Configuration classes are special classes of the Entry node that function identically but have different sets of attributes and correspond to their same-name structures in the TASM language. The Call, Formal In, Formal Out, Actual In, and Actual Out vertices have many uses. They are usually used to define a new scope, such as one that would exist when calling a function machine. But, they are also used as 'pass-through' vertices, that act as place holders in the TASM SDG representation of machines and constructors, so that data dependency edges can be compartmentalized. The remaining vertices roughly correspond to those found in an expression tree and are used to represent functional behavior.

The following section formally defines each of the vertex classes and their attribute set. To simplify description, the vertex at the tail of an edge will be referred to as the parent and the vertex at the head will be called a child.

3.2.2.1 Entry

An Entry vertex acts as the single starting vertex for a set of operations. It can have multiple Formal In, Formal Out, and Operation vertices as children, and is connected to all of its children via a control dependency. It can have more than one parent, which could be a Specification, Call or Operator vertex. Its attributes are "type" and "name". A "type" attribute can have as its paired value: "constructor", "main machine", "function machine", or "sub machine". A "name" attribute has as its paired value the name of the main, function, or sub machine for which the Entry vertex starts. The name attribute of a "constructor" type Entry vertex has no "name".

3.2.2.2 Specification

There is only one Specification vertex per instance of TASM SDG. It is a special class of an Entry vertex that corresponds to the specification construct of the TASM language. It must have as its children one Environment vertex, and may have zero or more Configuration and Entry vertices, all of which are connected to it via a control

dependency. It has no parents, and is considered the top-most Entry vertex to the entire TASM SDG representation of a specification. Its attributes correspond to those elements of the project construct of the TASM language. Specifically, it can have as attributes a "project-id", "project-name", and "project-description".

3.2.2.3 Environment

There is only one Environment vertex per instance of TASM SDG. It is a special class of an Entry vertex that corresponds to the environment construct of the TASM language. Its only children are of class Assignment, all of which are connected to it via a control dependency. The children correspond to the environment variable initializations. Its only parent is the Specification vertex. Its attributes are "resources" and "channels", each of which has for values a list of resources and channels. If the TASM SDG is ever extended to support either, the corresponding attribute can be removed.

3.2.2.4 Configuration

A Configuration vertex is also a special class of Entry vertex. There can be more than one Configuration vertex, which correspond to the named configuration constructs in the TASM language. It can have zero or more Assignment or Call verticies as children. The Assignment verticies correspond to the 'overriding' variable initializations that can occur in the configuration construct of the TASM language. The Call verticies correspond to the main machine instantiations. Like all special classes of the Entry vertex, a Configuration vertex is connected to its children via a control dependency. Each Configuration vertex has one parent, the Specification vertex. The attributes are "name" and "description", whose value pairs correspond to the name and description of the configuration construct.

3.2.2.5 Operation

The operation vertex is the primary vertex for composing functional behavior. It always has one parent that is either a Region vertex or an Entry vertex, to which it

is connected via a control dependency. And it can have any number of other parent nodes, including Constant, Assignment, or Formal In, on which it depends for input. Operators that are not commutative, must have distinguished parent connections, such that the inputs it depends on can have a clearly defined order.

It can have as a child zero or more Operation, Formal Out, or Formal In verticies. Operators that produce a logical output, are required to have only one child, a Region vertex, by which it is connected via a control edge. That control edge must also be labeled with a T or F to indicate whether that dependency is active when it matches the output of the Operation.

An operation vertex is attributed with "operator", the specific TASM operator represented by that vertex, and "type", the output data type of that operation.

3.2.2.6 Constant

A Constant vertex represents a value in the domain of one of the types in the TASM Type Universe. A constant vertex has no parent, but can have one or more children, which must be an Operator, Actual In, or Formal Out vertex, by which it is connected via a data dependency edge. A constant vertex is attributed with "value", whose pair is the actual value of the vertex, an element of a data type. And, "type", the data type from which the value was drawn.

3.2.2.7 Assignment

An Assignment vertex represents a variable assignment. It is attributed with the "name" of the variable, and it's "type". It can have as its parent a Formal In, Actual Out, or Operator vertex, by which it is connected via a data dependency. It can have as a child a Formal Out, Actual In, or Operator vertex. The meaning of which is that the variable assigned to is being used in the operation of a subsequent vertex.

3.2.2.8 Region

A Region vertex is a very special vertex that, like the Entry vertex, is used to group operations. Specifically, a region vertex can only have one parent, an Operator vertex,

to which it is connected via a control edge labeled T. Each Region vertex can have zero or more child vertices of class Call, Operator, or Assignment, by which it is connected with a data dependency edge. The children verticies of a Region can only be evaluated if its parent vertex evaluates to True.

3.2.2.9 Call

A Call vertex corresponds to a constructor, function, or sub machine call in the TASM language. As its children, a Call vertex can have zero or more Actual In and Actual Out verticies, which correspond to the arguments passed to the called construct. These children are connected to the Call vertex via a control dependency. Each Call vertex must also have one child Entry vertex, to which it is connected via a call edge. A Call vertex is a special class of Operator vertex, and when calling a function machine, can be used as an Operator vertex. The parent of a Call vertex is almost always a Region vertex. The only time a Call operator has a different parent is when it is used to call a main machine's constructor in the Configuration vertex. The attributes of a Call vertex are "name", which has as its value pair the name of the called construct. The value of this "name" attribute matches the value of the "name" attribute of the child Entry vertex.

3.2.2.10 Formal In, Formal Out, Actual In, Actual Out

The Formal In, Formal Out, Actual In, and Actual Out verticies are attributed like an Assignment vertex, and can be thought to perform a similar activity. These verticies mark boundaries between reusable language constructs, such as constructors, main machines, function machines, and sub machines. When using one of these constructs, the Formal In and Formal Out vertices represent all the possible inputs and outputs. For a function, the inputs and outputs correspond to an input variable and an output variable. But, for a main and sub machine they correspond to monitored and controlled variables. Similarly, the Actual In and Actual Out vertices represent the boundaries of a Call vertex, such that the Actual In vertices correspond to arguments and Actual Out correspond to returned values.

Formal In and Formal Out verticies each have only one Entry vertex as a parent. Actual In and Actual Out vertices each have only one Call vertex as a parent. When calling a function for example, the Call vertex is parent on a call edge to the Entry vertex. Then, each Actual In of the Call vertex is parent on a linkage-entry edge to its corresponding Formal In of the Entry vertex. And each Formal Out of the Entry vertex is parent on a linkage-exit edge to its corresponding Actual Out on the Call vertex.

3.2.3 Construction

The control subgraph of a TASM SDG, which includes all control edges and the vertices they connect, along with a few data dependencies can be built using a one-pass, recursive algorithm over an Abstract Syntax Tree (AST) of a TASM specification, with three tables for memoization. We outline the construction process here.

There are three tables required for memoization. One records the set of variable name-type pairs. As variable declarations are encountered, they should be memoized into this table. Another records a quadruple entry consisting of a Call vertex, all corresponding Actual In and Actual Out vertices, and a unique call number for each entry of the table. As calls are encountered, they should be memoized into this table. And the third records a triple entry consisting of an Entry vertex and all corresponding Formal In and Formal Out vertices. As constructors, or main, function, or sub machines are encountered, their entry sites should be memoized into this table.

When traversing the AST in a top-down manner the Specification, Environment, Configuration and a subgraph rooted at an Entry vertex for each constructor and machine can be created. Upon completion, each entry in the call site table can be matched to a corresponding entry site in the entry table, to create the call, linkage-entry, and linkage-exit edges. Assuming the number of tokens that constitute a method call are small in comparison to the number of tokens in the entire AST, the procedural generation of the TASM SDG is linear in the number of tokens in the AST.

The one caveat to this generation procedure is that unlike a normal SDG which

represents sequentially executing programs, the TASM language uses rules with sets of effects that execute in parallel. When generating a vertex structure corresponding to two effect statements in the same rule, one which uses and the other assigns to the same variable, there is no dependency relation. Semantically, this is exactly what the SDG should show, however when a TASM SDG is transformed to a JSDG, this lack of dependency information suggests that these statements can be executed in an arbitrary order. Since this is not the case, we must make some dependency information explicit. The additional processing that must be done is whenever statements in the same rule both use and assign to the same variable, we introduce a new statement, upon which both original statements are data dependent. The new statement will assign the variable to a new a variable that will be subsequently used in the original effect statements.

Of course, a control subgraph does not constitute an entire TASM SDG. This construction process is still missing the data dependency information. To add these dependencies, we need to know a reaching set of runs.

3.2.4 Example

Figure 3-2 illustrates the control subgraph TASM SDG that results from transformation from the TASM specification given in Listings 3.1 and 3.2. The Listings specify the behavior of the hierarchically composed `LIGHT_CONTROL` main machine that was described in chapter 2. It also includes modifications to provide a richer example of a what a TASM SDG would look like. The modifications include a main machine with a dummy internal variable and constructor (lines 35-40), and a configuration that overwrites the initial value of the environment variable `intensity` with `0` (line 15).

3.3 Extracting Transitions

In this section we describe the second half of the construction algorithm for a TASM SDG, the one that adds the remaining data dependencies. The general algorithm that will be described computes the set of all reachable states of an ASM.

Listing 3.1 Light switch example - partial, hierarchial TASM Specification, part 1

```
PROJECT:
    ID:
      7190e928-9081-4f9a-9241-1c8aaebc5a09
    NAME:
      Light Switch Example
    DESCRIPTION:
      This is a partial TASM specification for the light switch example.
CONFIGURATIONS:
    CONFIGURATION:
        NAME:
          Hierarchial Composition
        DESCRIPTION:
          One main machine models the light intensity controller.
        VARIABLE INITIALIZATIONS:
          intensity := 0;
        MACHINE INITIALIZATIONS:
          light_control := new LIGHT_CONTROL(0);
ENVIRONMENT:
    TYPES:
      switch_status := {UP, DOWN};
    RESOURCES:
    CHANNELS:
    VARIABLES:
      Integer[0,100]  intensity  := 0;     // 0=OFF, 100=FULL ON
      switch_status   switch     := DOWN;
MAIN MACHINES:
    MAIN MACHINE:
        NAME:
          LIGHT_CONTROL
        DESCRIPTION:
        MONITORED VARIABLES:
          switch;
        CONTROLLED VARIABLES:
          intensity;
        INTERNAL VARIABLES:
          Float dummy := 0.0;
        CONSTRUCTOR:
          LIGHT_CONTROL(Float dummyIn){
            dummy := dummyIn;
          }
        RULES:
          R1: Turn On
          {
            if intensity < 100 and switch = UP then
               intensity := TURN_ON(switch,intensity); // function machine call
          }
          R2: Turn Off
          {
            if switch = DOWN then
               TURN_OFF();                              // uses sub machine
          }
          R3: Wait
          {
            t:=next;
            if intensity >= 100 and switch = UP then
              skip;
          }
```

Listing 3.2 Light switch example 2 – partial, hierarchial composition

```
FUNCTION MACHINES:
    FUNCTION MACHINE:
        NAME:
          TURN_ON
        DESCRIPTION:
        INPUT_VARIABLES:
          switch_status  switch;
          Integer[0,100] intensityIn;
        OUTPUT VARIABLES:
          Integer[0,100] intensityOut;
        MONITORED VARIABLES:
        RULES:
          R1: Turn On
          {
            if switch = UP then
               intensityOut := intensityIn + 1;
          }
          R2: Else
          {
            t := next;
            else then
               skip;
          }
SUB MACHINES:
    SUB MACHINE:
        NAME:
          TURN_OFF
        DESCRIPTION:
        MONITORED VARIABLES:
          switch;
        CONTROLLED VARIABLES:
          intensity;
        RULES:
          R1: Turn Off
          {
            if switch = DOWN then
               intensity := 0;
          }
          R2: Else
          {
            t := next;
            else then
               skip;
          }
```

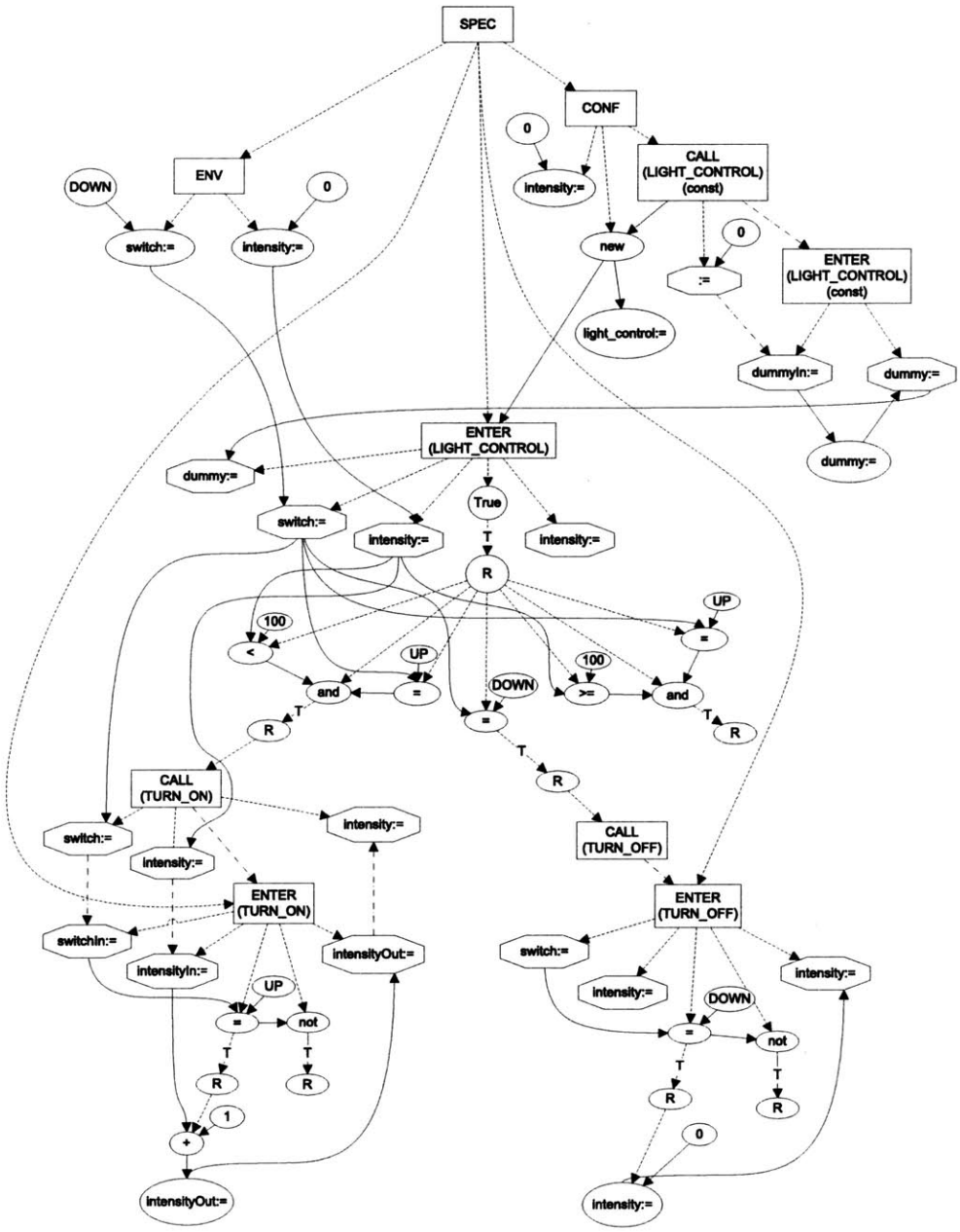

Figure 3-2: Light Switch Example Partial, Hierarchial TASM SDG

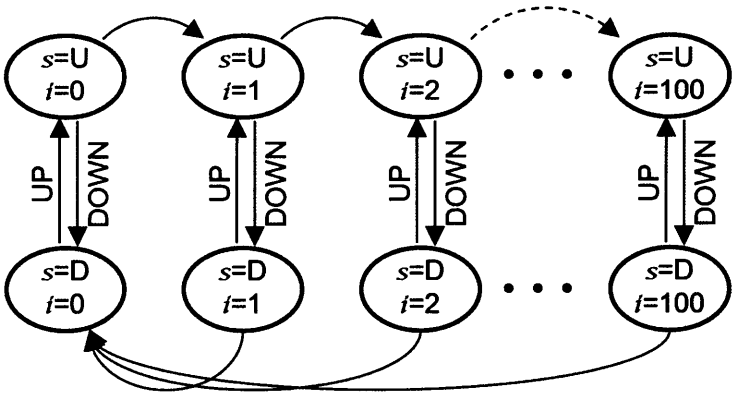

Figure 3-3: The global states of the light switch example.

3.4 Implicit Transitions

Computing the set of all reachable states for an ASM is difficult because an ASM makes its abstract state transitions implicit. In a Finite State Machine, each state has a set of outgoing arcs, whose heads explicitly identify the subsequent state. However, an ASM uses guard-effect rules to represent behavior. Given a certain state, it is impossible to know the next state without executing the rules.

We return to the Light Switch Example of chapter 2 to provide a motivating example. Figure 3-3 shows a basic FSM specification (with no entry actions) including all the reachable global states, and all the transitions possible by the main machines. The `switch` variable is abbreviated `s`, and the `intensity` variable is abbreviated `i`. It should be obvious from this diagram, which is a common graphical representation of an FSM specification what the state transitions are. In contrast, Figure 3-4 graphically depicts the equivalent ASM specification. The guards used in an ASM group the global [abstract] states in sets known as *hyperstates*. For each hyperstate there may be an associated effect that causes a transition, but the effect is in the form of an expression (or sets of expressions) that must be evaluated.

Our goal is to therefore make the transitions between hyperstates explicit. Such that we can say given a certain hyperstate what the following hyperstates will be, without evaluating any expressions.

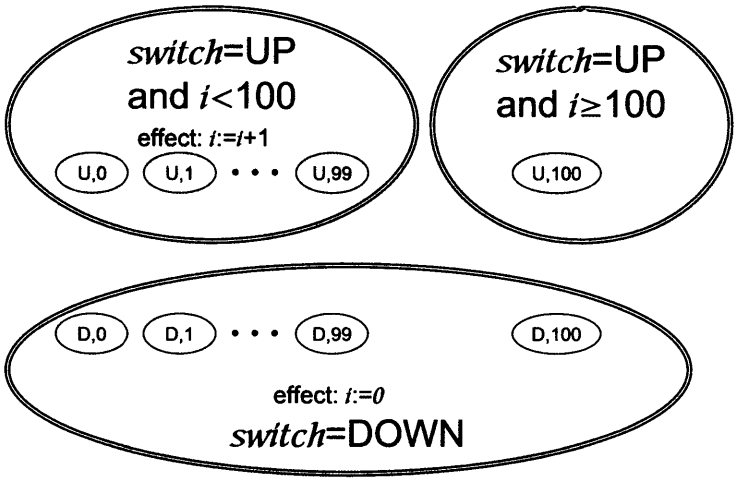

Figure 3-4: The global states of the light switch example.

3.5 Variable Space Introduction

The algorithm we formulate learns transitions by traversing variable space. Intuitively, variable space is simply the set of all possible global states. For a problem with n variables, the Variable Space is formalized as the set $VS = \{VU_0 \times VU_1 \times ... \times VU_n\}$, where VU_i is the value universe or domain of the type of variable i. Graphically, variable space can be thought of as a geometrical space, where each axis corresponds to a variable and spans the values of that variable's domain. A 'point' in the variable space is a sequence of values, where one value is drawn from the domain of each axis.

In order to identify a region of the variable space, we can specify all value sequences that constitute the region. But, a more succinct way is to use set builder notation. For example, for a variable space consisting of 3 integer variables bounded to a value universe ranging from 0-2, inclusive, one way to identify a region is to write:

$$\{(0,0,1),(0,0,2),(0,1,0),(0,1,1),(0,1,2),(0,2,0),$$
$$(0,2,1),(0,2,2),...,(1,0,2),...,(2,2,2)\}$$

But the equivalent set can also be expressed as:

$$\{s|s = (v_1, v_2, v_3) \text{ and } v_1 < v_2 + v_3\}$$

In terms of an ASM language, a hyperstate is a region of variable space, identified by the guards. The guards in an ASM specification divide the variable space into regions in the same manner as the set builder notation. The guard of a rule, which uses n variables, x_i , with values v, drawn from the variable's type, identifies a region of variable space that in set builder notation can be expressed as: $\{s|s = (v_0, ..., v_n)$ and $\texttt{guard}(\text{v}_0, ..., v_n)\}$. Where we use the notation $\texttt{guard}(v_0, ..., v_n)$ to indicate the evaluation of the guard at those values.

A rule, in the form `if <guard> then <effect>`, identifies a region of variable

space and transforms it to another region of a variable space of the form:

$$\{s|s = effect(v_0, ..., v_n) \text{ and } \texttt{guard}(v_0, ..., v_n)\}$$

Where, we use the expression $effect(v_0, ..., v_n)$ to indicate the application of the effect on those values. $effect$ is taken to be a function of the form $f : VS \rightarrow VS$.

3.6 Extracting Transitions

The goal in extracting transitions is to create a complete set of reachable states. For a bounded ASM, where all the types have a finite value universe (a.k.a all the variables are bounded), this corresponds to a finite state space, and thus can be perfectly described as a finite state machine. But, what states should constitute this FSM?

Instead of uniquely identifying each abstract state in the ASM as a state in the FSM, we will use regions of variable space, identified by guards. But, in general, an ASM, can have more than one rule enabled at the same time. The region identified by those rules is a combination of the boolean indecomposable parts of the guard. Where a boolean indecomposable part is an expression that produces a boolean value, but for which all subsets do not produce a boolean value. We now redefine a *hyperstate* to be the set of all such combinations. For a machine with n boolean indecomposable parts, this constitutes 2^n possible hyperstates. The FSM we wish to produce from our ASM will therefore consists of states that are hyperstates of the ASM.

What are the transitions? The guard of a rule identifies a region in variable space, which is then transformed to another region in variable space by the rule's effect. By finding out which regions defined by other guards intersects the transformed region, we can identify the sequence of rule executions, and hence the transitions between states. Unlike a typical FSM, where transitions are guarded by events, these transitions will be unguarded. A state transition corresponds to a step of the ASM.

3.6.1 Algorithm

The basic algorithm for extracting transitions is fairly straightforward, and is defined here as operating in variable space. The recursive algorithm takes 5 arguments: s, a, τ, C, B. s is the current state, represented by a boolean expression. a is the active region, a subset of variable space, VS. τ is a set of transitions, where a transition is a pair of states (s_i,s_j). C is a subset of variable space, and keeps track of the portions of variable space already visited by the algorithm. B is the complete, bounded, variable space of the problem, i.e. $B = VS$.

The algorithm also makes use of the function $region()$, which takes in a guard, which is a boolean expression and return the corresponding region in variable space in which that guard is true.

For the algorithm to work properly, the set of rules of the ASM must be consistent, that is to say no two rules can be enabled at the same time. If the ASM is not consistent, then the rules should be combined to form a single set of rules that is consistent. The set G, used in the algorithm is the complete set of consistent guards from the ASM.

The algorithm should be called with the arguments: s_0, the initial state, which can be the set of the guards that define the initial active region. a_0, the initial active region represents the region of variable space from which the user believes the ASM will start executing. τ is initially the empty set. C is initially the empty set. And B is the variable space of the ASM. When the algorithm completes, τ will hold the set of transitions that the ASM will execute.

3.6.2 Example

Given: 1.) the variables defined in global bounded data types, i.e. [IntegerMinValue, IntegerMaxValue]. 2.) each variable set to the domain of initial values they could be. (or a function of other variable's domains), and 3.) a program in guarded-action form. Symbolic Interval Analysis follows the evolution of the domain of initial values as each guard is triggered.

Algorithm 1: TransitionExtraction(s,a,τ,C,B)

```
begin
    if B - C = ∅ then
        return;
    for g ∈ G do
        a' ← region(g) ∩ a
        if a' ≠ ∅ then
            a' ← effect(a')
            τ ← τ ∪ {(s, region(g))}
            C ← C ∪ a
            TransitionExtraction(g,a',τ, C, B)
    end
```

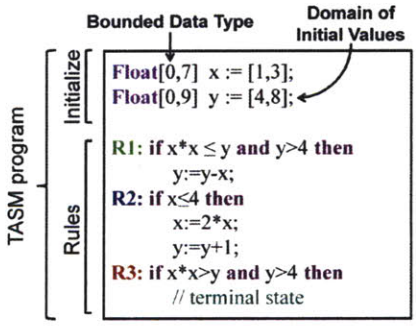

Figure 3-5: Simplified TASM Specification with 3 rules.

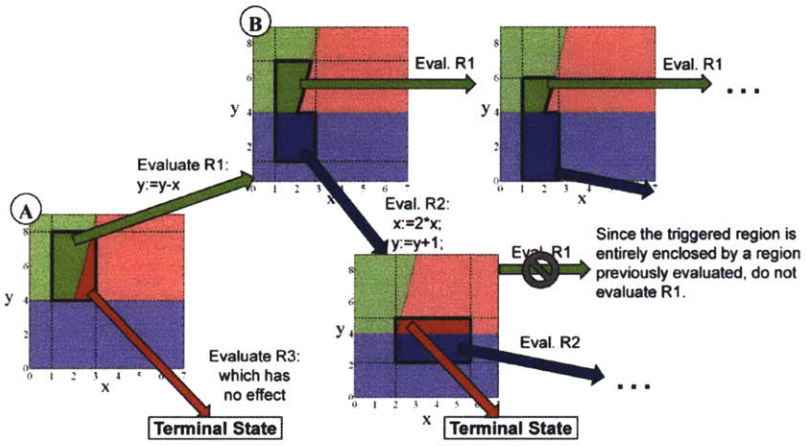

Figure 3-6: The charts represent the variable space, divided into regions in which each rule is true. A. The region defined by the initial values are outlined in black, and are shown to trigger (intersect the regions defined by) rules 1 & 3. B. The evaluation of rule 1s effect (y:=y-x) via the properties of interval arithmetic alters the shape of the region and triggers rules 1 & 2. Evaluation proceeds in a branching manner, with each intersection identifying an abstract state transition.

In the Figure 3-5, there are 2 variables, x and y, defined in terms of an explicitly bounded data type, initialized with not a single value, but a domain of values. To the immediate right is a colored graph 3-6. The three colored regions, green, blue, red, correspond to the values of variables x and y that make rule 1, rule 2, and rule 3 true, respectively. The black box indicates the triggering region, or the domain of initial values. Since the black box intersects with the green and red regions, it triggers the corresponding rules. In the graph to the immediate top-right, we see that rule 1 has been triggered, and by applying the rules of interval arithmetic to the effect expression, the shape of the triggering region (the black outlined box) has changed and is now triggering a new rule. The algorithm proceeds in this fashion exploring a tree of possible transitions, each arrow in this tree corresponds to an abstract state transition in the program.

The example uses continuous, floating point data types, but the algorithm is extendable to integers and user defined types.

3.6.3 Implementation

The algorithm uses variables that represent regions of variable space. These in turn are just large sets of values. Instead of storing these values, a more efficient approach is to store them as intervals or boolean expressions, the intersections of which define the boundaries of the variable space, as done in set builder notation. The catch with using such representations is not everything can be efficiently represented as regions. For example, if an effect expression does the operation $i := i + 2$, and a subsequent guard checks for $i > -10$ and $i < 10$. The region cannot be stored as an interval from -10 to 10, because depending on the initial value of i, the region may actually be every odd value of i in that interval.

3.6.4 Soundness

The algorithm may not be sound. The algorithm may also include transitions that do not actually occur. For the specific application of the algorithm, in creating a flow-

dependency graph for mappings, this is not terrible. The extra dependency arrows simply restricts the possible transformations that can be applied to the graph (i.e. a dependency might say rule 1 must execute before rule 2, when their execution order doesn't actually matter).

The problem lies in tolerance. The geometry based representation of real data types assumes the interior space of each region is continuous. But, even continuous data types in programming languages take on discrete values. Let's say we have the following program:

Listing 3.3 Light switch example 2 – partial, hierarchial composition

```
Float x:= [100e200,100e201];
      // Assume the domain is large, where the precisely
      // representable numbers are very sparse.
-----------------------------------
Rule 1:
{

  if <some boolean statement> then
     x:= x/100e200;
     // the new x should have interval [1,10],
     // but do we know if 100e201 is representable?
     // if it's not representable, the interval may be [1,1].
}
Rule 2:
{
  if x>5 then
     some effect ...
}
```

Assume we are executing the effect in rule 1, in the method of detecting intersections we can accurate calculate the interval after rule 1's effect is applied is [1,10]. But, in Java, the calculation may be [1,1]. In the continuous interpretation, Rule 1 transitions to Rule 2. In the discretized interpretation, it doesn't.

Of course, this problem with discretization is evident even in some accepted transformation algorithms (i.e. loop skewing), so it is important to be aware of this limitation.

3.6.5 Termination

The algorithm terminates. There are only two leaves possible in the search tree: the algorithm either evaluates a terminal rule (which has no effects), or the algorithm does not need to evaluate a region because it is enclosed by a previously evaluated region.

3.6.6 Adding Dependencies

We now return to adding dependencies to the TASM SDG. With the transition extraction algorithm, the process is also fairly simple. First, construct a connected graph from the list of transitions created by the transition extraction algorithm, the FSM. Starting at a terminal node in the FSM, follow the transitions backwards. For each hyperstate encountered until the initial hyperstate, identify the hierarchy of rules in the TASM SDG that are enabled when in that state. Create data dependency links between the rules and effects that are enabled in the current hyperstate to the next. Mark the current hyperstate in the FSM. Repeat this process starting at an unmarked terminal node, and continue until all nodes in the tree are marked.

Chapter 4

Mapping from ASM to Java

4.1 Triple Graph Grammars

Triple Graph Grammars (TGGs) are a technique for defining the correspondence between two different types of models in a declarative way [36]. The power of TGGs comes from the fact that the relation between the two models cannot only be defined, but the definition can be made operational so that one model can be transformed into the other in either direction; even more, TGGs can be used to synchronize and to maintain the correspondence of the two models, even if both of them are changed independently of each other; i.e., TGGs work incrementally [16]. TGGs were introduced more than 10 years ago by Andy Schurr. Since that time, there have been many different applications of TGGs for transforming models and for maintaining the correspondence between these models. To date, there have been several modifications, generalizations, extensions, and variations [29]. Moreover, there are different approaches for implementing the actual transformations and synchronizations of models. In this section, we present the essential concepts of TGGs, their spirit, their purpose, and their fields of application.

4.1.1 Syntax

A Triple Graph Grammar is fundamentally a set of disconnected graphs, called grammars, which can be used to relate two graphs, often called the source and the target. Each grammar is a graph that consists of triple of subgraphs. One of the subgraphs corresponds to a structure that can be found in the source. Another of the subgraphs corresponds to a structure that can be found in the target. And the third subgraph connects these two halves with relational information. In its most basic application, a Triple Graph Grammar consists of a set of grammars, a grammar can be applied to one graph if its corresponding subgraph matches a structure in the source or target graph. In more advanced applications, the grammars are generated by graph matching searches. Therefore, there does not need to be a complete set of grammars that correspond to all permutations of structures in the source and target.

Triple Graph Grammars have several important features that apply to our transformation framework. First, the use of a grammar does not modify an existing graph. To generate a new graph from an existing one, match one end of each grammar to the source graph, and duplicate the target end of each grammar to generate a target graph. Second, grammars do not define an order of application, thus the application of different grammars can yield diverse results. Third, each grammar stores relational information that is bidirectional traceability information.

4.2 Code Generation

Code generation is the simplest of the processes that the transformation framework can be used for. It fundamentally consists of 3 steps. First, generate a TASM SDG as described in chapter 3. Second, apply a set of TGGs to transform the TASM SDG into a JSDG. And third, transform the JSDG into a Java program [Figure 4-1].

The Traceability Information that needs to be stored is the TASM SDG, applied TGGs, and JSDG.

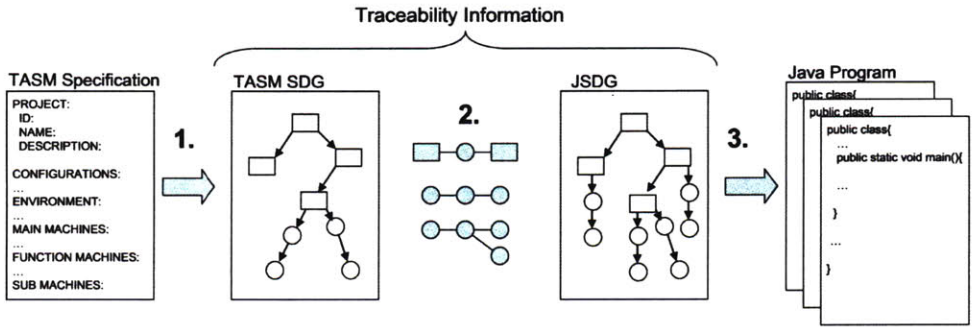

Figure 4-1: Code Generation

4.3 Reverse Engineering

Reverse engineering is the extraction of a model from code, and follows a similar procedure as code generation, but in reverse. First, a JSDG must be generated from the Java program. Second, a set of TGGs and a user defined set of TGGs is used to transform the JSDG to a TASM SDG. Finally, the inverse transform is used to generate a TASM specification from the TASM SDG [Figure 4-2].

The set of user specified transformations allows the user to identify states in the Java program by collapsing multiple variables into the domain of a smaller subset of variables. For example, a user may wish to specify that multiple variables related to thrust vector, force, and fuel mixture be simplified into a single variable that represents whether the rocket is performing nominally.

4.4 Synchronization

By definition, a model is at a higher level of abstraction than implementation code, and therefore should contains less information. However, models can also represent information implicit in the implementation. For example, timing and resource consumption models cannot be translated directly into code, but instead express system-level behavior.

As a result the generated code is just a template. As programmers modify the

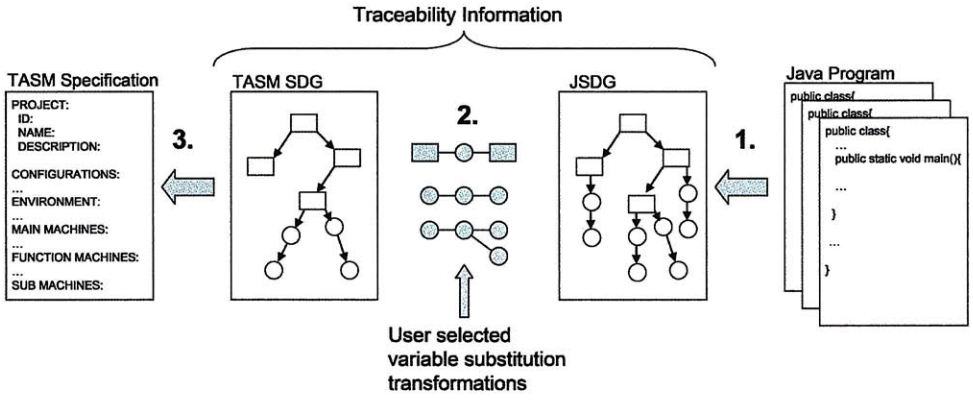

Figure 4-2: Reverse Engineering

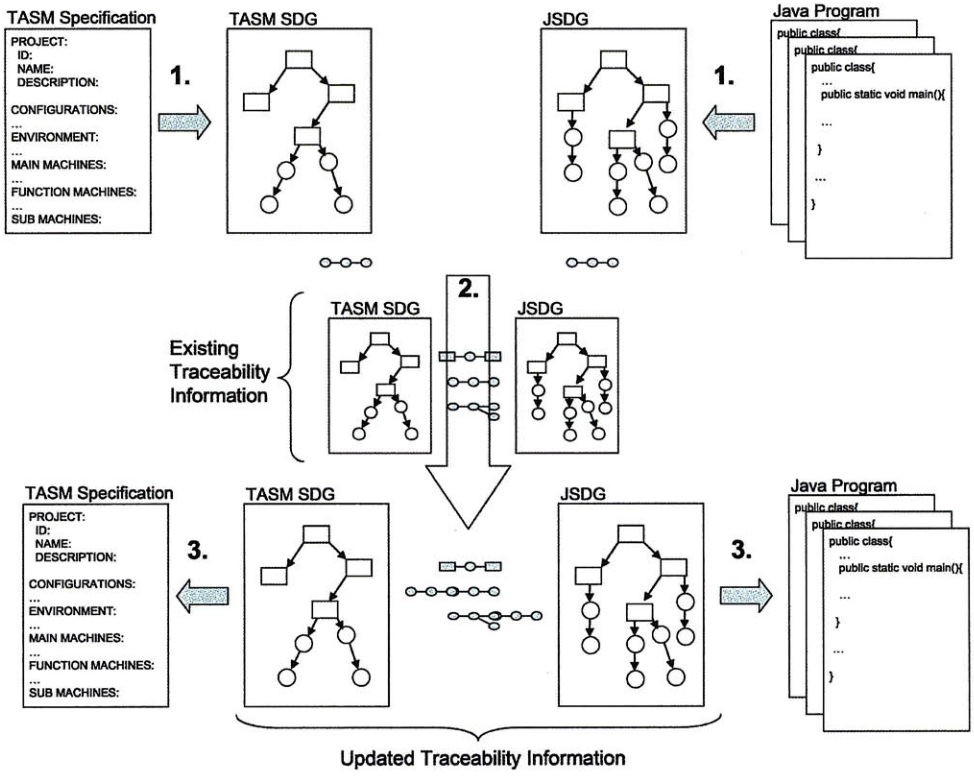

Figure 4-3: Synchronization

generated code (by introducing new variables, or speed/memory enhancing transformations, etc...) traceability connections stored in a tool or via code annotations can become outdated. The annotations can be deleted, or worse, preserved even though the surrounding code has had a fundamental functional change.

The transformation framework does not require the use of annotations, and instead relies on the presence of previous traceability information to operate. Figure 4-3 illustrates how the transformation framework can be used for synchronization. For simplicity, the process has been distilled into three steps. In the first step, a TASM SDG and a JSDG are generated from their corresponding documents. In the second step, each newly generated SDG is compared with their corresponding original SDG from the input traceability information. Changes between each pair of SDGs is reconciled by using rephrasing transformations/grammars (transformations in which the source and target language are the same). The rephrasing transformations can then be matched and merged with the original set of transformations. Rephrasing transformations can also add to or remove verticies from the original set of transformations. verticies added to the graph are tagged 'added' and vertices removed are tagged 'to be removed'. A graph matching algorithm must then complete the set of grammars [19]. Added verticies must have a new grammar added to associate that new vertex with one in the other SDG. Likewise, removed verticies belong to grammars in the original set of traceability information. Those grammars must be replaced by new grammars that account for the missing vertex.

Chapter 5

Case Study: Electronic Throttle Controller

5.1 Electronic Throttle Controller[1]

The Electronic Throttle Controller (ETC) is a "drive-by-wire" system that is currently in use at a major automotive company. The ETC was initially modeled by Griffiths [21] as a hybrid system using Mathworks' Simulink and Stateflow [31]. The ETC is used to optimize fuel consumption based on a set of criteria, including environmental conditions such as temperature and altitude, the state of the vehicle such as engine RPM and speed, and driver inputs such as cruise control and gas pedal angle [11]. The throttle controller is a piece of software which sits between the operator and the engine and replaces the mechanical linkage between the gas pedal and the engine throttle. The software interprets driver input and operating conditions, through sensors, to decide on the desired angle of the engine throttle for optimal fuel efficiency.

The throttle angle governs how much air can enter the engine and, consequently, how much power is produced by the engine. The relationship between throttle angle and fuel consumption is intuitive. The angle of the engine throttle determines how much air can go in the cylinder, and hence controls the volume of the charge. Con-

[1]This description of the ETC was originally written by Martin Ouimet. [33]

sequently, the throttle position governs the amount of torque produced. The fueling system is responsible for injecting an amount of fuel so that, immediately before combustion takes place, the Air-to-Fuel Ratio (AFR) is optimized. More specifically, the AFR should be stoichiometric (i.e., as close to 14.7:1 as possible for regular fuel) in order to allow for complete combustion, resulting in optimal efficiency. In order to optimize fuel efficiency, there are two main parameters to control: the angle of the throttle and the AFR as commanded by the fuel injectors. The ETC uses these two outputs to control the behavior of the engine.

Figure 5-1 shows the top level of the ETC model in Simulink, with the two key outputs – desired current (desired_current) and desired rate of fuel mass (dMfc). The angle of the throttle is controlled by the amount of current fed to the throttle servo. The desired current affects the position of the throttle and is determined based on the position of the gas pedal (as activated by the operator) and other external parameters (e.g., vehicle speed, O_2 concentration in the exhaust, engine speed, and temperature). The other controller output is the rate of fuel mass (dMfc). The dMfc value controls how much gas is sprayed in the combustion chamber. That value needs to be dynamically adjusted to maintain a stoichiometric combustion. The transfer function that characterizes the relationship between these two quantities (desired current and dMfc) is non-linear, and the model considered in this case study controls both factors independently.

The throttle controller uses modes to decide the control laws that govern the throttle actuation. For example, the throttle controller operates under different modes that have a priority ordering, depending on environmental conditions such as engine revolution, traction, cruise control settings, and driver input. The modes define the desired throttle angle, commanded through a current output from the throttle controller.

During nominal operation, the major modes of the controller are grouped into "driving modes" and "limiting modes". The limiting modes, defined as undesirable environment conditions, take precedence over driving modes. Limiting modes include "traction control", where the wheels rotate with too little friction, and "rev-

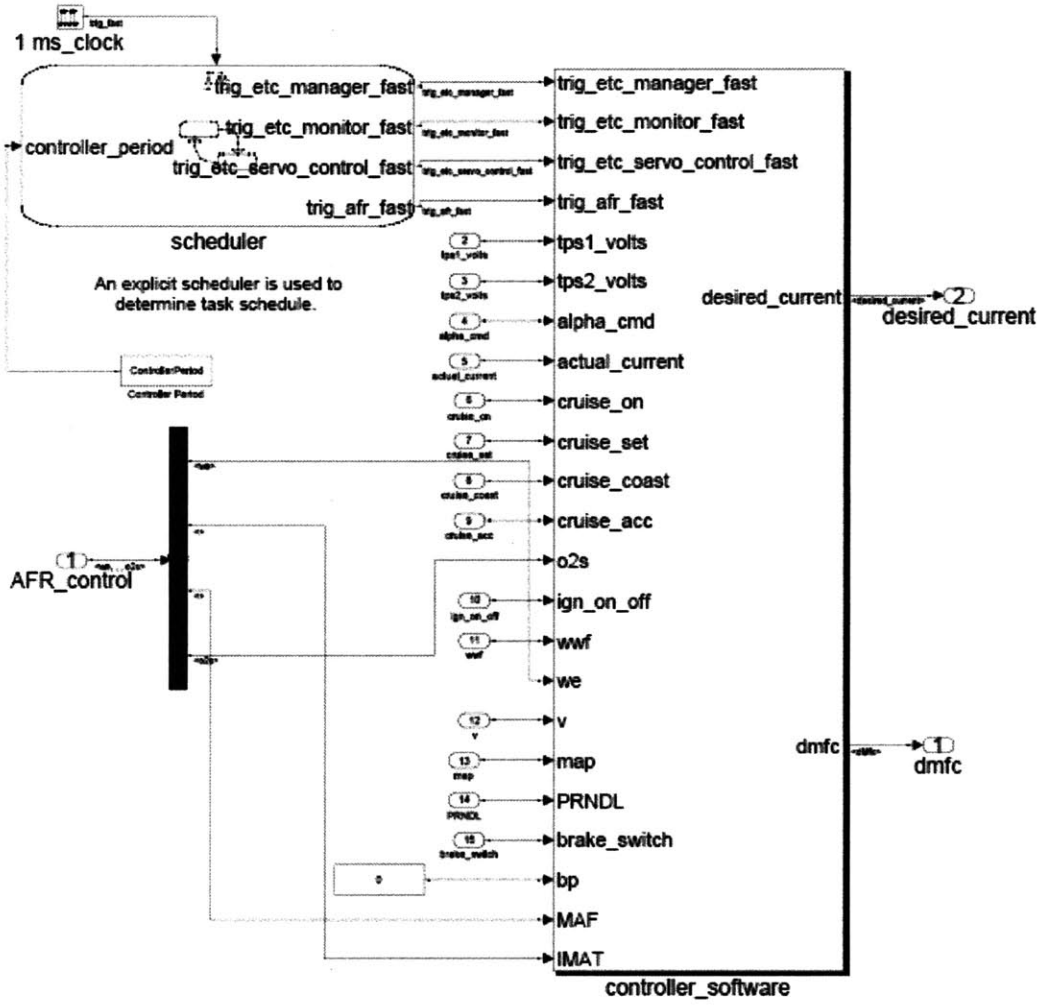

Figure 5-1: High level Simulink model of the ETC

olution control", where the engine operates over a predefined threshold of rotations per minute. The driving modes include "human control", where the throttle is commanded via the gas pedal, and "cruise control", where the throttle is commanded based on the desired vehicle speed. The different modes are shown in Figure 5-2, adapted from [21], represented visually as a Statechart variant. The "XOR" label indicates mutual exclusion between modes and the "AND" label indicates parallel composition of modes. The transitions to the "failure detected" mode are not shown in Figure 5-2 to keep the figure simple. In each mode, a transition to the "failure detected" is possible. The detection of failure takes precedence over all other modes and the behavior of the ETC is to gradually decrease the vehicle speed until shutdown is possible.

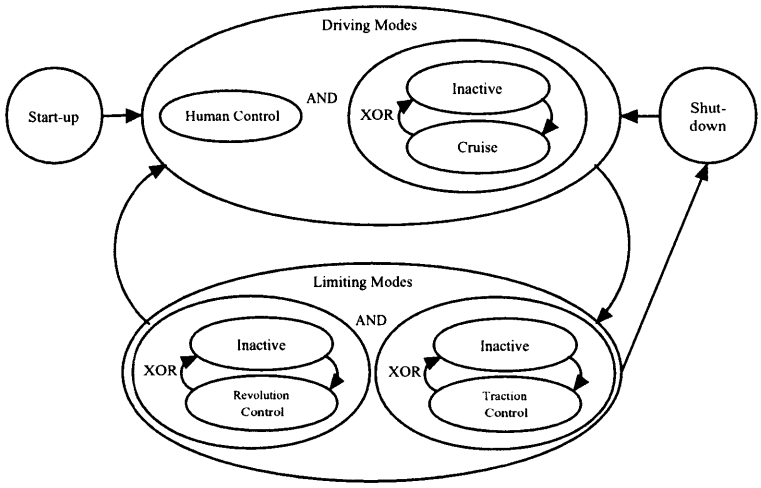

Figure 5-2: ETC modes

The modes of the throttle controller determine the desired throttle angle and, consequently, the amount of current output from the controller. The mode switching logic, as well as the calculation of the desired current represent the functional behavior of the ETC, dictating what the output should be based on various inputs. Because the calculation of the dMfc is completely isolated from the rest of the system, it is omitted from the case study.

The ETC represents an interesting case study for the proposed framework because the functional behavior is implemented using a set of tasks and a scheduler. The ETC

implementation is achieved using 3 tasks - a manager task, which sets the major and minor modes of the ETC, a monitor task, which periodically appraises the health of the system, and a servo task, which calculates and outputs the desired current based on the controller mode and the health of the system. The tasks have different periods and are driven by a scheduler with a 1 ms clock, as shown in Figure 5-3.

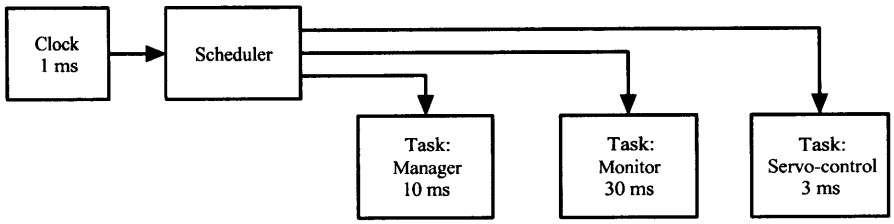

Figure 5-3: ETC tasks and scheduler

The scheduler does not support preemption and the tasks have fixed priority. The monitor task has the highest priority, followed by the monitor task, followed by the servo task.

5.2 Implementation

The ETC was originally implemented using three separate TASM models to exercise traceability within the modeling framework: One to model the scheduler and tasks without the functional aspects. Another to model the functional aspects alone. And a third, to integrate all the features together in what is referred to as a "low level" model. It is this "low level" model that we used to test code-generation [Appendix B].

The ETC case study is a useful demonstration of our ability to map model to code. It uses multiple main machines in parallel, with hierarchial composition of sub and function machines. It also utilizes floating point types, boolean types, and several user defined types, which test our ability to correctly map between the type system of TASM and Java. Boolean and user defined types were modeled as finite domain sets. This is reasonable because both Java and TASM do not allow the use of variables of these types as numbers. However, the use of floating point types in the ETC does

not fully test out the TASM SDG graph. Namely, floating point variables are only compared against constant values in inequalities, and their values are only updated through incrementing. The ETC also didn't use the TASM feature of communication channels. Even though the ETC lacks these features, it still fully tests the base algorithms used for mapping.

The TASM low-level model for the ETC was produced and tested for completeness and consistency through the TASM tool. The algorithms described in this thesis were implemented as a separate executable, via the Java 1.6 programming language. Parsing of both TASM and Java were implemented through ANTLR, an LL lexer-parser generator. Three methods were implemented for the calculation of the variable bounds as a result of effect expressions: interval analysis, min-max, and a combination of the two (for speed and robustness).

5.3 Results

All the tests were run under Windows, via the Java 1.6 update 16 runtime, using a single core of a 2.39Ghz quad core Intel Core 2 processor. The running times given for each of the tests were averaged over 3 runs.

5.3.1 Code Generation

The code generation tests were straightforward, and involved running a single executable with two parameters: the path to the TASM file modeling the low level controller and another option to select which bound calculation method was to be used.

Using interval analysis, it took 4.573 seconds to generate Java from TASM. Using pure min-max calculations, it took 6.737 seconds. And, using a combination of the two, which defaults to interval analysis whenever possible, it took 4.800 seconds. The resulting Java code in each case was identical, as expected [Appendix C].

Chapter 6

Conclusion

This research established a mapping from the TASM language to the Java language based on a transformation framework. The transformation framework made use of two intermediate forms: The TASM SDG, which is a graph-based semantic representation of a TASM specification, and a JSDG, the equivalent representation for a Java program. Bidirectional traceability was achieved by leveraging Triple Graph Grammars to describe the intermediate transformations between the two SDGs. By saving the applied grammars and the associated SDGs to which the grammar was matched, the traceability information could be preserved. It was shown how such information collected from the processes of code generation and reverse engineering could them be subsequently applied to synchronize or reconcile the difference between model and implementation.

The primary contributions of the thesis lie in the algorithms for creating the TASM SDG. One of which is generally applicable to bounded ASMs, and is useful for extracting a complete FSM from an ASM. In this thesis, that algorithm was applied to add the remaining, and complete set of, data dependency edges to the TASM SDG that could not be added by construction.

While the transformation framework does allow for a lot of flexibility, the use of TGG grammars make the graph matching process required for synchronization, in the worse case, to be NP complete. In theory, if the changes made to the code and/or implementation are incrementally synchronized, less graph matching to choose

the appropriate grammar to apply would be required, and synchronization could be significantly faster.

6.1 Future Work

Since this thesis primarily establishes a base of algorithms for extracting the information suitable for traceability, there is still a large body of work that can be done to add functionality.

6.1.1 Object Oriented Languages

The TGG profile presented can only generate procedural Java code. Furthermore, synchronization will only work if the modifications made to the code remain procedural. These limitations on code generation and synchronization exist so a single, connected, flow-dependency graph can be created for each program. The benefit of this approach is that code generation and synchronization become elegant operations involving graph transformations between two graphs. However, this limitation greatly strips the expressiveness of Java to something more akin to C. Namely, it removes the benefits of object oriented design.

The ability to map between ASMs and a fully supported object oriented language would increase the practicality of this research.

The first hurdle in adding this extension would be to address the lack of objects in ASM theory. There are many ways this can be solved. The simple solution would be to add the concept of compound data structures to TASM. Another solution could be to create a machine to model each class.

The second hurdle is representing non-static, or object-scoped methods. Because a potentially boundless number of objects can be created, a boundless number of environments could be created for the object-scoped methods to operate on. One way to solve this problem is to have an upper bound on the number of unique objects that can be created from a given class. This approach is popular in creating real-time subsets of languages to limit memory usage. In TASM, a main machine would then

Appendix A

TASM Language Reference

Authored By: Martin Ouimet, Edited By: David Wang

This document was originally prepared by Martin Ouimet. The Bakus-Naur form of the grammar has been corrected to more closely reflect the implemented interpreter. Additional non-terminals were also introduced to complete the abstract syntax tree required for the construction of the TASM SDG.

This appendix explains the concrete constructs of the TASM language as implemented in the TASM toolset. This appendix can be consulted as a supplement to Chapter 2. More specifically, this appendix describes the logical objects that make up the TASM language in the toolset, the rules for constructing names, the list of reserved keywords, the list of operators, and the general typing rules. Furthermore, the context-free grammar of the TASM language, presented in Section A.2, has been used to implement the compiler for the TASM toolset. Semantic implementation topics, such as operator precedence and calling convention, are explained in Section A.3.

A.1 TASM Objects

The concepts described in Chapter 2 are implemented in a suite of logical objects in the TASM language. This section gives the list of logical objects and their properties, as implemented in the TASM toolset. The concrete syntax of how these objects are

expressed is described in Section A.2.

A.1.1 Specification

In the logical objects, a *specification* is the overarching concept or object that includes all other logical objects. A *specification* is the complete document that results from capturing a system design in a model expressed in the TASM language.

A.1.2 Project

The *project* is the top level object that contains the high level metadata of the system specification. The project has three attributes, the project *name*, the project *description*, and the *version* of the syntax. The name and description are self-explanatory. The version of the syntax is used to identify older versions of the syntax to preserve backwards compatibility. Other attributes of the project that might be added in the future might include modification times, authors, etc. There is only one project object per specification.

A.1.3 Environment

The *environment* is the object that is used to represent the "outside world". The environment object contains the list of *user-defined types*, which are finite enumerations, the list of *resources*, which are finite quantities, and the list of *variables*, which are the values that affect and are affected by the execution of the various machines in the specification. The environment is a global object that is accessible by all machine instances.

A.1.4 Main Machine Template

In the TASM language, a main machine definition is a template. A machine template is a parameterized version of a machine that needs to be instantiated through a constructor given as part of the template definition. The use of templates enables

reuse of specifications and the ability to have multiple versions of a machine definition for a given system design. The concept of a machine template is analogous to the concept of a class in object-oriented programming languages [40]. Main machine templates are instantiated in a Configuration object.

A main machine template contains three attributes – a set of internal variables, a constructor, and a set of rules. The internal variables are typed variables that are visible only inside the machine. The constructor is used to initialize the machine through instantiation and to assign default values to internal variables. The set of rules is a set of guarded commands that govern the machine execution and its effects on the environment. Each rule also specifies the duration of the rule application and the resources consumed during execution of the rule, according to the principles explained in Chapter 2. Additional attributes of a main machine template include a set of monitored variables and a set of controlled variables. The set of monitored variables is the list of environment variables that are used in the guarding conditions of the rules. The set of controlled variables is the list of environment variables that are used in the effect conditions of the rules.

The main machines are the top-level abstract machines that represent a thread of execution. If more than one main machine is present in a system configuration, the resultant specification contains parallelism, also called a *multi-agent* ASM in the Abstract State Machine community [10]. Instantiating multiple main machines is the way to obtain parallel composition of specifications with interleaving semantics.

A.1.5 Function Machine

The idea behind a function machine is a machine with no side-effects that can be used to define abstractions and macros. A function machine is a machine that takes a set of typed inputs and returns a single typed output. A function machine contains a set of input variables and a single output variable. The set of input variables are typed variables that are used to invoke the machine. The output variable is a typed variable that is used to return a value from the machine when it is invoked. A function machine is not allowed to modify the environment and must compute its output solely

based on the input values and the values of its monitored variables.

A.1.6 Sub Machine

A sub machine is similar to a main machine except that the sub machine does not execute in its own thread of execution. Instead, a sub machine executes inside of a main machine and shares the thread of execution of the main machine. Sub machines are used to achieve hierarchical composition. A main machine can use more than one sub machine as part of its definition. A sub machine definition contains the same attributes as a main machine except that it does not contain internal variables nor does it contain a constructor.

A.1.7 Configuration

A configuration corresponds to a simulation scenario. A configuration contains a name and a description so that it can be referenced during simulation. A configuration also contains a list of main machine instantiations, defined by invoking the constructors of the main machine templates. Furthermore, the configuration can contain initial values for the environment variables. The initial values specified in a configuration override the initial values of environment variables specified in the environment. Multiple configurations can exist for a given project and a configuration must be selected to perform simulation.

A.2 Syntax

This section describes the concrete syntax of the TASM language, expressed in plain-text format. The plain-text syntax is the format used to read and write specifications using the TASM toolset and it is the input format for the parser and compiler. In the TASM toolset, a system design can be shown across different windows and other user interface components and does not need to be gathered into a single location.

A.2.1 Notational Conventions

The following notational conventions are used in this section and subsequent sections.

- Each abstract type uses the prefix *TASM*
- Constants are enclosed in single quotes (e.g. 'a', '1', etc.), except where set-theoric notation is used
- The formal grammar uses the basic symbols of Backus-Naur Form (BNF) [4] (e.g., {}, [], <>, etc.)

A.2.2 Names

The use of names is crucial in the TASM language; every type of object (variable, type, resource, machine, etc.) is uniquely identified by its name. We define the generic abstract type *TASMName* to express the restrictions on individual names. The type *TASMName* is used in the rest of this document when a name has the listed restrictions. The TASM language has a set of reserved keywords that cannot be used as names. The complete list of reserved keywords is shown in table A.1.

- TASMName is a string of characters
- Each character of TASMName can be either 'a'-'z' or 'A'-'Z' or '_' or '1'-'9' or '-'
- TASMName must start with 'a'-'z' or 'A'-'Z'
- TASMName has a length: 1-64
- TASMName is not a reserved keyword
- TASMName is case-sensitive
- Each TASMName is unique in a given TASM specification

The restrictions on the uniqueness of TASMName's might seem restrictive, especially in the absence of namespaces, but imposing this restriction removes potential ambiguities.

Table A.1: Reserved keywords

Keyword	Meaning
`t`	Used for time annotations
`next`	Used in time annotations to denote a special value of time
`now`	Used to obtain the value of the global clock
`new`	Used to instantiate a machine template
`Integer`	Denotes the integer datatype
`Float`	Denotes the float datatype
`Boolean`	Denotes the Boolean datatype
`False`	Denotes a constant in the Boolean datatype
`True`	Denotes a constant in the Boolean datatype
`and`	Denotes a logical connective
`or`	Denotes a logical connective
`not`	Denotes a unary operator
`skip`	Denotes the production of an empty update set
`else`	Denotes the special "else rule"
`//`	Used to comment out a given line

A.2.3 Types

The TASM language contains only simple types. There are no data structures, sub-types, or polymorphic types. The TASM language is also strongly typed; there are no dynamic types or type inference. All typing rules are enforced at compilation time and type safety is assured if a TASM specification compiles correctly. The TASM language supplies three default types:

- Integers = $\{\ldots, -1, 0, 1, \ldots\}$
- Floats = *Rational Numbers* (e.g., -1.11, -0.5, 0.0, 10.45, etc.)
- Booleans = $\{True, False\}$

TASM also allows the definition of user-defined types, which are analogous to *enumerations* in most programming languages. However, user-defined types are not assigned integer values and are unordered. A user-defined type is a named type that can be used to provide readable options and type safety. More specifically, a user-defined type is a named type that contains one ore more named values. For example, user-defined types can be defined to denote the status of a light status or the mode of an airplane:

- light_status = {ON, OFF}
- airplane_mode = {Idle, Taxi, Takeoff, Cruise, Landing}

User-defined types are unordered sets of one or more elements where elements must be unique. Each member element is a TASMName. Furthermore, the name of the type is a TASMName.

The TASM language is a strongly typed language, meaning that all variables are typed and that type-safety is enforced at compilation time. No type casting is allowed, even from *Float* to *Integer*. Future versions of the language might allow type casting through functions supplied by the TASM language.

A.2.4 Arithmetic Operators

For *Integer* and *Float* types, the TASM language provides the four basic arithmetic operators, applicable only to operands of the same type:

- addition: $+$
- subtraction: $-$
- multiplication: $*$
- division: $/$

Operations between operands of disparate types is undefined and results in a compilation error. For example, addition between an operand of type *Float* or an operand of type *Integer* results in a compilation error. The arithmetic operators are undefined for Boolean types and for user-defined types.

The assignment operator is the only operator which is defined for all types. Like for the other arithmetic operators, the assignment operator is defined only for operands of the same type:

- assignment: $:=$

The assignment operator does not return a value (denoted by the special character '$\perp$').

A.2.5 Logical Operators

The following two logical operators are defined for all types. The signature of the operators is $Type1 \times Type2 \rightarrow Boolean$ where $Type1 = Type2$. Operators applied to operands of different types are undefined and result in a compilation error.

- equal: $=$
- not equal: $!=$

For *Integer* and *Float* types, the TASM language supplies an additional four logical operators:

- greater than: >
- greater than or equal to: >=
- less than: <
- less than or equal to: <=

The signature of these operators is also $Type1 \times Type2 \rightarrow Boolean$ where $Type1 = Type2$. The logical operators are undefined when the operators are of different types or for *Boolean* and *user-defined* types. For *Boolean* types, the TASM language provides two logical connectives:

- conjunction: *and*
- disjunction: *or*

The signature of these operators is $Boolean \times Boolean \rightarrow Boolean$ and is undefined for non *Boolean* types. For *Boolean* types, the TASM language supplies one unary operator:

- negation: *not*

The signature of this operator is $Boolean \rightarrow Boolean$ and is undefined for non *Boolean* types.

All operators are summarized in Table A.2.

A.2.6 Context-Free Grammar

The following section explains the formal grammar that is used to express the basic concepts from the previous section, such as types, constants, variables, expressions, etc. The formal grammar is given in Backus-Naur Form (BNF) [4] where the syntactic symbol '|' means "or", '[]' means "optional", and '{}' means 0 or more instances

Table A.2: Operators

Operator	Signature	Types
`+`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`-`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`*`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`/`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`:=`	$Type1 \times Type2 \rightarrow \perp$, $Type1 = Type2$	All
`=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	All
`!=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	All
`>`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`>=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`<`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`<=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`and`	$Boolean \times Boolean \rightarrow Boolean$	Boolean
`or`	$Boolean \times Boolean \rightarrow Boolean$	Boolean
`not`	$Boolean \rightarrow Boolean$	Boolean

(Kleene closure). The special form of the closure operator, denoted '$\{\}^+$' means 1 or more instances. Any constant is given inside of single quotation marks. For example, the keyword denoting the type integer is given as 'Integer'. It is important to distinguish between the syntactical "optional symbol" ']' and the constant denoting the right bracket "]".

The BNF grammars describing the concepts of the TASM language is given below. The first part of the grammar supplies the rules for constructing names, constants, and types. The second part of the grammar supplies the rules for constructing expressions, variables, and formulas. The TASM language ignores whitespace unless whitespace is required. When whitespace is required, it is denoted by the token $< TASMWhitespace >$, which represents a single whitespace character. Tab characters, space characters, new line characters, carriage return characters, and form feed characters all represent a single whitespace character.

A.2.6.1 Basic Concepts

$$
\begin{aligned}
\text{< TASMUCaseLetter >} &::= 'A' \mid 'B' \mid \ldots \mid 'Z' \\
\text{< TASMLCaseLetter >} &::= 'a' \mid 'b' \mid \ldots \mid 'z' \\
\text{< TASMLetter >} &::= \text{< } TASMUCaseLetter \text{ > } \mid \text{< } TASMLCaseLetter \text{ >} \\
\text{< TASMDigit >} &::= '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' \\
\text{< TASMCharacter >} &::= \text{< } TASMLetter \text{ > } \mid \text{< } TASMDigit \text{ > } \mid '\_' \\
\text{< TASMASCIIChar >} &::= \text{All standard ASCII characters} \\
\text{< TASMWhiteSpaceChar >} &::= ' ' \mid '\backslash t' \mid '\backslash n' \mid '\backslash r' \mid '\backslash f' \\
\text{< TASMWhiteSpace >} &::= \{\text{< } TASMWhiteSpaceChar \text{ >}\}^{+} \\
\text{< TASMIntLit >} &::= ['-'] \text{< } TASMDigit \text{ > } \{\text{< } TASMDigit \text{ >}\} \\
\text{< TASMFloatLit >} &::= ['-'] \text{< } TASMDigit \text{ > } \{\text{< } TASMDigit \text{ >}\} \\
&\quad\ '.' \text{< } TASMDigit \text{ > } \{\text{< } TASMDigit \text{ >}\} \\
\text{< TASMBooleanLit >} &::= 'True' \mid 'False' \\
\text{< TASMStringLit >} &::= \{\text{< } TASMASCIIChar \text{ >}\} \\
\text{< TASMName >} &::= \text{< } TASMLetter \text{ > } \{\text{< } TASMCharacter \text{ >}\} \\
\text{< TASMDescription >} &::= \text{< } TASMStringLit \text{ >} \\
\text{< TASMVariable >} &::= \text{< } TASMName \text{ >}
\end{aligned}
$$

< **TASMUDTypeName** > ::= $< TASMName >$

< **TASMTypeName** > ::= $'Integer' \mid 'Float' \mid 'Boolean' \mid < TASMUDTypeName >$

< **TASMUDTypeMember** > ::= $< TASMName >$

< **TASMUDTypeDef** > ::= $< TASMUDTypeName >':='$
$'\{'< TASMUDTypeMember > \{',' < TASMUDTypeMember >\}'\}'';'$

< **TASMConstant** > ::= $< TASMIntLit > \mid < TASMFloatLit > \mid$
$< TASMBooleanLit > \mid < TASMUDTypeMember >$

< **TASMValue** > ::= $< TASMVariable > \mid < TASMConstant >$

< **TASMMachineName** > ::= $< TASMName >$

< **TASMFMachineCall** > ::= $< TASMMachineName >' ('[< TASMArithExpr > \{',' < TASMArithExpr >\}]')'$

< **TASMValueExpr** > ::= $< TASMValue > \mid < TASMFMachineCall > \mid 'now'$

< **TASMArithOp** > ::= $' +' \mid ' -' \mid ' *' \mid '/'$

< **TASMArithExpr** > ::= $< TASMValueExpr > \mid$
$< TASMArithExpr >< TASMArithOp >< TASMArithExpr > \mid$
$'(' < TASMArithExpr >< TASMArithOp >< TASMArithExpr >')'$

< **TASMBinLogicOp** > ::= $' >=' \mid ' >' \mid ' <=' \mid ' <' \mid ' =' \mid '! ='$

< **TASMLogicExpr** > ::= $< TASMBooleanLit > \mid$
$< TASMArithExpr >< TASMBinLogicOp >< TASMArithExpr > \mid$
$'(' < TASMArithExpr >< TASMBinLogicOp >< TASMArithExpr >')'$

< **TASMLogicBinConn** > ::= $'and' \mid 'or'$

< **TASMLogicUnConn** > ::= $'not'$

< **TASMLogicFormula** > ::= < *TASMLogicExpr* > |
< *TASMLogicFormula* >< *TASMLogicBinConn* >< *TASMLogicFormula* > |
< *TASMLogicUnConn* >< *TASMLogicFormula* > |
'('< *TASMLogicExpr* >')' |
'('< *TASMLogicFormula* >< *TASMLogicBinConn* >< *TASMLogicFormula* >')' |
'('< *TASMLogicUnConn* >< *TASMLogicFormula* >')'

< **TASMExpr** > ::= < *TASMArithExpr* > | < *TASMLogicExpr* > | < *TASMLogicFormula* >

< **TASMVarDecl** > ::= < *TASMTypeName* >< *TASMWhitespace* >< *TASMVariable* >';'

< **TASMVarDeclInit** > ::= < *TASMTypeName* >< *TASMWhitespace* >< *TASMVariable* >
' :='< *TASMConstant* >';'

< **TASMNameDescPair** > ::= '*NAME* :'< *TASMName* >< *TASMWhitespace* >
'*DESC* :'< *TASMDescription* >

< **TASMVarInit** > ::= < *TASMVariable* >':='< *TASMConstant* >';'

A.2.6.2 Environment

< **TASMResourceName** > ::= < *TASMName* >

< **TASMResourceDef** > ::= < *TASMResourceName* >':=" ['< *TASMIntLit* >','< *TASMIntLit* >']";'

< **TASMChannelName** > ::= < *TASMName* >

< **TASMChannelDef** > ::= '*Channel*' < *TASMWhiteSpace* >< *TASMChannelName* >';'

< **TASMEnvironDef** > ::= '*ENVIRONMENT* :'< *TASMEnvTypeDef* >
< *TASMEnvVarDef* >< *TASMEnvChannelDef* >
< *TASMEnvResourceDef* >

$$< \textbf{TASMEnvTypeDef} > \quad ::= \quad 'TYPES :' \{< TASMUDTypeDef >\}$$

$$< \textbf{TASMEnvVarDef} > \quad ::= \quad 'VARIABLES :' \{< TASMVarDeclInit >\}$$

$$< \textbf{TASMEnvChannelDef} > \quad ::= \quad 'CHANNELS :' \{< TASMChannelDef >\}$$

$$< \textbf{TASMEnvResourceDef} > \quad ::= \quad 'RESOURCES :' \{< TASMResourceDef >\}$$

A.2.6.3 Project

$$< \textbf{TASMProjectDef} > \quad ::= \quad 'PROJECT :'< TASMNameDescPair >$$

A.2.6.4 Machine Templates

$$\begin{aligned} < \textbf{TASMTemplateDef} > \quad ::= \quad & 'TEMPLATES :'< TASMMTemplatesDef > \\ & < TASMSTemplatesDef > \\ & < TASMFTemplatesDef > \end{aligned}$$

$$< \textbf{TASMMTemplatesDef} > \quad ::= \quad 'MAIN\ MACHINES :' \{< TASMMTemplateDef >\}$$

$$< \textbf{TASMSTemplatesDef} > \quad ::= \quad 'SUB\ MACHINES :' \{< TASMSTemplateDef >\}$$

$$< \textbf{TASMFTemplatesDef} > \quad ::= \quad 'FUNCTION\ MACHINES :' \{< TASMFTemplateDef >\}$$

A.2.6.5 Syntax Common to all Machines

$$
\begin{aligned}
&\textbf{< TASMVariableList >} ::= \{< TASMVariable >';'\}\\
&\textbf{< TASMRuleName >} ::= < TASMName >\\
&\textbf{< TASMRule >} ::= < TASMRuleName >' \{'\\
&\qquad [< TASMTimeSpec >]\{< TASMResourceSpec >\} < TASMRuleDef >'\}'\\
&\textbf{< TASMTimeSpec >} ::= 't'':='' ['< TASMIntLit >',' < TASMIntLit >']'';' \mid\\
&\qquad 't'':='< TASMIntLit >';' \mid\\
&\qquad 't'':='' next'';' \mid\\
&\qquad 't'':='' dt'';'\\
&\textbf{< TASMResourceSpec >} ::= < TASMResourceName >':='' ['< TASMIntLit >',' < TASMIntLit >']'';' \mid\\
&\qquad < TASMResourceName >':='< TASMIntLit >';'\\
&\textbf{< TASMRuleDef >} ::= < TASMRuleGuard >< TASMWhiteSpace >< TASMRuleEffect >\\
&\textbf{< TASMRuleGuard >} ::= 'if' < TASMLogicFormula >' then'\\
&\qquad \mid 'else' < TASMWhiteSpace >' then'\\
&\textbf{< TASMRuleEffect >} ::= \{< TASMEffectExpression >\}^{+}\\
&\textbf{< TASMEffectExpression >} ::= < TASMAssignment > \mid < TASMSubMachineCall > \mid\\
&\qquad < TASMChannelExpr > \mid 'skip;'\\
&\textbf{< TASMAssignment >} ::= < TASMVariable >':='< TASMArithExpr >';'\\
&\textbf{< TASMSubMachineCall >} ::= < TASMMachineName >' (')'';'\\
&\textbf{< TASMChannelExpr >} ::= < TASMChannelName >< TASMChannelOpChar >';'\\
&\textbf{< TASMChannelOpChar >} ::= '?' \mid '!'
\end{aligned}
$$

A.2.6.6 Main Machine

$$
\begin{aligned}
&< \textbf{TASMMTemplatesDef} > \quad ::= \quad 'MAIN\ MACHINE :' < TASMNameDescPair > \\
&\qquad\qquad\qquad\qquad\qquad\quad < TASMMVars >< TASMConstr >< TASMRules > \\
&< \textbf{TASMMVars} > \quad ::= \quad < TASMContVars >< TASMMonVars >< TASMIntVars > \\
&< \textbf{TASMContVars} > \quad ::= \quad 'CONTROLLED\ VARIABLES :' < TASMVariableList > \\
&< \textbf{TASMMonVars} > \quad ::= \quad 'MONITORED\ VARIABLES :' < TASMVariableList > \\
&< \textbf{TASMIntVars} > \quad ::= \quad 'INTERNAL\ VARIABLES :' \{< TASMVarDeclInit >\} \\
&< \textbf{TASMConstr} > \quad ::= \quad 'CONSTRUCTOR :' < TASMMachineName >' ('[< TASMParamList >]')''\{' \\
&\qquad\qquad\qquad\qquad\qquad\quad \{< TASMVarInit >\}'\}' \\
&< \textbf{TASMParamList} > \quad ::= \quad < TASMParam > \{',' < TASMParamList >\} \\
&< \textbf{TASMParam} > \quad ::= \quad < TASMTypeName >< TASMWhiteSpace >< TASMVariable > \\
&< \textbf{TASMRules} > \quad ::= \quad 'RULES :' < TASMRule > \{< TASMWhiteSpace >< TASMRule >\}
\end{aligned}
$$

A.2.6.7 Sub Machine

$$
\begin{aligned}
&< \textbf{TASMSTemplateDef} > \quad ::= \quad 'SUB\ MACHINE :' < TASMNameDescPair > \\
&\qquad\qquad\qquad\qquad\qquad\quad < TASMSVars >< TASMRules > \\
&< \textbf{TASMSVars} > \quad ::= \quad < TASMMonVars >< TASMContVars >
\end{aligned}
$$

A.2.6.8 Function Machine

$$
\begin{aligned}
&< \textbf{TASMFTemplateDef} > \quad ::= \quad 'FUNCTION\ MACHINE :'< TASMNameDescPair > \\
&\qquad\qquad\qquad\qquad\qquad\quad < TASMFVars >< TASMRules > \\
&< \textbf{TASMFVars} > \quad ::= \quad < TASMInVars >< TASMOutVars >< TASMIntVars > \\
&< \textbf{TASMInVars} > \quad ::= \quad 'INPUT\ VARIABLES :' \{< TASMVarDecl >\} \\
&< \textbf{TASMOutVars} > \quad ::= \quad 'OUTPUT\ VARIABLE :'< TASMVarDecl >
\end{aligned}
$$

A.2.6.9 Configurations

$$
\begin{aligned}
&< \textbf{TASMConfigurations} > \quad ::= \quad 'CONFIGURATIONS :' \{< TASMConfiguration >\} \\
&< \textbf{TASMConfiguration} > \quad ::= \quad 'CONFIGURATION :'< TASMNameDescPair > \\
&\qquad\qquad\qquad\qquad\qquad\quad < TASMConfMInit >< TASMConfVarInit > \\
&< \textbf{TASMConfMInit} > \quad ::= \quad 'MACHINE\ INITIALIZATIONS :' \{< TASMMachineInstance >\} \\
&< \textbf{TASMMachineInstance} > \quad ::= \quad < TASMName >':=''\ new' < TASMMachineName > \\
&\qquad\qquad\qquad\qquad\qquad\quad '('[< TASMConstant > \{',' < TASMConstant >\}]')'';' \\
&< \textbf{TASMConfVarInit} > \quad ::= \quad 'VARIABLE\ INITIALIZATIONS :' \\
&\qquad\qquad\qquad\qquad\qquad\quad \{< TASMVarInit >\}
\end{aligned}
$$

A.3 Semantics

Three dominant approaches stand out when expressing programming language semantics - operational semantics, denotational semantics, and axiomatic semantics. Denotational semantics has been used successfully for sequential programs, but the

paradigm becomes difficult to work with when concurrency is introduced. Axiomatic semantics has been used on smaller programs, but it is not clear that it works well for larger programs or for languages with numerous concepts. Operational semantics could be used to express the TASM semantics because it has concepts analogous to the TASM language, namely, that of an abstract machine progressing through configurations. Operational semantics has been used extensively to specify language semantics, for both sequential and concurrent programs. However, because the ASM paradigm is close to the operational semantics paradigm, an attempt is made to express the semantics of the TASM language using Abstract State Machines (ASM). The motivation is twofold. First, ASMs have been used to specify the semantics of executable languages, including VHDL, Prolog, and SDL. Second, because the TASM language is built on top of the ASM language, it makes sense to use ASM to express the semantics. In a sense, if the semantics are expressed properly, the TASM language could be viewed as "syntactic sugar" on top of the ASM language.

A.3.1 Operator Precedence

The use of parentheses is strongly encouraged to disambiguate operator precedence for language users. However, the TASM language defines rules for operator precedence when parentheses are not used. The precedence rules are listed in Table A.3.

A.3.2 Calling Convention

In the TASM language, all function ASM calls machine instantiations, and variable references use "call-by-value" semantics. There are no pointers or no references in the TASM language, only distinct variables. Machine instances are all different from one another. When variables are assigned to each other, the value gets copied over to the assigned variable. No variable can be "linked" to the same value, using "pointer-like" semantics.

Table A.3: Operator precedence

Operator	Meaning
$*$	Multiplication
$/$	Division
$+$	Addition
$-$	Subtraction
$>=$	Greater than or equal to
$>$	Greater than
$<=$	Less than or equal to
$<$	Less than
$=$	Equal to
$!=$	Not equal to
and	Logical connective 'AND'
or	Logical connective 'OR'
not	Logical negation 'NOT'
$:=$	Assignment

A.3.3 Types

All variables are strongly typed in the TASM language. There is no type-casting and operators are defined only for operators of the same type. The type checking ensures that all operations are type safe at compile-time. Syntactically, decimal values are interpreted with a required "decimal part", which is a period (".") followed by a digit. This is required even from decimal numbers without a decimal part (e.g., 9.0). "9.0" and "9" are constants of different types, namely the first one is of type "Integer" while the second one is of type "Float". There is no type inference or dynamic typing of any sort as all variables are statically typed and cannot be type casted.

A.3.4 Relation to Abstract State Machines

In this section, the execution semantics of the TASM language are formally expressed using ASM. This is accomplished by using Abstract State Machines (ASM), using the syntax from the Lipari guide [22]. The aim of this section is to express the semantics of the extended language using a "desugaring" into the syntax of the Lipari guide. For the syntax, we follow the notational conventions used in both the Lipari guide and the definition of the formal semantics of SDL [17]. For a detailed list of the ASM syntax used to express formal semantics, the reader is referred to the SDL guide [17], pages 25–27. The translation given in this section is similar except that the ASM version used in this section uses the Lipari guide syntax, which is closer to the classical definition of ASM.

The key extensions to the TASM language have to do with the addition of time passage and resource consumption. To illustrate time passage, the same conventions as in [15, 17] are adopted and a global dynamic and monotonic increasing function is introduced, called *currentTime*:

- **external** *currentTime*: $\rightarrow$ *REAL*

This function is used inside of machines to query the value of the current time. The function is modified by the environment only and returns a monotonically increasing value greater than 0.0.

A.3.5 Sugaring/Desugaring

The extensions to the TASM language have been introduced as "syntactic sugar" on top of the syntax and semantics of the ASM language as expressed in the Lipari guide [22]. In order to map a TASM specification into an ASM specification, two domains are introduced, namely $DTASM$ and $DASM$ to denote the domains of specifications expressed in the TASM language and the ASM language respectively. A function called $Desug$ is also introduced. This function maps a TASM specification into an ASM specification. The "desugaring" function is defined for all individual elements of the TASM language (specifications, variables, types, rules, etc.) and maps the TASM elements into elements of the ASM language.

- $Desug : DTASM \rightarrow DASM$

A.3.6 Resource definitions

A resource definition, $Rdef$, in the environment is desugared into a global shared dynamic function:

- $Desug[[Rdef]] =$ **shared** $Rdef$

The desugaring of the resource definition is a bit more complex with respect to usage, but the execution semantics of resource usage are detailed in sectionA.3.10.

A.3.7 Type definitions

Type definitions, $Tdef$ get desugared into static finite domains:

- $Desug[[Tdef]] =$ **static domain** $Tdef$

A.3.8 Variables

Controlled and monitored variables inside of machines get desugared into nullary controlled and dynamic functions, respectively.

A.3.9 Rules

The desugaring of the rules is the most complex desugaring in the TASM language, because this is where time and resource utilization play a role. To illustrate the desugaring of rules, an abstract syntax for a rule definition is defined:

- $Rules = (R_i^+)$
- $R_i = (t_i\ r_i^*\ if\ cond_i\ then\ effect_i)$

In the TASM, the set of rules for a given machine is implicitly mutually exclusive. In the ASM language, the mutual exclusion is explicit. The first desugaring of as set of rules is to generate the explicit mutual exclusion:

- $Desug[[Rules]] = Desug[[(t_0\ r_0^*\ if\ cond_0\ then\ effect_0)\ ...\ (t_n\ r_n^*\ if\ cond_n\ then\ effect_n))]] =$

  if $cond_0$ **then** $effect_0$
  else if $cond_1$ **then** $effect_1$
  ...
  else if $cond_n$ **then** $cond_n$

The *else* rule guard from the TASM language would get desugared into a simple **else** rule guard of the ASM language. The time annotations get desugared into an environment variable that affects each machine's execution to simulate "durative" actions. Conceptually, once a rule is triggered, a machine sets a specific variable to the duration of the rule application and will not do anything until the rule duration has elapsed. Once the rule duration has elapsed, the machine will generate the appropriate update set atomically and will be free to execute another rule. Desugaring a time annotation for a rule introduces a new branch if the "if" conditions to denote the time. The concept of a "fresh" variable is introduced to denote a newly generated variable whose name is not previously used. The desugaring introduces two variables, one to keep the time when the rule application will finish executing and one to denote that

the machine is "busy" doing work. These two variables are denoted by $tcomplete_{fresh}$ and $mbusy_{fresh}$. The $fresh$ underscore is used to indicate that the variable name is introduced by the desugaring and enforces that it does not clash with existing names. Both of these variables also desugar into controlled dynamic functions:

- $Desug[[tcomplete_{fresh}]]$ = **controlled** *tcomplete* initially *-1*
- $Desug[[mbusy_{fresh}]]$ = **controlled** *mbusy* initially *False*
- $Desug[[Rule]] = Desug[[(t_i\ r_i^*\ if\ cond_i\ then\ effect_i)]] =$

  if/else if $cond_i \wedge mbusy_{fresh} = False$ **then**
  $mbusy_{fresh} := True$, $tcomplete_{fresh} := currentTime + getDuration(t_i)$
  else if $currentTime = tcomplete_{fresh} \wedge mbusy_{fresh} = True$ **then**
  $effect_i$, $mbusy_{fresh} := False$, $timcomplete_{fresh}$:= -1
  ...

The function *getDuration* is a macro that is created using the condition and the time annotation of the rule. It returns the duration of the rule. If the time annotation is a single value, it returns that value. Otherwise, if the rule annotation is an interval, it returns a value non-deterministically selected from the interval. Using a macro will enable the desugaring to take into account possible concurrency semantics like WCET and BCET as defined in section **??**. The introduction of the two auxiliary variables and the time conditions will guarantee that the machine will not produce any update sets and that no other rules will be enabled while the machine is executing a rule. This behavior is exactly the desired behavior to simulate "durative" actions.

Resource annotations get desugared as well, but their usage is a bit different than for the time annotations. Resources are modeled as shared dynamic functions. Their values are set during at the beginning of a rule execution and at the end of a rule execution. Fresh variables are also introduced, for each machine, to denote resource usage:

- $Desug[[Rule]] = Desug[[(t_i\ r_i^*\ if\ cond_i\ then\ effect_i)]] =$

if/else if $cond_i \wedge mbusy_{fresh} = False$ **then**
$mbusy_{fresh} := True$,
$tcomplete_{fresh} := currentTime + getDuration(t_i)$,
$r_{i_{fresh}} := getResourceConsumption(r_i)$
else if $currentTime = tcomplete_{fresh} \wedge mbusy_{fresh} = True$ **then**
$effect_i$,
$mbusy_{fresh} := False$,
$tcomplete_{fresh}$:= -1,
$r_{i_{fresh}} := 0$
...

Function machines are desugared as macros and sub machines are desugared just like main machines and they are "inlined" inside the rule where they are invoked.

A.3.10 Execution Semantics

The desugaring of the TASM language into the ASM language is an easy way to express the formal semantics of the TASM language. In the ASM world, every main machine represents an "*Agent*", member of the **shared domain** *AGENT*. The TASM language also introduces concurrency semantics that are slightly different than for the ASM language. In the TASM language, time is used to synchronize the order of execution between different agents. It is the *currentTime* dynamic function that keeps all of the agents executing in a synchronized order. The time annotations create a partial order between the moves of agents. The *currentTime* function increases monotonically, at a rate that is congruent with the smallest step of a given main machine. For example, if the shortest duration of a rule is 3 time units, for all agents in *AGENT*, then the *currentTime* function will increment each time by 3 time units; this is denoted by this smallest value *dt*, which corresponds to a **static** function.

The one area that remains to be formally specified is the execution semantics of

resources. For each resource that is defined in the environment, an agent is created that is used to sum up all of the resources used by existing agents. These new agents are used to ensure that resource usage falls within the specified bounds.

Agent $RESOURCE_i$
controlled $last_{fresh}$ initially 0
controlled $totalresource_{i_{fresh}}$ initially 0

if $currentTime = last_{fresh} + dt$ **then**
$totalresource_{i_{fresh}} := sum(r_i)$
else
if $totalresource_{i_{fresh}} > resource_{i_{max}}$ **then**
$RESOURCE\_EXHAUSTED$

The role of the sum macro is to sum up all of the resource annotations from executing agents. The $RESOURCE\_EXHAUSTED$ macro simply halts execution to note that a given resource has been exhausted.

need to be created for each unique object.

6.1.2 Advanced Language Features

Beyond object orientation, other advanced language features such as variable-length arrays and exceptions could also be mapped to ASM. The more implementation language features supported, the more MBSE can exploit those features without fear of loosing automated tool support.

6.1.3 Reverse Engineering

As briefly discussed in chapter 5, reverse engineering an implementation to an ASM is difficult because there are many possible abstractions for an implementation. The simplest model to reverse engineer is one that describes an implementation's functional behavior. However, producing an ASM model that captures every control statement as a guard condition may be too low-level. An extension here could address how a user could efficiently and logically specify a level of abstraction to extract from implementation.

Timing and resource models could also be reverse engineered from implementation. This could be done through leveraging existing dynamic or static analysis engines on the implementation.

6.1.4 Test Case Generation for Timing and Resources

Since implementation code usually does not explicitly express the consumption of resources, this thesis only addresses the translation of functional aspects of the ASM to code. However, it would be meaningful to add the ability to generate code-wrappers for the generated code that would record running time and resource consumption whenever the code is executed. The resulting data recorded by those wrappers could then be imported either manually or automatically back into the TASM tool to verify whether the code executed within the prescribed time bounds.

An example development process that uses this procedure might work as follows: A TASM specification exists with timing annotations on its rules. Prior to using the code generation feature, the test-case generation for timing and resources is enabled. The resulting generated code has been augmented with statements that record the system time into a file before and after the code corresponding to each rule executes. After running the program several times, under whatever environmental variations are appropriate for the system, the resulting file can be imported into the TASM tool. The rules with violated timing constraints are then highlighted.

Appendix B

Electronic Throttle Controller — TASM Model

Authored By: Martin Ouimet, Edited By: David Wang

This appendix provides the listings for the low level TASM model of the Electronic Throttle Controller (ETC) case study, as discussed in Chapter 5. The model expresses the scheduling, tasking, and functional behavior of the ETC. In total, The TASM model contains 5 main machines, 14 function machines, and 20 sub machines.

Table B.1 and B.2 provide a summary of how the listings are organized. For brevity several aspects of the listings have been removed, that do not affect their functionality. Notably:

- TASM models allow multiple *configurations* of a single environment, but require at least one configuration per model. The configuration initializes the main machines and provides alternate initializations of the variables in the environment. No configuration is provided for this model because only one instance of each main machine is used and the default initializations of the environmental variables are used.

- Since TASM is a literate specification language, all constructs of the language (i.e. Environment, Main Machines, Sub Machines, etc...) have a name and description. The descriptions are not listed.

- The Main Machines all have default constructors: constructors that do not alter the initial values of any variables. These constructors are not listed.
- The TASM Specification uses a form of redundancy, where the variables involved in the main, sub, and function machines are listed separate from the rules. These listings are not included.
- The TASM grammar allows the machines to be saved to one file. For clarity, the model is shown, divided by machine. The keyword delimiters required for the saving of these machines into a single text file have been removed.

Name	**Type**	**Purpose**	**Listing**
Types	N/A	List of types	Listing B.1
Resources	N/A	List of resources	Listing B.2
Variables	N/A	List of variables	Listing B.3, B.4
CLOCK	Main	Ticks at 1 ms intervals	Listing B.5
DRIVER	Main	Simulates the behavior of the driver	Listing B.6, B.7
SCHEDULER	Main	Assigns tasks to the processor based on fixed priority and period	Listing B.8
TASKS	Main	Performs the controller functions	Listing B.9
VEHICLE	Main	Simulates the environment	Listing B.10, B.11
Cruise	Function	Determines the cruise control mode	Listing B.12
Cruise_Mode	Function	Sets the cruise mode	Listing B.13
Cruise_Throttle_C	Function	Calculates the cruise mode current	Listing B.14
Driver_Throttle_C	Function	Calculates the human mode current	Listing B.15
Driving_Throttle_C	Function	Calculates the driving mode current	Listing B.16
Fault	Function	Detects if a fault is present	Listing B.17
Limiting_Throttle_C	Function	Calculates the limiting mode current	Listing B.18
Over_Rev	Function	Determines whether the engine revolution is too high	Listing B.19
Over_Rev_Mode	Function	Sets the revolution limiting mode	Listing B.20
Over_Rev_Throttle_C	Function	Calculates the revolution limiting mode current	Listing B.21
Over_Torque	Function	Determines whether the vehicle torque is too high	Listing B.22

Table B.1: List of machines used in the low level ETC model (part 1)

Name	Type	Purpose	Listing
Over_Torque_Mode	Function	Sets the traction limiting mode	Listing B.23
Over_Torque_Throttle_C	Function	Calculates the traction limiting mode current	Listing B.24
finished_to_waiting	Function	Resets completed tasks	Listing B.25
CALCULATE_OUTPUT	Sub	Wrapper machine to calculate the desired current	Listing B.26
DO_SHUTDOWN	Sub	Performs the shut down functions	Listing B.27
DO_STARTUP	Sub	Performs the start up functions	Listing B.28
HANDLE_FAULT	Sub	Performs the fault tolerance functions	Listing B.29
MANAGER_TICK	Sub	Keeps track of manager task period	Listing B.30
MONITOR_HEALTH	Sub	Detects the presence of faults	Listing B.31
MONITOR_TICK	Sub	Keeps track of monitor task period	Listing B.32
SAMPLE_STATE	Sub	Reads the state through sensors for the controller	Listing B.33
SERVO_TICK	Sub	Keeps track of servo task period	Listing B.34
SET_EXECUTING_ TASK	Sub	Assigns execution of a task if the processor is free	Listing B.35
SET_EXECUTION_ PRIORITY	Sub	Decides on the next task to execute based on priority ordering	Listing B.36, B.37
SET_MAJOR_MODE	Sub	Wrapper machine to set the major controller mode	Listing B.38
SET_MAJOR_MODE_WORK	Sub	Sets the controller major mode	Listing B.39
SET_MINOR_MODE	Sub	Wrapper machine to set the minor controller mode	Listing B.40
SET_MINOR_MODE_WORK	Sub	Sets the controller minor mode	Listing B.41
UPDATE_TASK_ STATUSES	Sub	Resets finished tasks to waiting	Listing B.42
WAKE_UP_MANAGER	Sub	Releases manager task on the period	Listing B.43
WAKE_UP_MONITOR	Sub	Releases monitor task on the period	Listing B.44
WAKE_UP_SERVO	Sub	Releases servo task on the period	Listing B.45
WAKE_UP_TASKS	Sub	Wrapper machine to release tasks	Listing B.46

Table B.2: List of machines used in the low level ETC model (part 2)

B.1 Environment

Listing B.1 User-defined types of the model

```
Binary_Mode     := {active, inactive};
Binary_Status   := {on, off};
Health_Status   := {nominal, fault_detected};
Mode            := {off, startup, shutdown, driving, limiting, faulty};
Gear_Status     := {park, drive};
Desired_Current := {none_c, human_c, cruise_c, traction_c, rev_c, min_limiting_c,
                    max_driving_c, fault_c, error_c};
Simulation_Mode := {begin_s, drive_s, random_s, stop_s, done_s};
Task_Status     := {waiting, released, executing, finished};
Scheduler       := {wakeup, update, execute, wait, update_state};
Manager_Step    := {major_mode, minor_mode};
```

Listing B.2 Resources of the model

```
memory := [0, 2048000];     //in bytes
power  := [0, 1000000];     //in milliWatts
```

Listing B.3 Variables of the model (part 1)

```
//internal controller modes
Binary_Mode         rev_limiting_mode   := inactive;
Binary_Mode         traction_mode       := inactive;
Binary_Mode         cruise_mode         := inactive;
Mode                controller_mode     := off;
Control_Mode        control_mode        := sample;
Health_Status       system_health       := nominal;
Boolean             startup_done        := False;
Boolean             shutdown_done       := False;

//powertrain sensors
Integer[0, 120]     vehicle_speed       := 0;       //mph
Integer[0, 8000]    engine_speed        := 0;       //rpm
Integer[0, 250]     vehicle_torque      := 0;       //kPa
Boolean             fault               := False;   //is there a fault?

//driver inputs
Binary_Status       ignition            := off;
Binary_Status       cruise_switch       := off;
Integer[0, 45]      pedal_angle         := 0;       //degrees
Gear_Status         gear                := park;
Binary_Mode         break_pedal         := inactive;
```

Listing B.4 Variables of the model (part 2)

```
//constants
Const Integer       MAX_ENGINE_SPEED    := 6000;    //rpm
Const Integer       MAX_TORQUE          := 110;     //kPA
Const Integer       MIN_CRUISE_SPEED    := 30;      //mph

//controller inputs
Integer[0, 120]     c_vehicle_speed     := 0;
Integer[0, 8000]    c_engine_speed      := 0;       //rpm
Integer[0, 250]     c_vehicle_torque    := 0;       //kPa
Boolean             c_fault             := False;
Binary_Status       c_ignition          := off;
Binary_Status       c_cruise_switch     := off;
Integer[0, 45]      c_pedal_angle       := 0;       //degrees
Gear_Status         c_gear              := park;
Binary_Mode         c_break_pedal       := inactive;

//controller output
Desired_Current     desired_current     := none_c;

//simulation mode
Simulation_Mode     driver_s            := begin_s;
Boolean             vehicle_over_rev_s  := False;
Boolean             vehicle_over_tor_s  := False;

//Task properties
Task_Status         manager_s           := released;
Task_Status         monitor_s           := released;
Task_Status         servo_s             := released;

//Constants
Const Integer       MANAGER_PERIOD      := 10;  //in ms
Const Integer       MONITOR_PERIOD      := 30;  //in ms
Const Integer       SERVO_PERIOD        := 3;   //in ms
Const Integer       MAJOR_CYCLE         := 30;  //in ms

//Scheduler
Scheduler           scheduler_s         := update_state;
Integer             tick                := 0;
Integer             oldtick             := 0;
Integer             managertick         := 0;
Integer             monitortick         := 0;
Integer             servotick           := 0;

//Extra variables for refinement
Manager_Step        manager_s_step      := major_mode;
```

B.2 Main Machines

Listing B.5 CLOCK main machine

```
R1: Tick, no reset
{
  t := 1000;

  if tick != MAJOR_CYCLE then
    tick := tick + 1;
    MANAGER_TICK();
    MONITOR_TICK();
    SERVO_TICK();
}

R2: Tick, with reset
{
  t := 1000;

  if tick = MAJOR_CYCLE then
    tick := 1;
    MANAGER_TICK();
    MONITOR_TICK();
    SERVO_TICK();
}
```

Listing B.6 DRIVER main machine (part 1)

```
R1: Turn on the car
{
  if driver_s = begin_s and controller_mode = off and
     ignition = off then
    ignition := on;
    driver_s := drive_s;
}

R2: Start driving
{
  if driver_s = drive_s and controller_mode = driving and
     vehicle_speed = 0 and gear = park then
    gear          := drive;
    pedal_angle   := 22;
    vehicle_speed := 30;
    driver_s      := random_s;
}

R3: Turn on cruise, slow speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 10;
}

R4: Turn on cruise, normal speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 30;
}

R5: Turn off cruise
{
  if driver_s = random_s and cruise_switch = on then
    cruise_switch := off;
}
```

Listing B.7 DRIVER main machine (part 2)

```
R6: Press break pedal
{
  if driver_s = random_s and break_pedal = inactive then
    break_pedal := active;
}

R7: Depress break pedal
{
  if driver_s = random_s and break_pedal = active then
    break_pedal := inactive;
}

R8: Stop
{
  if driver_s = random_s then
    gear          := park;
    vehicle_speed := 0;
    ignition      := off;
    driver_s      := stop_s;
}

R9: Do nothing
{
  if driver_s = random_s then
    skip;
}

R10: Stopped
{
  if driver_s = stop_s then
    skip;
}

R11: Else
{
  else then
    skip;
}
```

Listing B.8 SCHEDULER main machine

```
R0: Step 0, update state
{
  if scheduler_s = update_state then
    SAMPLE_STATE();
    scheduler_s := update_tasks;
}

R1: Step 1, set status
{
  if scheduler_s = update_tasks then
    UPDATE_TASK_STATUSES();
    scheduler_s := wakeup;
}

R2: Step 2, wake up tasks
{
  if scheduler_s = wakeup then
    WAKE_UP_TASKS();
    scheduler_s := execute;
}

R3: Step 3, set executing
{
  if scheduler_s = execute then
    SET_EXECUTING_TASK();
    scheduler_s := wait;
}

R4: Wait for a tick
{
  t := 1000;

  else then
    scheduler_s := update_state;
}
```

Listing B.9 TASKS main machine

```
R11: Execute manager
{
  t := [0, 3];

  if manager_s = executing and manager_s_step = major_mode then
    SET_MAJOR_MODE();
    manager_s_step := minor_mode;
}

R12: Execute manager
{
  t := [0, 2];

  if manager_s = executing and manager_s_step = minor_mode then
    SET_MINOR_MODE();
    manager_s_step  := major_mode;
    manager_s       := finished;
}

R2: Execute monitor
{
  t := [100, 200];

  if monitor_s = executing then
    MONITOR_HEALTH();
    monitor_s := finished;
}

R3: Execute servo
{
  t := [70, 100];

  if servo_s = executing then
    CALCULATE_OUTPUT();
    servo_s := finished;
}

R4: Else, do nothing, wait for event
{
  t := next;

  else then
    skip;
}
```

Listing B.10 VEHICLE main machine (part 1)

```
R1: Randomly change, do nothing
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    skip;
}

R2: Randomly change RPM
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    engine_speed       := MAX_ENGINE_SPEED + 1;
    vehicle_over_rev_s := True;
}

R3: Randomly change Traction
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    vehicle_torque     := MAX_TORQUE + 1;
    vehicle_over_tor_s := True;
}

R4: Randomly change Both
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    engine_speed       := MAX_ENGINE_SPEED + 1;
    vehicle_over_rev_s := True;
    vehicle_torque     := MAX_TORQUE + 1;
    vehicle_over_tor_s := True;
}

R5: Randomly change RPM, correct
{
  if driver_s = random_s and vehicle_over_rev_s = True and
     vehicle_over_tor_s = False and desired_current = rev_c then
    engine_speed       := MAX_ENGINE_SPEED;
    vehicle_over_rev_s := False;
}
```

Listing B.11 VEHICLE main machine (part 2)

```
R6: Randomly change traction, correct
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = True and desired_current = traction_c then
    vehicle_torque     := MAX_TORQUE;
    vehicle_over_tor_s := False;
}

R7: Randomly change both, correct
{
  if driver_s = random_s and vehicle_over_rev_s = True and
     vehicle_over_tor_s = True and desired_current = min_limiting_c then
    vehicle_torque     := MAX_TORQUE;
    vehicle_over_tor_s := False;
    engine_speed       := MAX_ENGINE_SPEED;
    vehicle_over_rev_s := False;
}

R8: Randomly put in a fault
{
  if driver_s = random_s and fault = False then
    fault := True;
}

R9: Else
{
  else then
    skip;
}
```

B.3 Function Machines

Listing B.12 Cruise function machine

```
R1: Cruise condition
{
  if c_vehicle_speed >= MIN_CRUISE_SPEED
     and c_gear = drive
     and c_break_pedal = inactive
     and c_cruise_switch = on then
    outb := True;
}

R2: Else
{
  else then
    outb := False;
}
```

Listing B.13 Cruise_Mode function machine

```
R1: Cruise Active
{
  if Cruise() then
    out := active;
}

R2: Else
{
    else then
      out := inactive;
}
```

Listing B.14 Cruise_Throttle_C function machine

```
R1: Always
{
  memory := 128;
  power  := 800;

  if True then
    out := cruise_c;
}
```

Listing B.15 Driver_Throttle_C function machine

```
R1: Always
{
  memory := [196, 360];
  power  := [769, 895];

  if True then
    out := human_c;
}
```

Listing B.16 Driving_Throttle_C function machine

```
R1: Cruise enabled, driver input
{
  memory := [324, 826];
  power  := [864, 1695];

  if cruise_mode = active and c_pedal_angle != 0 then
    out := max_driving_c;
}

R2: Cruise enabled, no driver input
{
  if cruise_mode = active and c_pedal_angle = 0 then
    out := Cruise_Throttle_C();
}

R3: Else
{
  else then
   out := Driver_Throttle_C();
}
```

Listing B.17 Fault function machine

```
R1: Main loop
{
  if fault then
    outb := True;
}

R2: Else
{
  else then
    outb := False;
}
```

Listing B.18 Limiting_Throttle_C function machine

```
R1: Both        //if both over rev and over torque are active
{
  memory := 648;
  power  := 1425;

  if rev_limiting_mode = active and traction_mode = active then
    out := min_limiting_c;
}

R2: Over_Rev    //if only over rev is active
{
  if rev_limiting_mode = active and traction_mode = inactive then
    out := Over_Rev_Throttle_C();
}

R3: Over_Torque //if only over torque is active
{
  if rev_limiting_mode = inactive and traction_mode = active then
    out := Over_Torque_Throttle_C();
}

R4: Else        //both are inactive. This should never happen!
{
  else then
    out := error_c;
}
```

Listing B.19 Over_Rev function machine

```
R1: Over Rev Condition
{
  if c_engine_speed > MAX_ENGINE_SPEED then
    outb := True;
}

R2: No Over Rev
{
  else then
    outb := False;
}
```

Listing B.20 Over_Rev_Mode function machine

```
R1: Over Rev Mode
{
  if Over_Rev() then
    out := active;
}

R2: Else {
  else then
    out := inactive;
}
```

Listing B.21 Over_Rev_Throttle_C function machine

```
R1: Always
{
  memory := 256;
  power  := 1200;

  if True then
    out := rev_c;
}
```

Listing B.22 Over_Torque function machine

```
R1: Over Torque Condition
{
  if c_vehicle_torque > MAX_TORQUE then
    outb := True;
}

R2: No Over Torque
{
  else then
    outb := False;
}
```

Listing B.23 Over_Torque_Mode function machine

```
R1: Over Torque Mode
{
  if Over_Torque() then
    out := active;
}

R2: Else
{
  else then
    out := inactive;
}
```

Listing B.24 Over_Torque_Throttle_C function machine

```
R1: Always
{
  memory := 256;
  power  := 1200;

  if True then
    out := traction_c;
}
```

Listing B.25 finished_to_waiting function machine

```
R1:
{
  if in = finished then
    out := waiting;
}

R2:
{
  else then
    out := in;
}
```

B.4 Sub Machines

Listing B.26 CALCULATE_OUTPUT sub machine

```
R1: Driving Mode
{
  if controller_mode = driving then
    desired_current := Driving_Throttle_C();
}

R2: Limiting Mode
{
  if controller_mode = limiting then
    desired_current := Limiting_Throttle_C();
}

R3: Fault Mode
{
  if controller_mode = faulty then
    HANDLE_FAULT();
}

R4: Startup Mode
{
  if controller_mode = startup then
    DO_STARTUP();
}

R5: Shutdown Mode
{
  if controller_mode = shutdown then
    DO_SHUTDOWN();
}

R6: Fault Mode
{
  if controller_mode = off then
    desired_current := none_c;
}
```

Listing B.27 DO_SHUTDOWN sub machine

```
R1: Do shutdown only when vehicle is stationary
{
  memory := 256;
  power  := 900;

  if c_vehicle_speed = 0
     and c_gear = park
     and ignition = off then
    desired_current := none_c;
    shutdown_done   := True;
}

R2: No shutdown
{
  else then
    desired_current := none_c;
    shutdown_done   := False;
}
```

Listing B.28 DO_STARTUP sub machine

```
R1: Do startup only when vehicle is stationary
{
  memory := 128;
  power  := 900;

  if c_vehicle_speed = 0 and c_gear = park and
     c_break_pedal = active and c_cruise_switch = off then
    desired_current := none_c;
    startup_done    := True;
}

R2: No startup
{
  else then
    desired_current := none_c;
    startup_done    := False;
}
```

Listing B.29 HANDLE_FAULT sub machine

```
R1: Handle the fault
{
  memory := 512;
  power  := 895;

  if c_vehicle_speed = 0 and c_gear = park then
    desired_current := none_c;
    controller_mode := shutdown;
    fault           := False;
    c_fault         := False;
}

R2: Else
{
  memory := 512;
  power  := 895;

  else then
    desired_current := fault_c;
}
```

Listing B.30 MANAGER_TICK sub machine

```
R1: tick
{
  if managertick = MANAGER_PERIOD then
    managertick := 1;
}

R2: reset
{
  else then
    managertick := managertick + 1;
}
```

Listing B.31 MONITOR_HEALTH sub machine

```
R1: Find Fault
{
  memory := [512, 1024];
  power  := [1530, 1624];

  if Fault() then
    system_health := fault_detected;
}

R2: Else do nothing
{
  memory := [512, 1024];
  power  := [1530, 1624];

  else then
    system_health := nominal;
}
```

Listing B.32 MONITOR_TICK sub machine

```
R1: tick
{
  if monitortick = MONITOR_PERIOD then
    monitortick := 1;
}

R2: reset
{
  else then
    monitortick := monitortick + 1;
}
```

Listing B.33 SAMPLE_STATE sub machine

```
R1: Cache the state
{
  if True then
    c_vehicle_speed     := vehicle_speed;
    c_engine_speed      := engine_speed;
    c_vehicle_torque    := vehicle_torque;
    c_ignition          := ignition;
    c_cruise_switch     := cruise_switch;
    c_pedal_angle       := pedal_angle;
    c_gear              := gear;
    c_break_pedal       := break_pedal;
}
```

Listing B.34 SERVO_TICK sub machine

```
R1: tick
{
  if servotick = SERVO_PERIOD then
    servotick := 1;
}

R2: reset
{
  else then
    servotick := servotick + 1;
}
```

Listing B.35 SET_EXECUTING_TASK sub machine

```
R1: Someone is still executing, do nothing
{
  if manager_s = executing or monitor_s = executing or
     servo_s = executing then
    skip;
}

R2: Processor is free, assign a task
{
  else then
    SET_EXECUTION_PRIORITY();
}
```

Listing B.36 SET_EXECUTION_PRIORITY sub machine (part 1)

```
R1: All released
{
  if manager_s = released and monitor_s = released and
     servo_s = released then
    manager_s := executing;
}

R2: manager, monitor released
{
  if manager_s = released and monitor_s = released and
     servo_s != released then
    manager_s := executing;
}

R3: manager, servo released
{
  if manager_s = released and servo_s = released and
     monitor_s != released then
    manager_s := executing;
}

R4: monitor, servo released
{
  if monitor_s = released and servo_s = released and
     manager_s != released then
    monitor_s := executing;
}
```

Listing B.37 SET_EXECUTION_PRIORITY sub machine (part 2)

```
R5: only manager
{
  if manager_s = released and monitor_s != released and
     servo_s != released then
    manager_s := executing;
}

R6: only monitor
{
  if monitor_s = released and manager_s != released and
     servo_s != released then
    monitor_s := executing;
}

R7: only servo
{
  if servo_s = released and manager_s != released and
     monitor_s != released then
    servo_s := executing;
}

R8: no one released
{
  if manager_s != released and monitor_s != released and
     servo_s != released then
    skip;
}
```

Listing B.38 SET_MAJOR_MODE sub machine

```
R1: No fault
{
  if system_health = nominal then
    SET_MAJOR_MODE_WORK();
}

R2: Else there are faults
{
  if system_health = fault_detected and controller_mode != shutdown then
    controller_mode = faulty;
}

R3: Else
{
  else then
    skip;
}
```

Listing B.39 SET_MAJOR_MODE_WORK sub machine

```
R1: Off -> Startup
{
  if controller_mode = off and ignition = on then
    controller_mode := startup;
}

R2: Startup -> Driving
{
  if controller_mode = startup and startup_done = True then
    controller_mode := driving;
}

R3: Driving -> Limiting
{
  if controller_mode = driving and (Over_Rev() or Over_Torque()) and
     ignition = on then
    controller_mode := limiting;
}

R4: Limiting -> Driving
{
  if controller_mode = limiting and not (Over_Rev() or Over_Torque()) and
     ignition = on then
    controller_mode := driving;
}

R5: Driving, Limiting, Faulty -> Shutdown
{
  if (controller_mode = limiting or controller_mode = driving or
      controller_mode = faulty) and ignition = off then
    controller_mode := shutdown;
}

R6: Shutdown -> Off
{
  if controller_mode = shutdown and shutdown_done = True then
    controller_mode := off;
}

R7: Any other case, do nothing
{
  else then
    skip;
}
```

Listing B.40 SET_MINOR_MODE sub machine

```
R1: No fault
{
  if system_health = nominal then
    SET_MINOR_MODE_WORK();
}

R2: Else
{
  else then
    skip;
}
```

Listing B.41 SET_MINOR_MODE_WORK sub machine

```
R1: Else
{
  if True then
    cruise_mode             := Cruise_Mode();
    rev_limiting_mode       := Over_Rev_Mode();
    traction_mode           := Over_Torque_Mode();
}
```

Listing B.42 UPDATE_TASK_STATUSES sub machine

```
R1: We are at a tick
{
  if tick != oldtick then
    manager_s := finished_to_waiting(manager_s);
    monitor_s := finished_to_waiting(monitor_s);
    servo_s   := finished_to_waiting(servo_s);
}

R2: Not at a tick
{
  else then
    skip;
}
```

Listing B.43 WAKE_UP_MANAGER sub machine

```
R1: wakeup
{
  if manager_s = waiting and managertick = MANAGER_PERIOD then
    manager_s   := released;
}

R2: otherwise
{
  else then
    skip;
}
```

Listing B.44 WAKE_UP_MONITOR sub machine

```
R1: wakeup
{
  if monitor_s = waiting and monitortick = MONITOR_PERIOD then
    monitor_s   := released;
}

R2: otherwise
{
  else then
    skip;
}
```

Listing B.45 WAKE_UP_SERVO sub machine

```
R1: wakeup
{
  if servo_s = waiting and servotick = SERVO_PERIOD then
    servo_s   := released;
}

R2: otherwise
{
  else then
    skip;
}
```

Listing B.46 WAKE_UP_TASKS sub machine

```
R1: wakeup
{
  if oldtick != tick then
    WAKE_UP_MANAGER();
    WAKE_UP_MONITOR();
    WAKE_UP_SERVO();
    oldtick := tick;
}

R2: Else
{
  else then
    skip;
}
```

Appendix C

Electronic Throttle Controller — Java Model

This appendix includes the listing of the Java code generated from the low level ETC TASM model listed in Appendix A.

Name	Page
Environment.java	141
ConfigurationSimple.java	144
Clock.java	145
Driver.java	147
Scheduler.java	149
Tasks.java	153
Vehicle.java	160

Table C.1: List of Java files generated from the low level ETC model.

Environment.java

```
package ETC_Low_Level;

public class Environment {

    public enum Binary_Mode {active, inactive};
    public enum Binary_Status {on, off};
    public enum Health_Status {nominal, fault_detected};
    public enum Mode {off, startup, shutdown, driving, limiting, faulty};
    public enum Gear_Status {park, drive};
    public enum Control_Mode {sample, mode_set_major, mode_set_minor, output, health};
    public enum Desired_Current {none_c, human_c, cruise_c, traction_c, rev_c, min_limiting_c, max_driving_c,
    public enum Simulation_Mode {begin_s, drive_s, random_s, stop_s};
    public enum Task_Status {waiting, released, executing, finished};
    public enum Scheduler {wakeup, update_tasks, execute, wait, update_state};
    public enum Manager_Step {major_mode, minor_mode};

    //internal controller modes
    public static Binary_Mode rev_limiting_mode = Binary_Mode.inactive;
```

```
public static Binary_Mode traction_mode = Binary_Mode.inactive;
public static Binary_Mode cruise_mode = Binary_Mode.inactive;
public static Mode controller_mode = Mode.off;
public static Control_Mode control_mode = Control_Mode.sample;
public static Health_Status system_health = Health_Status.nominal;
public static boolean startup_done = false;
public static boolean shutdown_done = false;

//powertrain sensors
public static int vehicle_speed = 0; //mph
public static int engine_speed = 0; //rpm
public static int vehicle_torque = 0; //kPa
public static boolean fault = false; //is there a fault?

//driver inputs
public static Binary_Status ignition = Binary_Status.off;
public static Binary_Status cruise_switch = Binary_Status.off;
public static int pedal_angle = 0; //degrees
public static Gear_Status gear = Gear_Status.park;
public static Binary_Mode break_pedal = Binary_Mode.inactive;

//constants
public static int MAX_ENGINE_SPEED = 6000; //rpm
public static int MAX_TORQUE = 110; //kPA
public static int MIN_CRUISE_SPEED = 30; //mph

//controller inputs
public static int c_vehicle_speed = 0;
public static int c_engine_speed = 0; //rpm
public static int c_vehicle_torque = 0; //kPa
public static boolean c_fault = false;
public static Binary_Status c_ignition = Binary_Status.off;
public static Binary_Status c_cruise_switch = Binary_Status.off;
public static int c_pedal_angle = 0; //degrees
public static Gear_Status c_gear = Gear_Status.park;
public static Binary_Mode c_break_pedal = Binary_Mode.inactive;

//controller output
public static Desired_Current desired_current = Desired_Current.none_c;

//simulation mode
public static Simulation_Mode driver_s = Simulation_Mode.begin_s;
public static boolean vehicle_over_rev_s = false;
public static boolean vehicle_over_tor_s = false;

//Task properties
public static Task_Status manager_s = Task_Status.released;
public static Task_Status monitor_s = Task_Status.released;
public static Task_Status servo_s = Task_Status.released;

//Constants
public static final int MANAGER_PERIOD = 10; //in ms
public static final int MONITOR_PERIOD = 30; //in ms
public static final int SERVO_PERIOD = 3; //in ms
```

```
    public static final int MAJOR_CYCLE = 30; //in ms

    //Scheduler
    public static Scheduler scheduler_s = Scheduler.wakeup;
    public static int tick = 0;
    public static int oldtick = 0;
    public static int managertick = 0;
    public static int monitortick = 0;                                    80
    public static int servotick = 0;

    //Extra variables for refinement
    public static Manager_Step manager_s_step = Manager_Step.major_mode;

}
```

ConfigurationSimple.java

```
package ETC_Low_Level;

public class ConfigurationSimple {

    public static void main(String[] args){
        // VARIABLE INITIALIZATIONS

        // MACHINE INITIALIZATIONS
        Clock clock = new Clock();
        Driver driver = new Driver();                                    10
        Scheduler scheduler = new Scheduler();
        Tasks tasks = new Tasks();
        Vehicle vehicle= new Vehicle();

        clock.start();
        driver.start();
        scheduler.start();
        tasks.start();
        vehicle.start();
    }                                                                    20

}
```

Clock.java

```
package ETC_Low_Level;

public class Clock extends Thread {

    public Clock() {
    }

    public void run() {
        if (Environment.tick != Environment.MAJOR_CYCLE) {
            // R1: Tick, no reset                                    10
            Environment.tick = Environment.tick + 1;
            MANAGER_TICK();
            MONITOR_TICK();
            SERVO_TICK();
        } else if (Environment.tick == Environment.MAJOR_CYCLE) {
            // R2: Tick, with reset
            Environment.tick = 1;
            MANAGER_TICK();
            MONITOR_TICK();
            SERVO_TICK();                                          20
        }
    }

    public void MANAGER_TICK() {
        if (Environment.managertick == Environment.MANAGER_PERIOD) {
            // R1: tick
            Environment.managertick = 1;
        } else {
            // R2: reset
            Environment.managertick = Environment.managertick + 1; 30
        }
    }

    public void MONITOR_TICK() {
        if (Environment.monitortick == Environment.MONITOR_PERIOD) {
            // R1: tick
            Environment.monitortick = 1;
        } else {
            // R2: reset
            Environment.monitortick = Environment.monitortick + 1; 40
        }
    }

    public void SERVO_TICK() {
        if (Environment.servotick == Environment.SERVO_PERIOD) {
            // R1: tick
            Environment.servotick = 1;
        } else {
            // R2: reset
            Environment.servotick = Environment.servotick + 1;     50
        }
    }
```

```
}
```

Driver.java

```
package ETC_Low_Level;

import ETC_Low_Level.Environment.Binary_Mode;
import ETC_Low_Level.Environment.Binary_Status;
import ETC_Low_Level.Environment.Gear_Status;
import ETC_Low_Level.Environment.Mode;
import ETC_Low_Level.Environment.Simulation_Mode;

public class Driver extends Thread {

    public Driver() {
    }

    public void run() {
        while (true) {
            if (Environment.driver_s == Simulation_Mode.begin_s
                    && Environment.controller_mode == Mode.off
                    && Environment.ignition == Binary_Status.off) {
                // R1: Turn on the car
                Environment.ignition = Binary_Status.on;
                Environment.driver_s = Simulation_Mode.drive_s;
            } else if (Environment.driver_s == Simulation_Mode.drive_s
                    && Environment.controller_mode == Mode.driving
                    && Environment.vehicle_speed == 0
                    && Environment.gear == Gear_Status.park) {
                // R2: Start driving
                Environment.gear = Gear_Status.drive;
                Environment.pedal_angle = 22;
                Environment.vehicle_speed = 30;
                Environment.driver_s = Simulation_Mode.random_s;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.cruise_switch == Binary_Status.off) {
                // R3: Turn on cruise, slow speed
                Environment.cruise_switch = Binary_Status.on;
                Environment.vehicle_speed = 10;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.cruise_switch == Binary_Status.off) {
                // R4: Turn on cruise, normal speed
                Environment.cruise_switch = Binary_Status.on;
                Environment.vehicle_speed = 30;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.cruise_switch == Binary_Status.on) {
                // R5: Turn off cruise
                Environment.cruise_switch = Binary_Status.off;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.break_pedal == Binary_Mode.inactive) {
                // R6: Press break pedal
                Environment.break_pedal = Binary_Mode.active;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.break_pedal == Binary_Mode.active) {
                // R7: Depress break pedal
                Environment.break_pedal = Binary_Mode.inactive;
```

```
            } else if (Environment.driver_s == Simulation_Mode.random_s) {
                // R8: Stop
                Environment.gear = Gear_Status.park;
                Environment.vehicle_speed = 0;
                Environment.ignition = Binary_Status.off;
                Environment.driver_s = Simulation_Mode.stop_s;
            } else if (Environment.driver_s == Simulation_Mode.random_s) {
                // R9: Do nothing                                          60
                continue; // skip;
            } else if (Environment.driver_s == Simulation_Mode.stop_s) {
                // R10: Stopped
                continue; // skip;
            } else {
                continue; // skip;
            }
        }
    }
                                                                          70
}
```

Scheduler.java

```
package ETC_Low_Level;

import ETC_Low_Level.Environment.Task_Status;

public class Scheduler extends Thread {

    public Scheduler() {
    }

    public void run() {                                                    10
        while (true) {
            if (Environment.scheduler_s == Environment.Scheduler.update_state) {
                // R0: Step 0, update state
                SAMPLE_STATE();
                Environment.scheduler_s = Environment.Scheduler.update_tasks;
            } else if (Environment.scheduler_s == Environment.Scheduler.update_tasks) {
                // R1: Step 1, set status
                UPDATE_TASK_STATUSES();
                Environment.scheduler_s = Environment.Scheduler.wakeup;
            } else if (Environment.scheduler_s == Environment.Scheduler.wakeup) {   20
                // R2: Step 2, wake up tasks
                WAKE_UP_TASKS();
                Environment.scheduler_s = Environment.Scheduler.execute;
            } else if (Environment.scheduler_s == Environment.Scheduler.execute) {
                // R3: Step 3, set executing
                SET_EXECUTING_TASK();
                Environment.scheduler_s = Environment.Scheduler.wait;
            } else if (Environment.scheduler_s == Environment.Scheduler.wait) {
                // R4: Wait for a tick
                Environment.scheduler_s = Environment.Scheduler.update_state;   30
            }
        }
    }

    public void SAMPLE_STATE() {
        if (true) {
            // R1: Cache the state
            Environment.c_vehicle_speed = Environment.vehicle_speed;
            Environment.c_engine_speed = Environment.engine_speed;
            Environment.c_vehicle_torque = Environment.vehicle_torque;          40
            Environment.c_ignition = Environment.ignition;
            Environment.c_cruise_switch = Environment.cruise_switch;
            Environment.c_pedal_angle = Environment.pedal_angle;
            Environment.c_gear = Environment.gear;
            Environment.c_break_pedal = Environment.break_pedal;
            Environment.c_fault = Environment.fault;
        }
    }

    public void SET_EXECUTING_TASK() {                                     50
        if (Environment.manager_s == Task_Status.executing
                || Environment.monitor_s == Task_Status.executing
```

```
            || Environment.servo_s == Task_Status.executing) {
        // R1: Someone is still executing, do nothing
    } else {
        // R2: Processor is free, assign a task
        SET_EXECUTION_PRIORITY();
    }
}

public void SET_EXECUTION_PRIORITY() {
    if (Environment.manager_s == Task_Status.released
            && Environment.monitor_s == Task_Status.released
            && Environment.servo_s == Task_Status.released) {
        // R1: All Task_Status.released
        Environment.manager_s = Task_Status.executing;
    } else if (Environment.manager_s == Task_Status.released
            && Environment.monitor_s == Task_Status.released
            && Environment.servo_s != Task_Status.released) {
        // R2: manager, monitor Task_Status.released
        Environment.manager_s = Task_Status.executing;
    } else if (Environment.manager_s == Task_Status.released
            && Environment.servo_s == Task_Status.released
            && Environment.monitor_s != Task_Status.released) {
        // R3: manager, servo Task_Status.released
        Environment.manager_s = Task_Status.executing;
    } else if (Environment.monitor_s == Task_Status.released
            && Environment.servo_s == Task_Status.released
            && Environment.manager_s != Task_Status.released) {
        // R4: monitor, servo Task_Status.released
        Environment.monitor_s = Task_Status.executing;
    } else if (Environment.manager_s == Task_Status.released
            && Environment.monitor_s != Task_Status.released
            && Environment.servo_s != Task_Status.released) {
        // R5: only manager
        Environment.manager_s = Task_Status.executing;
    } else if (Environment.monitor_s == Task_Status.released
            && Environment.manager_s != Task_Status.released
            && Environment.servo_s != Task_Status.released) {
        // R6: only monitor
        Environment.monitor_s = Task_Status.executing;
    } else if (Environment.servo_s == Task_Status.released
            && Environment.manager_s != Task_Status.released
            && Environment.monitor_s != Task_Status.released) {
        // R7: only servo
        Environment.servo_s = Task_Status.executing;
    } else if (Environment.manager_s != Task_Status.released
            && Environment.monitor_s != Task_Status.released
            && Environment.servo_s != Task_Status.released) {
        // R8: no one Task_Status.released
    }
}

public void UPDATE_TASK_STATUSES() {
    if (Environment.tick != Environment.oldtick) {
        // R1: We are at a tick
```

```
        Environment.manager_s = finished_to_waiting(Environment.manager_s);
        Environment.monitor_s = finished_to_waiting(Environment.monitor_s);
        Environment.servo_s = finished_to_waiting(Environment.servo_s);
    } else {                                                                    110
        // R2: Not at a tick
    }
}

public void WAKE_UP_MANAGER() {
    if (Environment.manager_s == Task_Status.waiting
            && Environment.managertick == Environment.MANAGER_PERIOD) {
        // R1: wakeup
        Environment.manager_s = Task_Status.released;
    } else {                                                                    120
        // R2: otherwise
    }
}

public void WAKE_UP_MONITOR() {
    if (Environment.monitor_s == Task_Status.waiting
            && Environment.monitortick == Environment.MONITOR_PERIOD) {
        // R1: wakeup
        Environment.monitor_s = Task_Status.released;
    } else {                                                                    130
        // R2: otherwise
    }
}

public void WAKE_UP_SERVO() {
    if (Environment.servo_s == Task_Status.waiting
            && Environment.servotick == Environment.SERVO_PERIOD) {
        // R1: wakeup
        Environment.servo_s = Task_Status.released;
    } else {                                                                    140
        // R2: otherwise
    }
}

public void WAKE_UP_TASKS() {
    if (Environment.oldtick != Environment.tick) {
        // R1: wakeup
        WAKE_UP_MANAGER();
        WAKE_UP_MONITOR();
        WAKE_UP_SERVO();                                                        150
        Environment.oldtick = Environment.tick;
    } else {
        // R2: Else
    }
}

public Task_Status finished_to_waiting(Task_Status in) {
    Task_Status out;
    if (in == Task_Status.finished) {
        // R1:                                                                  160
```

```
            out = Task_Status.waiting;
        } else {
            // R2:
            out = in;
        }
        return out;
    }

}
```

Tasks.java

```
package ETC_Low_Level;

import ETC_Low_Level.Environment.Binary_Mode;
import ETC_Low_Level.Environment.Binary_Status;
import ETC_Low_Level.Environment.Desired_Current;
import ETC_Low_Level.Environment.Gear_Status;
import ETC_Low_Level.Environment.Health_Status;
import ETC_Low_Level.Environment.Manager_Step;
import ETC_Low_Level.Environment.Mode;
import ETC_Low_Level.Environment.Task_Status;

public class Tasks extends Thread {

    public Tasks() {
    }

    public void run() {
        while (true) {
            if (Environment.manager_s == Task_Status.executing
                    && Environment.manager_s_step == Manager_Step.major_mode) {
                // R11: Execute manager
                SET_MAJOR_MODE();
                Environment.manager_s_step = Manager_Step.minor_mode;
            } else if (Environment.manager_s == Task_Status.executing
                    && Environment.manager_s_step == Manager_Step.minor_mode) {
                // R12: Execute manager
                SET_MINOR_MODE();
                Environment.manager_s_step = Manager_Step.major_mode;
                Environment.manager_s = Task_Status.finished;
            } else if (Environment.monitor_s == Task_Status.executing) {
                // R2: Execute monitor
                MONITOR_HEALTH();
                Environment.monitor_s = Task_Status.finished;
            } else if (Environment.servo_s == Task_Status.executing) {
                // R3: Execute servo
                CALCULATE_OUTPUT();
                Environment.servo_s = Task_Status.finished;
            }
        }
    }

    public void CALCULATE_OUTPUT() {
        if (Environment.controller_mode == Mode.driving) {
            // R1: Driving Mode
            Environment.desired_current = Driving_Throttle_C();
            // from
            // throttle_v !
        } else if (Environment.controller_mode == Mode.limiting) {
            // R2: Limiting Mode
            Environment.desired_current = Limiting_Throttle_C();
        } else if (Environment.controller_mode == Mode.faulty) {
            // R3: Fault Mode
```

```
        HANDLE_FAULT();
    } else if (Environment.controller_mode == Mode.startup) {
        // R4: Startup Mode
        DO_STARTUP();
    } else if (Environment.controller_mode == Mode.shutdown) {
        // R5: Shutdown Mode
        DO_SHUTDOWN();
    } else if (Environment.controller_mode == Mode.off) {                    60
        // R6: Fault Mode
        Environment.desired_current = Desired_Current.none_c;
    }
}

public void DO_SHUTDOWN() {
    if (Environment.c_vehicle_speed == 0
            && Environment.c_gear == Gear_Status.park
            && Environment.ignition == Binary_Status.off) {
        // R1: Do shutdown only when vehicle is stationary                   70
        Environment.desired_current = Desired_Current.none_c;
        Environment.shutdown_done = true;
    } else {
        // R2: No shutdown
        Environment.desired_current = Desired_Current.none_c;
        Environment.shutdown_done = false;
    }
}

public void DO_STARTUP() {                                                   80
    if (Environment.c_vehicle_speed == 0
            && Environment.c_gear == Gear_Status.park
            && Environment.c_break_pedal == Binary_Mode.active
            && Environment.c_cruise_switch == Binary_Status.off) {
        // R1: Do startup only when vehicle is stationary
        Environment.desired_current = Desired_Current.none_c;
        Environment.startup_done = true;
    } else {
        // R2: No startup
        Environment.desired_current = Desired_Current.none_c;                90
        Environment.startup_done = false;
    }
}

public void HANDLE_FAULT() {
    if (Environment.c_vehicle_speed == 0
            && Environment.c_gear == Gear_Status.park) {
        // R1: Handle the fault
        Environment.desired_current = Desired_Current.none_c;
        Environment.controller_mode = Mode.shutdown;                         100
        Environment.fault = false;
        Environment.c_fault = false;
    } else {
        // R2: Else
        Environment.desired_current = Desired_Current.fault_c;
    }
```

```
}

public void MONITOR_HEALTH() {
    if (Fault()) {
        // R1: Find Fault
        Environment.system_health = Health_Status.fault_detected;
    } else {
        // R2: Else do nothing
        Environment.system_health = Health_Status.nominal;
    }
}

public void SET_MAJOR_MODE() {
    if (Environment.system_health == Health_Status.nominal) {
        // R1: No fault
        SET_MAJOR_MODE_WORK();
    } else if (Environment.system_health == Health_Status.fault_detected
            && Environment.controller_mode != Mode.shutdown) {
        // R2: Else there are faults
        Environment.controller_mode = Mode.faulty;
    } else {
        // R3: Else
    }
}

public void SET_MAJOR_MODE_WORK() {
    if (Environment.controller_mode == Mode.off
            && Environment.ignition == Binary_Status.on) {
        // R1: Off -> Startup
        Environment.controller_mode = Mode.startup;
    } else if (Environment.controller_mode == Mode.startup
            && Environment.startup_done == true) {
        // R2: Startup -> Driving
        Environment.controller_mode = Mode.driving;
    } else if (Environment.controller_mode == Mode.driving
            && (Over_Rev() || Over_Torque())
            && Environment.ignition == Binary_Status.on) {
        // R3: Driving -> Limiting
        Environment.controller_mode = Mode.limiting;
    } else if (Environment.controller_mode == Mode.limiting
            && !(Over_Rev() || Over_Torque())
            && Environment.ignition == Binary_Status.on) {
        // R4: Limiting -> Driving
        Environment.controller_mode = Mode.driving;
    } else if ((Environment.controller_mode == Mode.limiting
            || Environment.controller_mode == Mode.driving
            || Environment.controller_mode == Mode.faulty)
            && Environment.ignition == Binary_Status.off) {
        // R5: Driving, Limiting, Faulty -> Shutdown
        Environment.controller_mode = Mode.shutdown;
    } else if (Environment.controller_mode == Mode.shutdown
            && Environment.shutdown_done == true) {
        Environment.controller_mode = Mode.off;
        // R6: Shutdown -> Off
```

```
    } else {
        // R7: Any other case, do nothing
    }
}

public void SET_MINOR_MODE() {
    if (Environment.system_health == Health_Status.nominal) {
        // R1: No fault
        SET_MINOR_MODE_WORK();
    } else {                                                          170
        // R2: Else
    }
}

public void SET_MINOR_MODE_WORK() {
    if (true) {
        // R1: Else
        Environment.cruise_mode = Cruise_Mode();
        Environment.rev_limiting_mode = Over_Rev_Mode();
        Environment.traction_mode = Over_Torque_Mode();               180
    }
}

public boolean Cruise() {
    boolean outb;
    if (Environment.c_vehicle_speed >= Environment.MIN_CRUISE_SPEED
            && Environment.c_gear == Gear_Status.drive
            && Environment.c_break_pedal == Binary_Mode.inactive
            && Environment.c_cruise_switch == Binary_Status.on) {
        // R1: Cruise condition                                       190
        outb = true;
    } else {
        // R2: Else
        outb = false;
    }
    return outb;
}

public Binary_Mode Cruise_Mode() {
    Binary_Mode out;                                                  200
    if (Cruise()) {
        // R1: Cruise Active
        out = Binary_Mode.active;
    } else {
        // R2: Else
        out = Binary_Mode.inactive;
    }
    return out;
}
                                                                      210
public Desired_Current Driver_Throttle_C() {
    Desired_Current out;
    if (true) {
        // R1: Always
```

```
        out = Desired_Current.human_c;
    }
    return out;
}

public Desired_Current Driving_Throttle_C() {
    Desired_Current out;
    if (Environment.cruise_mode == Binary_Mode.active
            && Environment.c_pedal_angle != 0) {
        // R1: Cruise enabled, driver input
        out = Desired_Current.max_driving_c;
    } else if (Environment.cruise_mode == Binary_Mode.active
            && Environment.c_pedal_angle == 0) {
        // R2: Cruise enabled, no driver input
        out = Desired_Current.cruise_c;
    } else {
        // R3: Else
        out = Driver_Throttle_C();
    }
    return out;
}

public boolean Fault() {
    Boolean outb;
    if (Environment.c_fault) {
        // R1: Main loop
        outb = true;
    } else {
        // R2: Else
        outb = false;
    }
    return outb;
}

public Desired_Current Limiting_Throttle_C() {
    Desired_Current out;
    if (Environment.rev_limiting_mode == Binary_Mode.active
            && Environment.traction_mode == Binary_Mode.active) {
        // R1: Both //if both over rev and over torque are active
        out = Desired_Current.min_limiting_c;
    } else if (Environment.rev_limiting_mode == Binary_Mode.active
            && Environment.traction_mode == Binary_Mode.inactive) {
        // R2: Over_Rev //if only over rev is active
        out = Over_Rev_Throttle_C();
    } else if (Environment.rev_limiting_mode == Binary_Mode.inactive
            && Environment.traction_mode == Binary_Mode.active) {
        // R3: Over_Torque //if only over torque is active
        out = Over_Torque_Throttle_C();
    } else {
        // R4: Else //both are inactive. This should never happen!
        out = Desired_Current.error_c;
    }
    return out;
}
```

```
public boolean Over_Rev() {
    Boolean outb;
    if (Environment.c_engine_speed > Environment.MAX_ENGINE_SPEED) {
        // R1: Over Rev Condition
        outb = true;
    } else {
        // R2: No Over Rev
        outb = false;
    }
    return outb;
}

public Binary_Mode Over_Rev_Mode() {
    Binary_Mode out;
    if (Over_Rev()) {
        // R1: Over Rev Mode
        out = Binary_Mode.active;
    } else {
        // R2: Else
        out = Binary_Mode.inactive;
    }
    return out;
}

public Desired_Current Over_Rev_Throttle_C() {
    Desired_Current out;
    if (true) {
        // R1: Always
        out = Desired_Current.rev_c;
    }
    return out;
}

public boolean Over_Torque() {
    Boolean outb;
    if (Environment.c_vehicle_torque > Environment.MAX_TORQUE) {
        // R1: Over Torque Condition
        outb = true;
    } else {
        // R2: No Over Torque
        outb = false;
    }
    return outb;
}

public Binary_Mode Over_Torque_Mode() {
    Binary_Mode out;
    if (Over_Torque()) {
        // R1: Over Torque Mode
        out = Binary_Mode.active;
    } else {
        // R2: Else
        out = Binary_Mode.inactive;
```

```
        }
        return out;
    }

    public Desired_Current Over_Torque_Throttle_C() {
        Desired_Current out;
        if (true) {
            // R1: Always                                                    330
            out = Desired_Current.traction_c;
        }
        return out;
    }

}
```

Vehicle.java

```
package ETC_Low_Level;

import ETC_Low_Level.Environment.Desired_Current;
import ETC_Low_Level.Environment.Simulation_Mode;

public class Vehicle extends Thread {

    public Vehicle() {
    }

    public void run() {
        while (true) {
            if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.vehicle_over_rev_s == false
                    && Environment.vehicle_over_tor_s == false) {
                // R1: Randomly change, do nothing
                continue;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.vehicle_over_rev_s == false
                    && Environment.vehicle_over_tor_s == false) {
                // R2: Randomly change RPM
                Environment.engine_speed = Environment.MAX_ENGINE_SPEED + 1;
                Environment.vehicle_over_rev_s = true;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.vehicle_over_rev_s == false
                    && Environment.vehicle_over_tor_s == false) {
                // R3: Randomly change Traction
                Environment.vehicle_torque = Environment.MAX_TORQUE + 1;
                Environment.vehicle_over_tor_s = true;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.vehicle_over_rev_s == false
                    && Environment.vehicle_over_tor_s == false) {
                // R4: Randomly change Both
                Environment.engine_speed = Environment.MAX_ENGINE_SPEED + 1;
                Environment.vehicle_over_rev_s = true;
                Environment.vehicle_torque = Environment.MAX_TORQUE + 1;
                Environment.vehicle_over_tor_s = true;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.vehicle_over_rev_s == true
                    && Environment.vehicle_over_tor_s == false
                    && Environment.desired_current == Desired_Current.rev_c) {
                // R5: Randomly change RPM, correct
                Environment.engine_speed = Environment.MAX_ENGINE_SPEED;
                Environment.vehicle_over_rev_s = false;
            } else if (Environment.driver_s == Simulation_Mode.random_s
                    && Environment.vehicle_over_rev_s == false
                    && Environment.vehicle_over_tor_s == true
                    && Environment.desired_current == Desired_Current.traction_c) {
                // R6: Randomly change traction, correct
                Environment.vehicle_torque = Environment.MAX_TORQUE;
                Environment.vehicle_over_tor_s = false;
            } else if (Environment.driver_s == Simulation_Mode.random_s
```

```
                && Environment.vehicle_over_rev_s == true
                && Environment.vehicle_over_tor_s == true
                && Environment.desired_current == Desired_Current.min_limiting_c) {
            // R7: Randomly change traction, correct
            Environment.vehicle_torque = Environment.MAX_TORQUE;
            Environment.vehicle_over_tor_s = false;
            Environment.engine_speed = Environment.MAX_ENGINE_SPEED;
            Environment.vehicle_over_rev_s = false;                              60
        } else if (Environment.driver_s == Simulation_Mode.random_s
                && Environment.fault == false) {
            // R8: Randomly put in a fault
            Environment.fault = true;
        } else {
            // R9: Else
            continue;
        }
    }
}                                                                                70

}
```

Bibliography

[1] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza. Information retrieval models for recovering traceability links between code and documentation. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 40, Washington, DC, USA, 2000. IEEE Computer Society.

[2] Giuliano Antoniol. *Recovery of traceability links in software artifacts and systems.* PhD thesis, Montreal, P.Q., Canada, Canada, 2004. Adviser-Merlo, Ettore M.

[3] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[4] John W. Backus and Peter Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1), 1963.

[5] Utpal Banerjee. *Loop Parallelization.* Loop Transformations for Restructuring Compilers. Kluwer Academic Publishers, Norwell, Massachusetts, 1994.

[6] Utpal Banerjee. *Dependence Analysis.* Loop Transformations for Restructuring Compilers. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[7] Mike Barnett, Egon Borger, Yuri Gurevich, Wolfram Shculte, and Margus Veanes. Using Abstract State Machines at Microsoft: A Case Study. Available from http://research.microsoft.com, 1998.

[8] Egon Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM '95)*, volume 1012 of *LNCS*. Springer-Verlag, 1995.

[9] Egon Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Computer Science*, 8(5), 2001.

[10] Egon Börger and Robert Stärk. *Abstract State Machines.* Springer-Verlag, 2003.

[11] Yves Boussemart, Sébastien Gorelov, Martin Ouimet, and Kristina Lundqvist. Non-Intrusive System-Level Fault Tolerance for an Electronic Throttle Controller. In *Proceedings of the Fifth International Conference on Networking (ICN '06) and International Conference on Systems (ICONS '06) and International*

Conference on Mobile Communications and Learning. IEEE Computer Society Press, April 23–29 2006.

[12] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Traceability recovery using numerical analysis. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 195–204, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Jiun-Liang Chenapos;, Feng jian Wang, and Yung lin Chen. Slicing object-oriented programs. In *Proceedings of the APSECapos;97*, pages 395–404, 1997.

[14] Edmund M. Clarke and Jeannette Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(3), 1996.

[15] Joëlle Cohen and Anatol Slissenko. On Verification of Refinements of Asynchronous Timed Distributed Algorithms. In *International Workshop on Abstract State Machines (ASM '00)*, pages 100–114. Springer-Verlag, 2000.

[16] Hartmut Ehrig, Claudia Ermel, and Frank Hermann. On the relationship of model transformations based on triple and plain graph grammars. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*, pages 9–16, New York, NY, USA, 2008. ACM.

[17] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of Design Languages: A Compilation Approach using Abstract State Machines. In *Proceedings of the International Workshop on Abstract State Machines - ASM 2000*, volume 1912 of *LNCS*. Springer-Verlag, 2000.

[18] Jeanne Ferrante and Karl J. Ottenstein. A program form based on data dependency in predicate regions. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 217–236, New York, NY, USA, 1983. ACM.

[19] Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS*, pages 543–557. Springer Verlag, 2006.

[20] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating Finite State Machines From Abstract State Machines. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software Testing and Analysis*, pages 112–122, 2002.

[21] Paul G. Griffiths. Embedded Software Control Design for an Electronic Throttle Body. Master's thesis, University of California, Berkeley, 2002.

[22] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, pages 9–36, 1995.

[23] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[24] Mohammad R. Haghighat. *Symbolic Analysis for Parallelizing Compilers.* Kluwer Academic Publishers, Norwell, Massachusetts, 1995.

[25] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–157, New York, NY, USA, 1988. ACM.

[26] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs, 1990.

[27] James K. Huggins and Charles Wallace. An abstract state machine primer. Computer Science Technical Report CS-TR-02-04, Michigan Technological University, December 2004.

[28] ALT Software Inc. Do-178b and do-178b a certification and safety critical testing - alt software. http://www.altsoftware.com/embedded-products/supplements/do-178b-safety-critical.html, May 2009.

[29] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Department of Computer Science, University of Paderborn, 2007.

[30] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graph, 1998.

[31] Mathworks. Simulink and Stateflow Product Web Page. http://www.mathworks.com/products/simulink.

[32] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.

[33] Martin Ouimet. *A Formal Framework for Specification-Based Embedded Real-Time System Engineering.* PhD dissertation, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, 2008.

[34] Juergen Rilling, Philippe Charland, and René Witte. Traceability in software engineering – past, present and future. Technical Report TR-74-211, IBM Technical Report, CASCON 2007 Workshop, October 25 2007.

[35] John E. Savage. *The Complexity of Computing.* Krieger Publishing Co., Inc., Melbourne, FL, USA, 1987.

[36] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, London, UK, 1995. Springer-Verlag.

[37] Wuwei Shen, Kevin Compton, and James Huggins. A uml validation toolset based on abstract state machines. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 315, Washington, DC, USA, 2001. IEEE Computer Society.

[38] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 432–441, New York, NY, USA, 1999. ACM.

[39] United States. *RTCA, Inc., Document RTCA/DO-178B [electronic resource].* U.S. Dept. of Transportation, Federal Aviation Administration, [Washington, D.C.] :, 1993.

[40] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[41] D.Y. Tamanaha, W.C. Wenjen, and B.K. Patel. The application of case in large aerospace projects. pages 18 pp.–, Feb 1989.

[42] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 389–402, New York, NY, USA, 2010. ACM.

[43] Hans Van Vliet. *Software Engineering: Principles and Practice.* John Wiley and Sons, 2001.

[44] Neil Walkinshaw, Marc Roper, Murray Wood, and Neil Walkinshaw Marc Roper. The java system dependence graph. In *In Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 5–5, 2003.

[45] Kirsten Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[46] Marvin V. Zelkowitz. Perspectives in Software Engineering. *ACM Computing Surveys*, 10(2):197–216, 1978.

[47] Jianjun Zhao. Multithreaded dependence graphs for concurrent java programs. In *In Proceedings of 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 13–23. IEEE Computer Society, 1999.