Florida International University
FIU Digital Commons

FIU Electronic Theses and Dissertations | University Graduate School

6-20-2014

A Middleware to Support Services Delivery in a Domain-Specific Virtual Machine

Karl A. Morris
Florida International University, karl.morris@fiu.edu

DOI: 10.25148/etd.FI14071101
Follow this and additional works at: https://digitalcommons.fiu.edu/etd

Recommended Citation
Morris, Karl A., "A Middleware to Support Services Delivery in a Domain-Specific Virtual Machine" (2014). *FIU Electronic Theses and Dissertations*. 1437.
https://digitalcommons.fiu.edu/etd/1437

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A MIDDLEWARE TO SUPPORT SERVICES DELIVERY IN A DOMAIN-SPECIFIC VIRTUAL MACHINE

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Karl A. Morris

2014

To: Dean Amir Mirmiran, Ph.D.
College of Engineering and Computing

This dissertation, written by Karl A. Morris, and entitled A Middleware to Support Services Delivery in a Domain-Specific Virtual Machine, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Masoud Milani

Xudong He

Jinpeng Wei

Debra VanderMeer

Peter J. Clarke, Major Professor

Date of Defense: June 20, 2014

The dissertation of Karl A. Morris is approved.

Dean Amir Mirmiran, Ph.D.
College of Engineering and Computing

Dean Lakshmi Reddi, Ph.D.
University Graduate School

Florida International University, 2014

© Copyright 2014 by Karl Morris

All rights reserved.

DEDICATION

To my family and friends who have, and continue to believe in me.

ACKNOWLEDGMENTS

I would like to thank my colleagues, collaborators, teachers, and mentors for their help and motivation throughout my journey. I would like to give a special thanks to my committee members, Drs. Milani, He, Wei, and VanderMeer, as well as my dissertation advisor, Dr. Peter J. Clarke.

ABSTRACT OF THE DISSERTATION

A MIDDLEWARE TO SUPPORT SERVICES DELIVERY IN A DOMAIN-SPECIFIC VIRTUAL MACHINE

by

Karl A. Morris

Florida International University, 2014

Miami, Florida

Professor Peter J. Clarke, Major Professor

The increasing use of model-driven software development has renewed emphasis on using domain-specific models during application development. More specifically, there has been emphasis on using domain-specific modeling languages (DSMLs) to capture user-specified requirements when creating applications. The current approach to realizing these applications is to translate DSML models into source code using several model-to-model and model-to-code transformations. This approach is still dependent on the underlying source code representation and only raises the level of abstraction during development. Experience has shown that developers will many times be required to manually modify the generated source code, which can be error-prone and time consuming.

An alternative to the aforementioned approach involves using an interpreted domain-specific modeling language (i-DSML) whose models can be directly executed using a Domain Specific Virtual Machine (DSVM). Direct execution of i-DSML models require a semantically rich platform that reduces the gap between the application models and the underlying services required to realize the application. One layer in this platform is the domain-specific middleware that is responsible for the management and delivery of services in the specific domain.

In this dissertation, we investigated the problem of designing the domain-specific middleware of the DSVM to facilitate the bifurcation of the semantics of the do-

main and the model of execution (MoE) while supporting runtime adaptation and validation. We approached our investigation by seeking solutions to the following sub-problems: (1) How can the domain-specific knowledge (DSK) semantics be separated from the MoE for a given domain? (2) How do we define a generic model of execution (GMoE) of the middleware so that it is adaptable and realizes DSK operations to support delivery of services? (3) How do we validate the realization of DSK operations at runtime?

Our research into the domain-specific middleware was done using an i-DSML for the user-centric communication domain, Communication Modeling Language (CML), and for microgrid energy management domain, Microgrid Modeling Language (MGridML). We have successfully developed a methodology to separate the DSK and GMoE of the middleware of a DSVM that supports specialization for a given domain, and is able to perform adaptation and validation at runtime.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1

2 LITERATURE REVIEW 4
2.1 Background 4
2.1.1 Middleware 4
2.1.2 Models at Runtime 5
2.1.3 Model Checking 6
2.1.4 Execution of Domain-Specific Models 6
2.1.5 CML and CVM 7
2.1.6 Interpreted Domain-Specific Modeling Languages 9
2.1.7 Domain-Specific Virtual Machines 10
2.2 Related Work 14
2.2.1 Adaptive Middleware 14
2.2.2 Middleware Execution Models 16
2.2.3 Model Validation 17
2.2.4 User-centric Communication Middleware 19
2.2.5 Summary of Middleware Comparison 21

3 PROBLEM DEFINITION AND METHODOLOGY 24
3.1 Motivation 24
3.2 Problem Statement 27
3.3 Categorize and Encapsulate Domain-Specific Knowledge (DSK) 28
3.3.1 Encapsulation of Domain-Specific Knowledge 28
3.4 Develop a Generic Model of Execution (GMoE) 30
3.4.1 Generic Model of Execution for a DSVM Middleware 30
3.5 Define a Mechanism for Efficient Functionality Validation 31
3.5.1 Validation of Intent Models 32

4 DOMAIN KNOWLEDGE ENCAPSULATION 34
4.1 Overview 34
4.2 Domain Knowledge Encapsulation 36
4.2.1 Domain Specific Classifiers 36
4.2.2 Structural Metamodel 38
4.3 Chapter Summary 41

5 GENERIC MODEL OF EXECUTION FOR A DSVM MIDDLEWARE . . . 43
5.1 Model Generation 43
5.2 Model of Execution 49
5.2.1 Middleware Execution Model 49
5.3 DSVM Middleware Design 54
5.3.1 Structural Design 55

5.3.2 Behavioral Design 58
5.4 Chapter Summary 63

6 VALIDATION OF INTENT MODELS 65
6.1 Model Validation 65
6.2 Chapter Summary 69

7 EXPERIMENTATION 70
7.1 Generic Model of Execution 70
7.2 Model Validation 78
7.2.1 Middleware Artifacts 78
7.2.2 Translation 80
7.2.3 Validation 80
7.2.4 Comparative Analysis 82
7.3 MicrogridVM 83

8 CONCLUSION 88
8.1 Summary of Research 88
8.2 Future Work 89
8.2.1 Improved Efficiency and Robustness 90
8.2.2 Behavioral Adaptation 90
8.2.3 Cross platform applicability 91

BIBLIOGRAPHY 92

APPENDICES 98

A CVM Control Scripts 98

B MGRID Control Scripts 99

C Network Communication Broker API 100

D Alloy Specifications 101

LIST OF FIGURES

FIGURE PAGE

2.1 Layered architecture of the Communication Virtual Machine 7

2.2 CML Models for the scenario. (a) Control schema. (b) Data schema-1. (c) Data Schema-2 11

2.3 Layered architecture of the Communication Virtual Machine 12

2.4 Control scripts generated during the synthesis of the CML instnaces in Figure 2.2. CI_n represents the nth control instance being synthesized. Similarly for the data instances. 14

2.5 UCM Architecture 20

3.1 DSVM specialization during instantiation. 26

4.1 Model Generation and Selection Process 34

4.2 Composition of an Intent Model 38

4.3 Class Diagram for an Intent Model. 41

5.1 Model Generator Class Diagram 43

5.2 Package diagram showing the main components in the DSVM middleware. 55

5.3 Class diagram for the `m_interpreter` package in the DSVM middleware. 56

5.4 Class diagram for `im_generator` package. 57

5.5 Command Received Statechart 59

5.6 Event Received Statechart 59

5.7 Model Executor Class Diagram 61

5.8 Model Execution Statechart Diagram 62

7.1 Minimum, maximum and average model generation times 74

7.2 Variable File Transfer Operation 76

7.3 Simulated Transfer of 10 files 78
7.4 Scenario Intent Models: Send command 79
7.5 Middleware Polices using DSCs 80
7.6 Alloy Counterexample 82
7.7 Inconsistent Model Validation 84
7.8 MGrid Policy 85
7.9 Scenario Intent Models: Send command 86
7.10 Alloy Counterexample 87
A.1 CVM Control Scripts 98
B.1 MGridVM Control Scripts 99

LIST OF ACRONYMS

CML	Communication Modeling Language
CVM	Communication Virtual Machine
DSC	Domain Specific Classifier
DSK	Domain Specific Knowledge
DSML	Domain Specific Modeling Language
DSVM	Domain Specific Virtual Machine
ECA	Event Condition Action
FoL	First Order Logic
GMoE	Generic Model of Execution
i-DSML	Interpreted Domain Specific Modeling Language
MDSD	Model Driven Software Development
MGridML	Microgrid Modeling Language
MGridVM	Microgrid Virtual Machine
MoE	Model of Execution
NCB	Network Communication Broker
SE	Synthesis Engine
SPL	Software Product Line
UCI	User Communication Interface
UCM	User-Centric Communication Middleware

CHAPTER 1

INTRODUCTION

Model Driven Software Development (MDSD) has become a very widely used paradigm in the area of Software Engineering with its growth increasing tremendously in recent years [23]. Conventional approaches to MDSD focus on model transformation where models in one language are translated into another language prior to execution, e.g., models created in UML are translated into Java [41, 45]. A developing trend in this area is to remove the steps involved in conventional model translation, and to instead execute the models directly. This requires a semantically rich environment that is able to interpret models of sufficient abstraction.

One such environment that supports model execution is the execution engine for *Interpreted Domain-Specific Modeling Language* (i-DSML) models. An i-DSML execution engine facilitates the direct execution of models through a 4-layer architecture, where each layer receives and performs operations on an increasingly granular view of the model, before passing the transformed version of the model to the next layer in the stack. We will refer to the i-DSML execution engine as a *Domain-Specific Virtual Machine* (DSVM).

The architecture of a DSVM has applications in numerous domains, and therefore their efficient instantiation is a desired property. In order to achieve this goal, we must define methods to separate the domain-specific knowledge (DSK) of a DSVM from its model of execution (MoE). Having done so, we are then able to instantiate a DSVM by combining the generic model of execution (GMoE) with the necessary DSK at the various layers of the DSVM. The methods defined for the instantiation of a DSVM will vary by layer, as each layer's view of the model is less-abstract and its operations more platform specific than the layer in the stack that precedes it. Additionally, domains may require an instantiation of the DSVM to

operate in a stand-alone or a distributed manner and therefore the defined architectures must support these modes of operation.

We agree with the findings of Schantz et al. [51] that today's middleware must move beyond simple connectivity to, but provide support for effective distributed systems. This includes end-to-end QoS, an open, extensible system, and the ability to provide sustained, and correctly functioning operations in diverse environments. As Schantz describes, a distributed middleware must provide a programming model that allows clients to program distributed and stand-alone applications in the same manner.

The research outlined in this proposal will investigate the following question. *How to design a DSVM middleware layer so that the DSK can be separated from the GMoE, it supports adaptability at runtime and effectively provides the delivery of services in the domain.* The solution to the research question will be achieved by solving the following sub-problems. (1) How can the DSK semantics be separated from the MoE for a given domain? (2) How do we define a generic model of execution (GMoE) for the DSVM middleware so that it is adaptable and realizes DSK operations to support delivery of services? (3) How do we validate the realization of DSK operations at runtime?

This proposal incorporates previously published work by Morris et al. [43] and will build on it in the following areas: (1) defining a full execution model for the DSVM middleware; (2) providing refinements to the described artifacts; (3) a prototype implementation of the DSVM middleware; (4) the demonstration of the architecture's ability to varying domains through the encapsulation of DSK in both the user-centric communication and microgrid energy management domains; and (5) the development of an efficient and extensible method of validating the DSK operations based on system policies.

The major contributions are as follows:

1. A mechanism and necessary artifacts for the proper separation of DSK from the GMoE in a DSVM middleware.

2. A method to dynamically realize operational semantics in the execution of control scripts through the context-aware generation of *intent models*.

3. A method for the selection of policy-complaint intent models for structural adaptation using model checking techniques.

In the next chapter we provide background information on i-DSML and DSVMs as well present some related work in the areas of middleware architectures and middleware execution platforms. Chapter 3 identifies the research problem, provides motivation, and outlines the goals of our proposed solution and a methodology to achieve them. Chapters 4, 5, and 6 detail our major contributions in achieving our stated goals. Chapter 7 provides experimentation results based on the evaluation methods described in Chapter 3. Finally, Chapter 8 summarizes the contributions of this dissertation and outlines future work towards furthering our research.

CHAPTER 2

LITERATURE REVIEW

In this chapter we provide necessary background information and related work deemed relevant to solving our research problem. In Section 2.1, we detail the concepts and existing approaches germane to our research. In Section 2.2, we present related work in the field and discuss the various shortcomings with respect to our research problem that we address in our approach.

2.1 Background

In this section we provide background on the areas that we focus on in our research including adaptive middleware, models at runtime, model checking, and an i-DSML and DSVM for user-centric communication.

2.1.1 Middleware

There are many definitions and types of middleware described in the literature [7, 61], we use a generic definition, which states that middleware is any software that allows other software to interact [11]. Adaptive middleware, as with adaptive systems in general, allows for the monitoring and reconfiguring of its structure and/or behavior at runtime [21]. This is achieved through various introspection mechanisms such as reflection. Adaptability is a desired property in many systems and one that is inherent to DSVMs. Adaptable middleware is a cornerstone of many of today's complex systems that require interoperability and context awareness.

Sadjadi [50] present a taxonomy of various adaptable middleware approaches, and provides a detailed comparison of each. Examples of such domains and systems include multimedia [29, 62], communication [58, 47, 40] and generally systems that incorporate distributed architectures or have deadlines and require end-to-end

quality of service [27, 56]. The ability to adapt a system based on system context gives it the ability to change based on the availability of new information and new resources, a feature that is mandatory for systems to provide guarantees on their operation. One obvious concern of adaptable systems is the general overhead intrinsic to the monitoring and adaptation process when compared to non-adaptable systems [13]. This concern must continuously be addressed and new research in adaptable systems must balance the granularity of adaptation with the required responsiveness of the system.

2.1.2 Models at Runtime

Models at runtime provide us with a mechanism to leverage models and the abstraction they provide during the real time operation of a system [6]. Similar to models used in the software development process, a runtime model allows us to reason about a system's environment and its behavior by presenting us with relevant information, while abstracting away superfluous information [35, 53]. This process opens the door to semantic based reasoning on, and modification of, a system's architecture and operations.

We are able to simulate actions to be performed on a systems using models to allow us to measure their results and potential side effects prior to effecting those actions onto the real system. Models that maintain a causal link with its system and environment can greatly increase the efficiency of analysis and change of that system [42]; the trade-off being that the mechanism to maintain that causal link can add a resource overhead that potentially mitigates the performance boost achieved by analyzing the model.

2.1.3 Model Checking

Model Checking allows us to algorithmically check whether a particular model satisfies some predefined specification [16]. Generally, we express models as a digraph with a set of nodes and edges, with each node representing a particular state of the program being modeled and possessing a set of atomic propositions, and each edge representing a state transition within the system. The model is checked for satisfiability using a specification language such a temporal logic [17]. Model checking has been used to perform reachability tests [9], to determine the time delays of systems [4], and to perform verification of system properties in various domains [28, 44]. We will be dynamically generating models in our solution and therefore expect that the concepts of model checking will be used to validate these models.

2.1.4 Execution of Domain-Specific Models

DSMLs allow end-users to easily generate solutions for problems in their respective domains since the solutions are created using abstractions closer to the problem space [22, 34, 36]. Conventional approaches to realize DSML models usually requires these models to be converted into source code in a traditional high-level language (HLL) using a series of model-to-model and model-to-text transformations. This source code must then be compiled and executed. An alternative approach is to execute these domain-specific models directly using an execution engine. We refer to the languages used to create these models as *Interpreted Domain-Specific Modeling Languages* (i-DSMLs) [18] and the execution engine as a *Domain-Specific Virtual Machine* (DSVM).

In our opinion, the execution engines used to interpret i-DSML models can be considered as a virtual machine based on the taxonomy of virtual machines presented by Smith et al. [54]. DSVMs can be classified as dynamic translators i.e., HLL VMs. Note however, we are moving to a higher level of abstraction, from

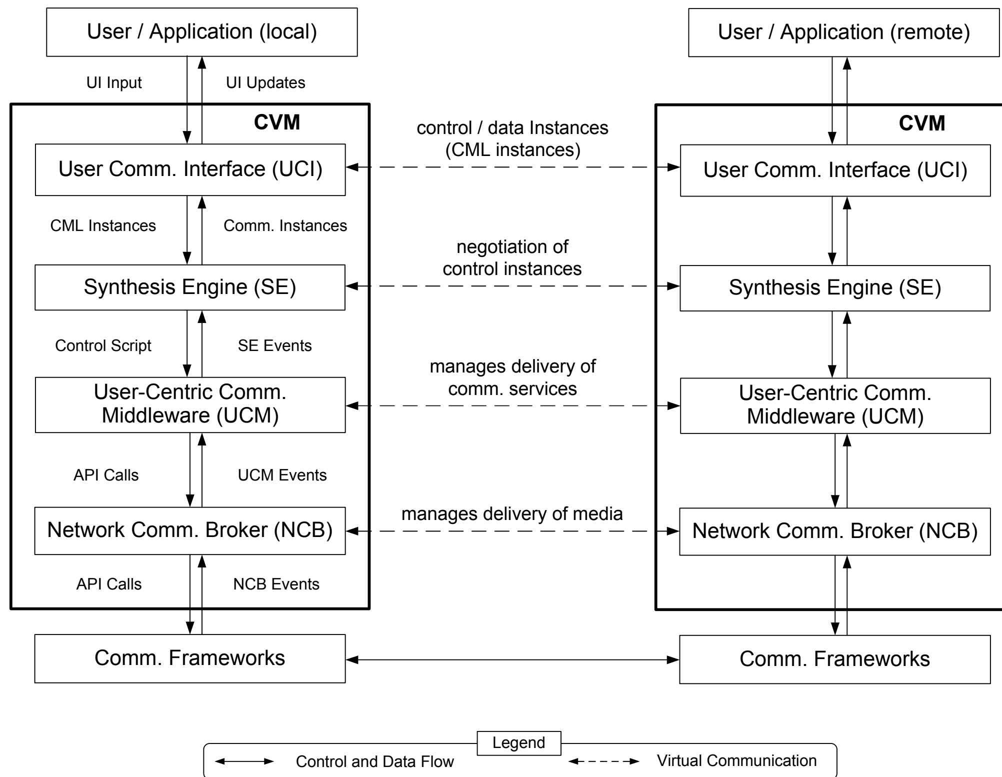

Figure 2.1: Layered architecture of the Communication Virtual Machine

HLLs to domain-specific models. In the subsequent subsections we introduce i-DSMLs and DSVMs. As previously stated our research team has worked on two DSVMs, the *Communication Virtual Machine* (CVM) in the user-centric communication domain [19, 64], and *Microgrid Virtual Machine* (MGridVM) in the energy management for smart microgrid domain [2, 3]. In this dissertation we will focus on the CVM, which is a distributed system unlike the MGridVM, which is currently a stand alone system.

2.1.5 CML and CVM

The *Communication Virtual Machine* (CVM) [19] is the DSVM for the *Communication Modeling Language* (CML), an i-DMSL for user-centric communication. CVM provides a runtime environment for the modeling and realization of user-centric

communication services (see Figure 2.3). Models are specified using CML, which contain representations of concepts relevant in the user-centric communication domain [65].

Two categories of communication models can be described using CML, communication schemas and communication instances, similar to the relationship between use cases and scenarios during requirements analysis. CML allows a domain expert to declaratively specify the requirements of a communication instance. It has no mechanism to dictate how a particular instance is realized. CML creates high level models that are interpreted by CVM.

The CVM platform is divided into four major levels of abstraction with each layer playing a role in the realization of communication services. The layers are:

1. *User Communication Interface* (UCI) - provides an environment for users to specify their communication requirements using CML

2. *Synthesis Engine* (SE) - synthesizes and negotiates CML models with other participants in a communication and generates control scripts (see Appendix Figure A.1)

3. *User-centric Communication Middleware* (UCM) - executes commands found in a communication control script to manage and coordinate the delivery of communication services to users

4. *Network Communication Broker* (NCB) - provides a network-independent API (see Appendix Table C.1) to UCM and manages underlying frameworks to deliver communication services.

The CVM is an inherently distributed system where communication is established between two or more parties. Our work focuses on the functions of the User-centric Communication Middleware, which has responsibility for achieving service delivery.

CML models provide us a mechanism for stating intent. It contains the necessary artifacts to stipulate the requirements of a communication instance. However it lacks the facility to state how a particular communication instance should be realized, and what, if any, constraints must be adhered to. We therefore utilize policies to express these non-functional requirements. Within the middleware, we utilize the Event Condition Action (ECA) policy model [49], where events are control script commands being realized or events from the underlying layer being responded to, conditions are state information retrievable directly from the middleware or through an external monitor, and actions are accompanying actions that must be performed when realizing the initially requested command or the action association with the event being responded to.

2.1.6 Interpreted Domain-Specific Modeling Languages

An i-DSML can be described as a five-tuple, similar to a DSML [12], consisting of a concrete syntax, e.g., graphical models; abstract syntax that defines the language syntax and integrity constraints; semantic domain, containing the domain-specific knowledge; a mapping that assigns syntactic constructs to elements in the abstract syntax; and a semantic mapping that relates abstract syntactic concepts to the semantic domain. The main difference between the traditional DSMLs and i-DSMLs is that the semantics of traditional DSMLs describe how to transform models into source code for a given HLL. The semantics for i-DSMLs define how the application captured by the model is executed to realize the intent of the requirements without first transforming the mode into an HLL.

An i-DSML model may either be a *control schema* or a *data schema*, which is based on the concepts of the program and data, respectively, as described by Plotkin [46]. The control schema specifies the logical configuration of some set of requirements (functional and non-functional) for an application in the domain. The data schema

contains instances of domain types and user-defined types as specified in the control schema. We use the terms *control instance* and *data instance* to refer to fully instantiated schemas, similar to objects in the OO paradigm. There are three concrete syntax notations used to represent i-DSML schemas, these include a graphical representation, a user-friendly interface representation, and an XML-based representation.

The following scenario from the user-centric communication domain will be used throughout this article to illustrate various aspects related to i-DSMLs and DSVMs, specifically related to the middleware.

Scenario: Following Dr. Burke's surgery on Baby Jane, he returns to his office and contacts Dr. Monteiro, the attending physician, to let him know the results of the surgery. During the conversation Dr. Burke shares several aspects of the patient's medical records with him, including the post-surgery echocardiogram, images of the patient's heart captured during the surgery, and the vital signs.

Figure 2.2 shows the control schema and two data schemas for the medical scenario presented above. The control schema shown in part (a) represents the configuration for the communication and the media types used across the connection. The data schema shown in part (b) initiates the audio video connection, the data schema in part (c) initiates the patient record to be sent a a form. For more details on the metamodel (abstract syntax and static semantics) for CML, see Wu et al. [64].

2.1.7 Domain-Specific Virtual Machines

The DSVM design is based on the architecture used first in the CVM and then in the MGridVM. The DSVM uses a four-layered architecture described as follows:

- *User Interface* (UI) - provides the user with an environment to specify their domain requirements using either a graphical model or a user friendly interface.

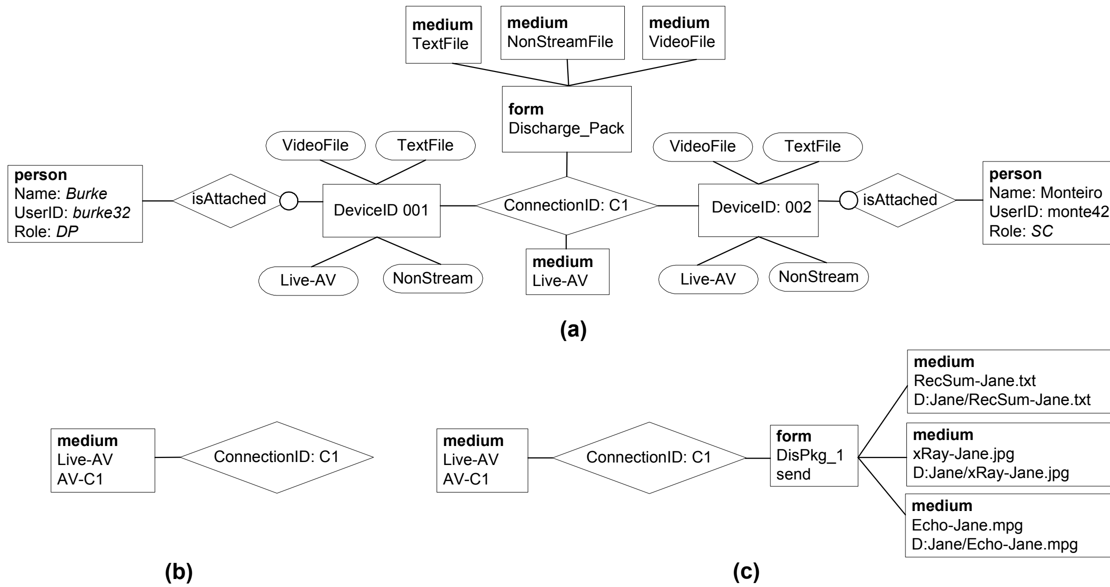

Figure 2.2: CML Models for the scenario. (a) Control schema. (b) Data schema-1. (c) Data Schema-2

The UI then transform the i-DSML modle into an XML-based representation for processing.

- *Synthesis Engine* (SE) - synthesizes models by comparing the current runtime model with a new user-defined model. Based on the changes between the models and the current state of the SE control scripts are generated to be executed by the middleware.

- *Middleware* (M) - executed the commands in the control scripts in order to manage and coordinate the delivery of domain services. During the execution of these scripts user-defined polices may be applied during the delivery of services.

- *Broker* (B) - provide an independent API to the middleware that hides the heterogeneity of the underlying services provided by frameworks and con-

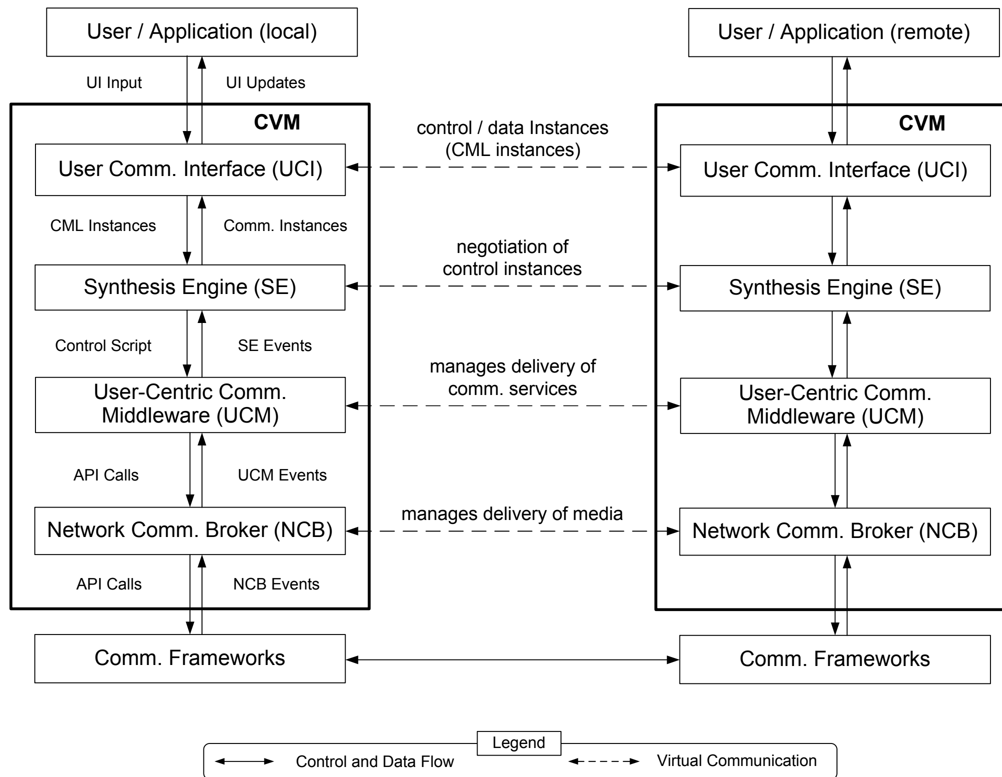

Figure 2.3: Layered architecture of the Communication Virtual Machine

trollers. The broker also interprets underlying events from the frameworks and controllers and generates events to be handled by the upper layers in the DSVM.

Based on the domain, the layers of the DSVM are specialized to service that specific domain. Figure 2.3 shows the specialized layers of the CVM. The CVM middleware (*User-Centric Communication Middleware* (UCM)) is responsible for interpreting controls scripts for the user-centric communication domain resulting in the delivery and management of communication services. These services may include sending files to all parties in a connection, encrypting/decryption files, starting/stopping audio-video streams and applying communication polices. The UCM realizes the delivery of services by making calls to the Network Communi-

cation Broker API. A list of the methods available in the Network Communication Broker API is shown in Table C.1.

The MGridVM middleware (*Microgrid Control Middleware* (MCM)) interprets control scripts in the energy management domain resulting in the delivery and management of energy management services for microgrids. These services may include mapping groups of physical devices to logical controllers, executing energy management algorithms, and applying policies to various device configurations [18]. Note that we described the functionality of the MCM to show the applicability of the DSVM middleware in another domain other than user-centric communication. However, our focus in this paper will be on the middleware in the CVM, the user-centric communication middleware (UCM).

Figure A.1 (Appendix)shows the structure of the control scripts generated during model synthesis using EBNF-like notation. The rules of the control scripts are shown using an attributed grammar, where the keywords are shown in bold and the attributes are denoted by a subscript. For example, Rule 1 states that a control script may contain one or more script commands and Rule 2 shows the various commands. Rules 5 states that the *addParticipantCmd* consists of the keyword `addParticipant` and takes two parameters a connection id and a list of one or more participant ids.

The table in Figure 2.4 shows an example of the the control scripts generated during the synthesis of the CML instances show in Figure 2.2. CI_n and DI_n represent the nth instance of the model used during synthesis. The initial instance with subscript *0* is referred to as the null model and is the initial runtime model. For each pair of models, the one with the higher subscript is the new user model received by the synthesis engine to be processed.

The control script generated in the first row second column of the table represents the scripts to establish a connection with the other participant in the commu-

Models Synthesized	Control Scripts Generated
CI_0 - Null Control Instance CI_1 - Control Instance in Figure 1(a)	createConnection("C1") sendSchema("C1", "burke32", "monte42", "CI_1, DI_0") sendSchema("C1", "burke32", "monte42", "CI_1, DI_0")
DI_0 - Null Data Instance DI_1 - Data Instance in Figure 1(b)	enableInitiator("C1", "Live-AV") sendSchema("C1", "burke32", "monte42", "CI_1, DI_1")
DI_1 - Data Instance in Figure 1(b) DI_2 - Data Instance in Figure 1(c)	sendForm("C1", "DisPkg_1", "D:Jane/RecSum-Jane.txt") sendForm("C1", "DisPkg_1", "D:Jane/xRay-Jane.jpg") sendForm("C1", "DisPkg_1", "D:Jane/Echo-Jane.mpg") sendSchema("C1", "burke32", "monte42, "CI_1, DI_2")

Figure 2.4: Control scripts generated during the synthesis of the CML instnaces in Figure 2.2. CI_n represents the nth control instance being synthesized. Similarly for the data instances.

nication. That is, a request is made to create a connection, then send the control schema to the other participant to be negotiated on. Note that there are two script commands that are identical, this is due to the three phrase protocol used during negotiation between the participants in the communication. In Figure B.1, we see the set of control scripts for the Microgrid domain. For more details on the synthesis process we refer the reader to the work by Wu et al. [64].

2.2 Related Work

Here we present related work in the areas of Adaptive Middleware Systems, Middleware Execution Models and previous work on the User-Centric Communication Middleware.

2.2.1 Adaptive Middleware

In the work of Kramer et al.[37], the authors present an approach to developing self-managing systems that utilize self-configuring components. Their work focuses

at the architectural level, where components are automatically aligned in order to achieve system goals. This approach does not present a mechanism to select among various components that achieve the same end, but instead rely on the presence of a specific component that undertakes predefined tasks. This is an expected limitation in the authors' work as there is no notion of an operation taxonomy as is present in our approach. As our work focuses on the service delivery layer of our DSVM, we are divorced from the physical mechanisms that effect actions within our environment. Instead, we focus on determining the best way to realize intent (or goals) by employing potentially numerous procedures that perform the needed function. Additionally, Kramer at al. state that planning and verification are performed offline, which they state is sufficient if all possible system states can be addressed. Our architecture must be able to perform all planning operations at runtime as (1) the combination of relevant state information is determined by policies in place at the time of command execution, and (2) the set of available procedures can be altered at runtime.

Zachariadis et al. [67] demonstrate an adaptable mobile middleware that can augment its functionality based on functional component availability. While interesting, this work does not address having variability in components that meet the needs of the system, nor does it purport to be agile in its execution as components are monolithic in their composition and operation. Additionally, this approach provides no consideration of policies nor does it provide a single dialect with which to analyze component capabilities against system policies. Our architecture facilitates adaptation with a minimal resource footprint due to the use of execution units. Additionally, we provide a mechanism to analyze models when more than one are able to execute an operation. This can have a direct impact on the middleware's operational speed.

2.2.2 Middleware Execution Models

Bellur et al. [5] present an approach to dynamically bind middleware components for execution based on user intent and context. Their work is similar to our own in that it treats the platform as a base for execution in a programmatic way. Our work differs in the granularity of operations as their components are analogous to our procedures. As procedures are further broken down into executable units, we are able to achieve finer grain execution and adaptation. Additionally, in the authors' work the variation points at which dependencies are defined do not specify any direct cost analysis mechanism for deciding on most efficient option for selection. Our approach is also inherently distributed, as the platform's event registration service allows procedures to be distributed across multiple remote instances of the middleware.

Madl et al. [39] describe a method to perform verification of distributed real-time properties via model checking. This, in principle, is similar to our proposed approach of utilizing model checking techniques to validate a proposed system transformation via the execution of an intent model. Both approaches apply formal methods to ensure the middleware behaves as expected. Their results show that model checking techniques can effectively verify event-driven behavior of a component based middleware. Our approach differs fundamentally in that our validation process takes place at runtime upon receipt of an event, instead of at design time as in the case of the cited work. Additionally, our validation is performed against Event Condition Action-type (ECA) policies, which requires the existence of a common communication dialect, as well as a mechanism for transforming these policies into logical formulas to check for satisfiability.

Veríssimo et al. [60] describe a middleware, CORTEX, that presents a programming model and architecture for creating ubiquitous, autonomous applications that address concerns such as safety, responsiveness, and mobility. The architec-

ture's enumerated list of first-class concerns facilitates easy development of applications that possess these characteristics. While this programming model is desirable in many instances, it may limit the applicability of the architecture in some domains, or increase the complexity of development as the model may not be apropos for the task at hand. As a result of the domain independence of our approach, our architecture does not enumerate any potential concerns and therefore provide fewer high level abstractions that would aid in addressing them. This helps to limit the presence of superfluous functionality of our middleware once it has been specialized for a specific domain as all first-class operations relate solely to the middleware's internal operations and are not directly accessible via the programming model.

2.2.3 Model Validation

While the application of first-order logic in the validation of DSVM middleware intent model functionality is novel, a large body of research exists that deals with the use of formal methods to validate system operations. This includes a strong focus on validating variability in systems, which is where much of our research lies.

Jiang et al. [32] describe a method of validating modeled variability in Software Product Lines (SPL) using first order logic. They detail a multi-layered approach to modeling and validating system variability that separates presentation and logical concerns. Their work differs from our own in that they address variability across domains within the SPL product family. While our overall architecture does facilitate variability across domains, this work focuses on intra-domain variability and the analysis and validation of operational constructs previously deemed valid for a given domain. Additionally our approach is designed to facilitate runtime validation that allows dynamic augmentation of domain specific functionality.

Cho et al. [15] developed a specification language based on temporal logic for the validation of dynamic systems. It allows for the declarative specification of properties that can be used to validate behavioral models. We find this approach to be ill-suited for our architecture as the static analysis of intent models have no requirement for a formal notion of ordering, which factors into the authors' work in developing half-order dynamic temporal logic (HDTL). While it follows that a procedure p within an intent model can only be entered from its parent procedure, therefore suggesting some level of ordering, the absence of any facilities for static or dynamic analysis of a procedure's execution leaves us with no guarantee that p will be executed, or how many times it might be called. This prevents us from reasoning about an intent model execution in accordance with a time line and we are unable to provide temporal validation of its behavior. This leaves the major tenet of HDTL, the ability to specify dynamic properties of a system utilizing a freeze quantifier, inapplicable.

Ma et al. [38] incorporate formal model checking for the validation of security policies. While there is close association with our need to validate intent models against system policies, there is a difference in the goals of the two approaches. We assume security policies have been externally verified and are valid, requiring only that our middleware behaves in accordance with them. Beyond this variance however, there are other salient differences in their approach that necessitates our work. Our architecture relies on runtime analysis of models and must facilitate validation within a dynamic environment. The authors' use of formal model checking pertains to offline validation of policies. We also do not view the execution of an intent model as resulting in potential infinite paths or behavior, as is the traditional approach taken in model checking and in the authors description of an *infinite sequence of states*. Instead, intent models are analyzed as acyclic graphs with finite state and full reachability. Essentially, the possibility of infinite execution through

loops and other path repetition mechanisms do not form a part of our reasoning in validating capabilities of an intent model. Additionally, intent models are not evaluated over time, and have no dependence on temporal systems such as LTL. We believe the authors' work merits further study however, as the approach may have application in other areas of our model generation process. Whereas our definition of a well-formed intent model addresses issues of reachability, we believe the incorporation of formal model checking in the analysis of these constraints may prove advantageous.

Finally, Frappier et al. [24] present an approach to verification of the absence property using Alloy. The authors describe a method of verifying guards that prohibit reaching a specified state until some future event. They claim that their methods provides some level of increased robustness over traditional LTL model checking. This research shares some similarities with our approach as it relates to state reachability, however we believe that the necessary abstractions required for temporal considerations increases the complexity of interpreting both our intent models and Alloy specifications. Static models prove sufficient to describe our system as we are unconcerned with change over time. We do believe however, that similar to [38], this work merits further investigation as we continue to refine our method of analyzing the behavior of intent models.

2.2.4 User-centric Communication Middleware

The User-centric Communication Middleware (UCM) is the layer of the CVM charged with ensuring the delivery of services resulting from the synthesis of a communication model by the Synthesis Engine (SE) [68, 63]. Upon completion of the model synthesis process, the SE packages and delivers a control script, which is an ordered set of commands, to the UCM. The list of control script used in the CVM is shown in Figure A.1) in the Appendix. It is the job of the UCM to realize the intent

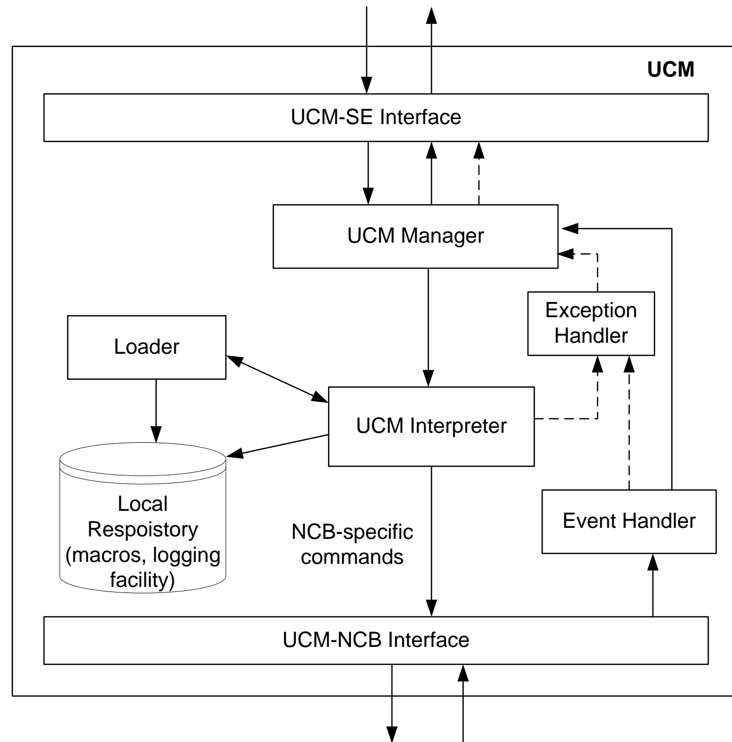

Figure 2.5: UCM Architecture

of the user by performing the necessary operations described by the commands found in the control script while adhering to the non-functional requirements of the system based on available state information. This may require the UCM to determine at runtime what the semantics of a particular command should be based on the current system context.

The current implementation of the UCM depicted in Figure 2.5 performs its functions through runtime adaptation of its operations using structural reflection [14]. Currently however, the UCM is not policy aware, and is therefore unable to incorporate context into its adaptation requirements. In its current incarnation, the DSML of the CVM must expose variability of its operations in the language and places the burden of complying with business rules and policies on the creator of the CML schema. While the CVM as a whole does incorporate the use of policies in specific layers [8], the UCM is not party to any policy knowledge. Our imple-

mentation of the DSVM middleware layer will address these issues by allowing variability in operations to be handled internally instead of passing the decision to the user. This stands to reduce complexity to the user and provide assurance in functionality by ensuring compliance with system policies.

2.2.5 Summary of Middleware Comparison

We have compared our work to several middleware designs that have been proposed in the literature. In this section we will summarize the comparison of these designs based on six features that is most important to our work. The works by Raychoudhury et al. [48] does a comprehensive comparison for common middleware for pervasive computing across three dimensions, including programming abstractions, system architecture, and system services and runtime support. Tigli et al. [57] also presents a comparison of several middleware used in ubiquitous computing, using features such as structural adaptation, behavioral adaptation, heterogeneity, and extensibility, among others. The criteria we use are different to those used by Raychoudhury et al. [48] and Tigli et al. [57].

The features we used to compare the middleware previously presented in this section to the DSVM middleware are as follows:

- *Variable Operation* - Multiple ways of realizing user intent. For example, operation to send a file may be done with or without encryption based on the user's intent.

- *Formal Validation* - A mechanism for the middleware to formally validate that the operation complies with predefined constraints. We are currently working on using first order logic to validated intent models to ensure they conform to user policies.

Middleware	Features					
	Variable Oper.	Formal Valid.	Dynamic Compos.	Funct. Aug.	Prog. Model	Granular Oper. Units
Kramer et al. [37]			✓			
Zachariadis et al. [67]			✓			
Bellur et al. [5]					✓	
Madl et al. [39]		✓				
Veríssimo et al. [60]			✓		✓	
UCM Ver. 1 [68, 63]			✓			
DSVM	✓	✓	✓	✓	✓	✓

Table 2.1: Middleware Feature Comparison

- *Dynamic Composition* - The ability to dynamically build functional constructs to realize delivery of services, which is part of the structural adaptation process [57].

- *Functional Augmentation* - The ability to dynamically add new domain-specific behavior to the middleware at runtime. Our current design allows new procedures to be added to the DSVM middleware at runtime.

- *Program Model* - Refers to the ability of the middleware to use its own programming model during the realization of the middleware services. This program model focuses on dynamically building call chains of procedures to be executed, type checking of procedures based on descriptors, managing state, and so on.

- *Granular Operational Units* - Ability for the middleware to use small execution units in the program model. This allows the middleware to reduce its resource footprint, e.g., memory and CPU time, while performing operations for the delivery of services.

Table 2.1 shows a summary of the middleware features in five middleware designs and the one presented in this paper for the DSVM.

CHAPTER 3

PROBLEM DEFINITION AND METHODOLOGY

In this chapter we describe our research problem in detail and the sub-problems that will comprise our research effort. We aim to achieve the efficient specialization of DSVMs in varying domains. The scope of our work will be restricted to the middleware layer of a DSVM.

3.1 Motivation

To date in our research group there has been one i-DMSL and DSVM that has been completely developed, CML [65] and CVM [19], respectively. Currently there is another i-DSML and DSVM for the microgrid energy management domain under development by Allison et al. [2, 3], *Microgrid Modeling Language* (MGridML), and *Microgrid Virtual Machine* (MGridVM), respectively. The prototypes for both DSVMs were built without exploiting any of the commonalities that may exist in the model of execution for the various layers in their DSVMs. Ideally we would like to define a mechanism to encapsulate domain-specific knowledge (DSK) and combine it with a generic model of execution (GMoE) in order to efficiently instantiate a DSVM for a specific domain (see Figure 3.1 [18]). The method for accomplishing this task varies by layer as each layers' view of the executing model is of varying level of granularity and its operations more platform specific.

Since the middleware of a DSVM is responsible for ensuring the delivery of services in a specific domain, the semantics to support such an operation must allow the implanting of domain knowledge at the time of instantiation of the middleware. In addition to this requirement, our middleware design must incorporate multiple methods of realizing user intent, and define a mechanism of selecting an execution path that complies with relevant policies. To achieve this, we must define a way

to classify operations that will allow the middleware to determine what operations are applicable for specific intentions, as well as the high level attributes that these operations are concerned with.

The middleware design we are proposing for DSVMs should provide most of the services identified by Vinoski [61], and yet have a structurally similar to that presented by Schmidt et al. [52]. The middleware design presented by Schmidt et al. shows how services are organized into contextual layers, these layers are as follows:

- *Domain-specific services* - services tailored to the requirements of specific domains, e.g., telecom, microgrid, and health care, among others.
- *Communication services* - provides the applications layers with the code required to develop distributed applications using the lower-level middleware.
- *Distribution services* - provides a higher-level distribution programming model to the communication services layer.
- *Host infrastructure services* - encapsulates and enhance native OS mechanisms, e.g., interprocess communication, concurrency, and synchronizations of objects among others.

Since our middleware will be used in a cross-section of domains, the composition of services provided by the middleware will vary depending on the domain. For example, in the CVM there is a need for many of the services associated with a distributed environment, while in the MGridVM this is not the case, since the current MGridVM design is mainly centralized. Figure 3.1 illustrates the need for a highly configurable middleware design that can be easily composed during the instantiation of the DSVM.

In addition to having a highly configurable middleware, there is also a need for the individual middleware services in a given layer to be adaptable. As an i-DSML

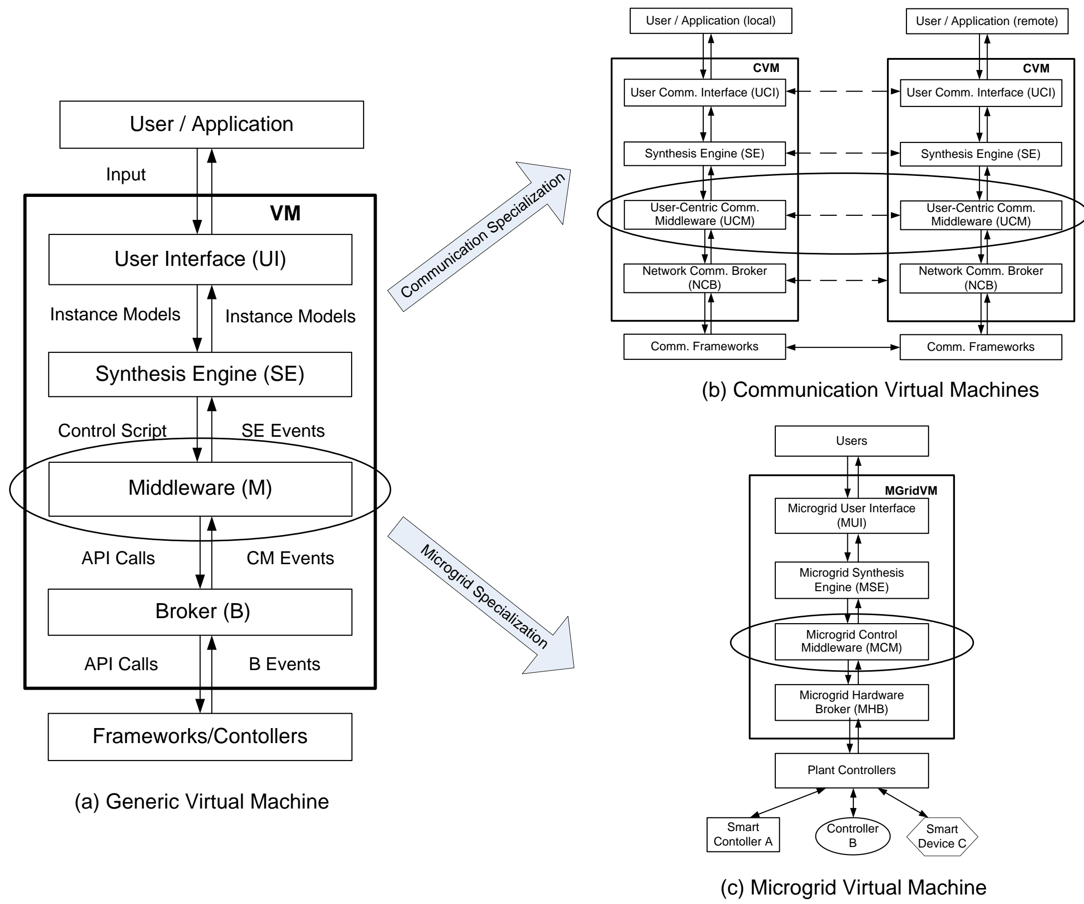

Figure 3.1: DSVM specialization during instantiation.

is a declarative language, it states non-functional requirements without specifying how these constraints should be applied to the functional requirements at runtime. Due to this limitation, our middleware design must have the capability of dynamically determining the operational semantics of a command to be executed based on the system context and policies that are in place.

To incorporate non-functional requirements, we enumerate all possible execution paths based on available resources, and then select an execution path based on it's adherence to the stated constraints at runtime. This presents us with the challenge of ensuring that one execution path matching a particular command is operationally equivalent to all others in terms of it's final outcome. Our middle-

ware must ensure that the chosen execution path is representative of the user's intent while at the same time ensuring adherence to all system constraints.

One other characteristic mentioned by Schmidt et al. [52] is affordability. As a DSVM is created for a new domain it is would not be cost-effective to build the middleware for that DSVM from the ground up using a stove-pipe approach. To mitigate the cost to develop a DSVM middleware, our middleware design should separate the *Domain-Specific Knowledge* (DSK) from the *Model of Execution* (MoE), thereby making the MoE reusable. In addition, if the DSK is separated from the MoE, it may be possible to capture the domain knowledge during feature analysis [33] of the domain. Assuming the MoE can be successfully applied in multiple DSVMs then it can be classified as a generic model of execution (GMoE).

Our focus for this research will be to develop a DSVM middleware design that is configurable during instantiation, provides adaptation at runtime, and separates the DSK from MoE. We will show how this design will be applied to the CVM middleware. Our future work will show how the generic model of execution can be applied in multiple domains.

3.2 Problem Statement

The problem we investigated was *how to design a DSVM middleware layer so that the DSK can be separated from the GMoE, it supports adaptability at runtime and effectively provides the delivery of services in the domain.* As the operational variability should be constrained by policies present within the system at runtime, the architecture had to posses the ability to validate a chosen path of execution against policies to ensure compliance. Our aim was to be able to encapsulate the operations and data of a domain in which a DSVM can be deployed, and reduce the engineering effort involved for the middleware's instantiation. Additionally, we aimed to efficiently perform runtime validation of the middleware's operational artifacts.

3.3 Categorize and Encapsulate Domain-Specific Knowledge (DSK)

In order to allow our middleware to generate and select varying execution options in realizing user intent, we required a mechanism to categorize operations, allowing us a facility to generate various options for execution regardless of their semantic makeup. Having various execution paths that fall under the same operational category gave us the ability to select specific paths based on whether they are apropos in light of any currently active policies.

3.3.1 Encapsulation of Domain-Specific Knowledge

The first sub-problem we investigated was the development of a set of artifacts that can fully capture DSK within a DSVM middleware.

Goal. Develop a set of operational artifacts that facilitate the dynamic composition of operational constructs that will serve as an execution plan to realize user intent. These operational constructs, as well as high level attributes, will be classified by a system that allows us to categorize and reason about the functionality of the middleware. We will provide an operational dialect that will allow our execution model, DSK encapsulating artifacts, policies, and other features of the system to refer to the operations of the middleware.

Evaluation Criteria. We demonstrated that:

1. Our artifacts are able to fully capture and describe relevant operations and attributes found within a domain.

2. Our DSK encapsulating operational constructs and attributes have sufficient flexibility to allow for variability in operations

3. Our artifacts facilitate validation at runtime of operational constructs within our architecture against ECA policies.

4. Our approach, when compared to the middleware in the previous CVM prototype, produced the same output for comparable control scripts.

Criterion 1 focused on capturing state and behavior information in the middleware through a classification system. Criteria 2 looked at the ability to achieve variability in realizing user intent. Criterion 3 showed how our operational constructs and a classification system facilitated the validation of an operation prior to adaptation. Criterion 4 demonstrated the operational equivalence of our approach to the existing UCM in terms of deterministic operations.

Methodology. To develop our DSK encapsulating artifacts, we undertook the following activities:

1. Reviewed the research literature on programming and computational models and identified constructs that were applicable in creating our set of operational artifacts.

2. Developed a generic mechanism to label operations (behavior) and attributes (state) of a system

3. Identified a naming mechanism to ensure uniqueness within a domain

4. Extended definitions to capture operational parameters

5. Defined the operational semantics for the above system of operations and attributes that support the delivery of services in a given domain.he current implementation of CVM.

6. Developed a prototype of the middleware for CVM (UCM) that can be used to validate the semantics against t

3.4 Develop a Generic Model of Execution (GMoE)

Having developed a mechanism to encapsulate domain knowledge, we needed to have a method to interpret and effect change based on these artifacts. We required a model of execution that, when combined with the DSK, could be instantiated to a fully operational middleware. This execution model provided all the facilities needed to perform the operations found in a given domain. It, in effect, provided a sufficiently functional environment as would be provided in a static, single domain middleware architecture.

3.4.1 Generic Model of Execution for a DSVM Middleware

The second problem we addressed was the development of a domain independent platform that provides the generic model of execution (GMoE) for a DSVM middleware.

Goal. Develop a GMoE for a DSVM middleware that will facilitate the generation and execution of domain-specific intent models. Given a set of domain-specific operational artifacts and the GMoE platform, a DSVM middleware will be instantiated that can interpret control scripts from the Synthesis Engine to ensure the delivery of services for that domain.

Evaluation Criteria. Demonstrate that our instantiated platform can:

1. Reliably generate intent models that are able to interpret control scripts from at least two domains.

2. Incorporate cost analysis capabilities for the selection of policy compliant models for execution

3. Execute generated models for a cross-section of control scripts from at least two domains.

To evaluate our instantiated platform, developed and executed a cross-section of control scripts representing various scenarios from the user-centric communication and microgrid energy management domains. We analyzed the model generation, validation and selection process detailed in Criteria 1 and 2. Criteria 3 focused on the ability to fully realize user intent through the execution of the selected model.

Methodology. To develop our domain independent platform, we did the following:

1. Identified and developed the artifacts required for runtime representation of DSK form two domains, user-centric communication and microgrid energy management.
2. Developed an efficient mechanism to generate and select models based on available operational constructs, active system policies, and operational cost analysis, respectively.
3. Identified and developed the mechanisms required for the execution of models both in a stand-alone and distributed environment.

3.5 Define a Mechanism for Efficient Functionality Validation

In order to ensure that our generated execution paths complied with applicable policies, we defined a mechanism to validate them. As our models are described using operational classifiers, so too does our mechanism for validating their operations. We ensured that within the generated model space, any model deemed appropriate for execution fully complied with any policies in place for the intended operation.

3.5.1 Validation of Intent Models

The third problem we addressed was the development of an efficient method of model validation using model checking techniques.

Goal. Develop a mechanism using model checking to efficiently validate generated models to determine their appropriateness for execution in consideration of the policies currently active within the system.

Evaluation Criteria. Demonstrate that our validation method can:

1. Represent ECA policies in a format that supports model checking at runtime
2. Represent the intent models created by the GMoE and DSK artifacts in a manner that supports model checking at runtime.
3. Perform model validation by checking for satisfiability of intent models against active policies, using the representations previously stated, for DSVM middleware in at least two domains.

Criteria 1 and 2 focused on the transformation of policies and intent models into an appropriate representation given the constraints of a running system. Criterion 3 looked at performing satisfiability checks to verify that a model respects the policies of the system.

Methodology To develop an efficient model validation process, we undertook the following:

1. Identified an appropriate technique for formal model checking at runtime.
2. Developed an algorithm to dynamically transform intent models into a representation that supports runtime model checking.

3. Developed an algorithm to transform ECA policies into a representation that supports runtime model checking, e.g., logical formulas.

4. Identified and developed tools to perform satisfiability checks on intent models and policies in the DSVM middleware in two domains, user-centric communication and microgrid energy management.

CHAPTER 4

DOMAIN KNOWLEDGE ENCAPSULATION

In this chapter we present our taxonomical labeling mechanism that provides the framework for domain knowledge encapsulation, and the operational constructs that capture the behavior of our middleware's operations.

4.1 Overview

Our middleware architecture achieves service delivery through the execution of a control script (see Figure A.1 in the Appendix) received from the synthesis engine of the DSVM. Upon receipt of a control script, the middleware parses and extracts the script's individual commands. The middleware then proceeds to generate and execute intent models to carry out the functions of the control script, see Figure 4.1.

Upon execution, intent models perform the necessary adaptation of our middleware based on current policies and the environmental context. It generates, validates and executes intent models in response to commands contained in a control scripts or from events received by the middleware. Our approach, by design, ensures that any model that is selected for execution to carry out a user's intent fully

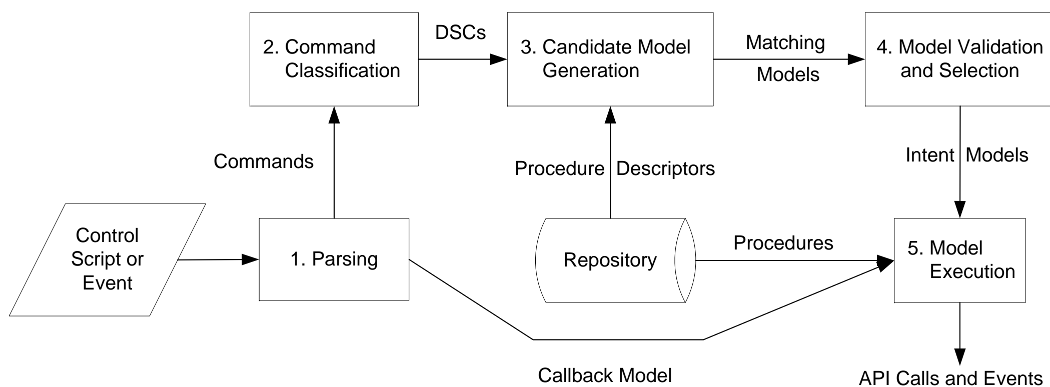

Figure 4.1: Model Generation and Selection Process

conforms to all constraints the system has in place. It achieves this through the full classification of the middleware's operations, the generation of runtime models based on the classifiers, and finally, validating and selecting a model for execution based on whether or not a model incorporates the features necessary to meet system constraints. This facility allows our middleware to only perform requested adaptation if it is able to do so within the current environment and with the available procedural components. If the middleware lacks the proper procedure to meet stated constraints, it throws an exception to the overlaying layer.

The stages in Figure 4.1 showing the model generation and selection process are described as follows:

1. Command Classification matches a command to a Domain Specific Classifier (DSC) in order to begin the model generation process. The relationship between DSCs and commands is discussed further in Section 4.2.

2. Candidate Model Generation enumerates all possible candidate models that are able to realize the current intent based on the set of available procedures. This process is bound by the Maximum Partition Product, which is discussed in Section 5.1.

3. Candidate Reduction and Selection first derives the subset of available models that conform to system policies. This process satisfies the assurance problem by ensuring that all resulting models match the user intent and fit all current system constraints imposed by policies. The resulting models are then passed to a cost function that selects the best model based on some predefined analysis.

4. Model Execution utilizes the selected intent model as an execution plan. This is discussed further in Section 5.3.2.

4.2 Domain Knowledge Encapsulation

Below we detail the various constructs that facilitate the encapsulation of domain-specific knowledge.

4.2.1 Domain Specific Classifiers

Domain Specific Classifiers (DSCs) form a top level taxonomy that categorizes the actions performed by the middleware (behavior) and the attributes that it is concerned with (state). Through this approach, DSCs catalog the domain specific concerns of the middleware and provide a framework on top of which all operational facilities will be built. They provide a common point of reasoning for commands, procedures and policies governing the middleware operation.

Classifiers have a one-to-one relationship with the middleware's commands, and a one-to-many relationship with procedures. That is, a classifier may have more than one procedure that can carry out a specified command. All commands to which the middleware can respond must form a subset of the set of DSCs. An intent model for adaptation is therefore derived by building a dependency tree of procedures with the root being a procedure that matches the classifier of the command being executed. The set of DSCs are extensible. This allows the capabilities of the middleware to be expanded or reduced by manipulating classifiers and associated procedures.

DSCs inherently possess semantics relevant to a domain and should therefore be composed through a feature analysis of said domain, and understood by the users of the middleware and containing DSVM. For example, in the Communication Virtual Machine (CVM), there are DSCs that describe command actions that the middleware is equipped to realize (e.g. Sending a file to a remote user), while others may reference internal middleware operations that are not explicitly exposed through the DSVM's modeling language (e.g. Encrypting a file to ensure secure

transmission). Additionally, DSCs may describe attributes that are applicable to its operation, but may not have meaning outside of the middleware (for example, the keys used to encrypt and decrypt a file).

DSCs that describe operations may, as a part of their definition, state parameters that are required for the completion of those operations. These parameters are themselves DSCs. For example, the DSC that describes sending a file, *Send*, would include a DSC for the name and path of the file to be sent, *FileURI*. Therefore, any procedure that conforms to the *Send* DSC must expect and handle the *FileURI* parameter.

A DSC is defined as a 4-tuple (N, NS, P, K) where:

- N - a name is a string that is unique in a given namespace (NS).
- NS - a namespace defines the scope for a given name.
- P - a list of parameters, an ordered set of *DSCs*. The order of parameters is used to match the parameters of the commands in the control scripts.
- K - a kind, such that $K \in$ {*ATTR*, *OPER*}, *ATTR* represents an attribute and *OPER* an operation.

DSCs have a one-to-one mapping to the commands a middleware can respond to, that is, for each command that may appear in a control script there is one associated DSC. On the other hand DSCs have a one-to-many mapping to procedures used to realize actions in the middleware, where a given DSC may have several procedural implementations to realize a command in the control script.

Table 4.1 presents a set of DSCs for the communication domain. The first column lists the name of the DSC along with any parameters if the DSC is an operation. The second column lists the type, which can be either an Operation (oper) or Attribute (attr).

Name	Kind
CommunicationModel	attr
FileURI	attr
Send(FileURI)	oper
Receive(FileURI)	oper
plainTextFileURI	attr
encryptedFileURI	attr
Encrypt(plainTextFileURI, encryptedFileURI)	oper
Decrypt(encryptedFileURI, plainTextFileURI)	oper
localNetwork	attr

Table 4.1: A set of DSCs for the user-centric communication domain.

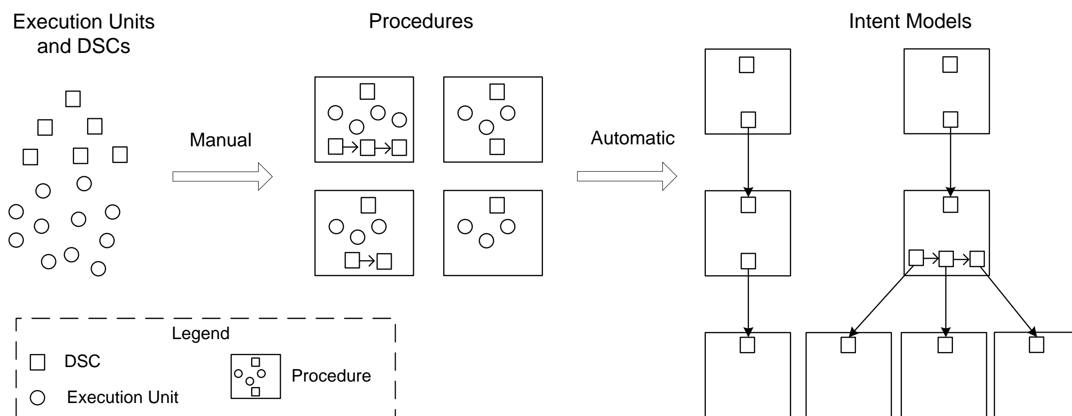

Figure 4.2: Composition of an Intent Model

4.2.2 Structural Metamodel

In this subsection we detail the constructs of our intent models and their relationships, as illustrated in Figure 4.2. The three components are *intent models*, which are composed of *procedures*, which are in turn composed of *execution units*. A procedure's execution units are specified at design time by the developer. An intent model is created at runtime by matching procedures based on the classifier and dependency DSCs.

A *Procedure* (P) is defined as a 6-tuple (I, N, C, EU, EU_0, D) where:

- I - the unique identifier for P

- N - the human readable name of P
- C - a classifier for the procedure's function, where $C \in \{DSC\}$, and $\{DSC\}$ is the set of all domain specific classifiers for a given domain
- EU - the set of execution units contained in P
- EU_0 - is the starting execution unit where $EU_0 \in EU$
- D - is a list of dependencies in P expressed as DSCs, where $D \subset \{DSC\}$ and $\forall d \in D, d \neq P.C$; $P.C$ refers to the DSC that classifies procedure P.

From an implementation perspective, we may view a procedure as an ordered collection of executable units, which may list a set of procedures on which it depends to perform its task. It is comprised of two parts.

- The descriptor, which provides the necessary meta-data for the procedure including name, the unique identifier, classification, starting component, and dependencies.
- The set of executable units that undertake the operations of the procedure including manipulating state information, making API calls, and calling for the execution of other units and procedures.

The set of dependencies of a given procedure is a proper subset of the set of DSCs. This is because a procedure of a given type cannot be dependent on that same type. To reduce potential complexity, we define a procedure as having only one classifier. Our architecture describes procedure dependencies through DSCs (typed), as well as through their IDs (named). By utilizing DSCs, a procedure can simply declare what type of functionality must exist within the middleware for it to perform its function. In contrast, when a dependency is expressed via an ID, a specific procedure will be selected.

An *execution unit* is an atomically executable set of instructions that performs some aspect of the operations of its parent procedure. It may perform any number of allowed system operations; however it should be limited to making a single API call to any external interface. This constraint facilitates a high level of adaptability as the operations of the parent procedure are granulated in terms of their effect outside the middleware. An execution unit may be triggered as the initial step of a procedure, or in response to internal or external events, such as a timer or a message from a remote middleware instance respectively. It should be noted that we detail execution units here only for the completeness of the metamodel as they do not factor into the model generation and selection process.

An *intent model* is an acyclic directed graph where the nodes are procedures and the root of any subtree is dependent on its child procedures. The root of an intent model is a procedure whose DSC matches that of the currently executing command. The composition of an intent model is discussed further is Section 5.1. For safety and to reduce complexity, we define a well formed intent model as meeting the following criteria:

- Singly classified - This prevents a node from having multiple parents.
- Unique procedural dependence - A model must have only one dependence of a given type. As such, we speak of the set of dependencies to infer the nonexistence of duplicates.

Figure 4.3 is an UML class diagram showing the structural relationships between intent models, procedures, DSCs and execution units. An intent model is an aggregate of procedures hence the relationship `procedureList`. There is a unique mapping between a DSC and a procedure through which the procedure is classified, and a procedure has an ordered dependency list of DSCs. Each procedure also

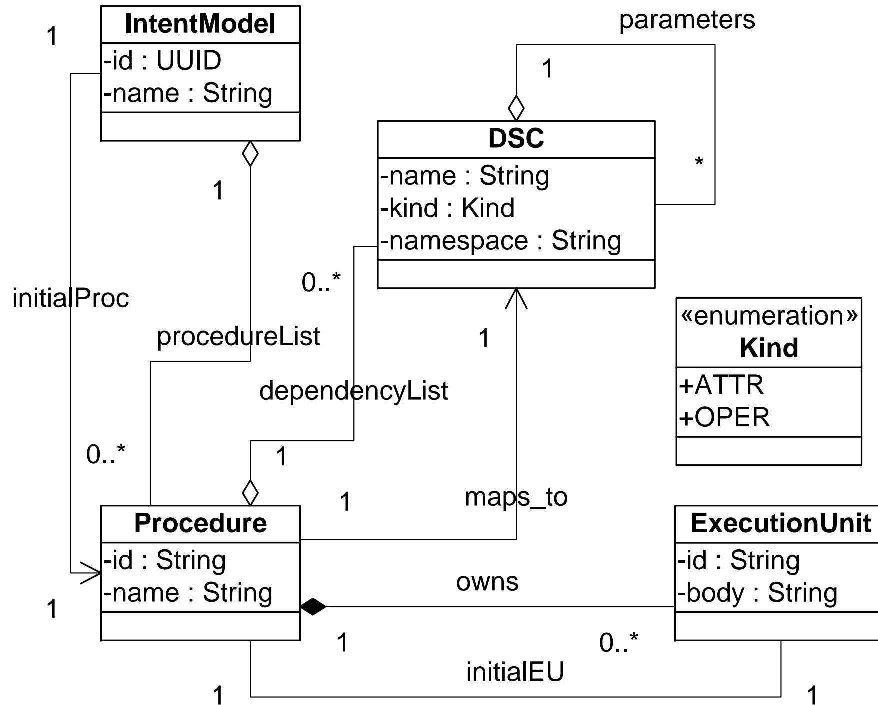

Figure 4.3: Class Diagram for an Intent Model.

owns one or more execution units. Recall a DSC may be classified as an attribute or an operation, which is consistent with the definition of DSCs.

4.3 Chapter Summary

This chapter outlined the artifacts present in our architecture for capturing the knowledge of a particular domain. This knowledge is captured though the use of a labeling system, Domain Specific Classifiers (DSCs), that presents a method to categorize the behavioral constructs and relevant high-level state information of a domain, as well as providing a description of the interfaces that exist for operations. Further, the architecture utilizes a construct referred to as a *Procedure* to undertake the operations of a domain. Procedures contain a descriptor with metadata regarding its type, makeup, and dependencies, and functions through the execution of *Execution Units*, which are written in the language of the implementing

platform. In the next chapter we will present the Generic Model of Execution that, when instantiated with a set of domain-knowledge encapsulating artifacts, provide the domain-specific middleware of the DSVM.

CHAPTER 5

GENERIC MODEL OF EXECUTION FOR A DSVM MIDDLEWARE

In this chapter we discuss our work towards a generic model of execution in the form of a domain independent platform for model execution.

5.1 Model Generation

Intent models are built using the concepts defined in Section 4.2.2. Figure 5.1 is a class diagram of the model generation component of our architecture. It lists the salient components such as the `Generator`, `Validator` and `Selector` abstract classes, as well as the representation of our intent model makeup that includes procedures and execution units.

Intent models are generated by invoking the `generateModels()` in the `Generator` class. The details of the method are shown in Algorithms 1 and 2. The system

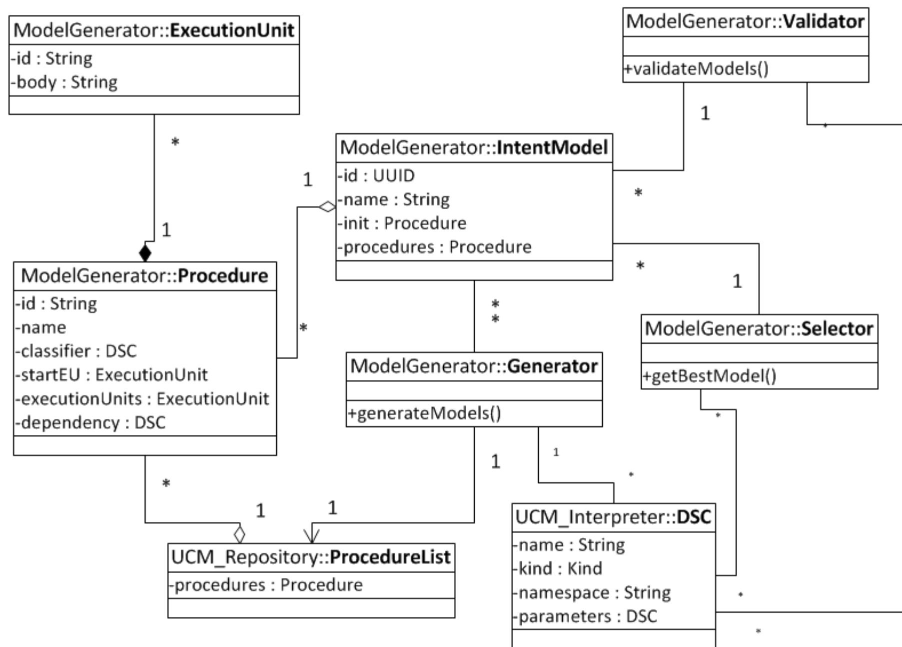

Figure 5.1: Model Generator Class Diagram

first loads all procedures from the repository that match the DSC of the command being executed. These procedures become the root of the soon to be generated intent models. For each loaded procedure, the system looks at the stated dependencies, loads all procedures that match those dependencies, and spawns a new intent model for each match with the dependent procedure added as a child of the initial procedure. This action is performed recursively until all dependencies are met.

The selection of procedures for inclusion in an intent model is akin to a model-driven approach to the Strategy design pattern [25]. The DSC that classifies a procedure serves as the interface to be implemented, and the procedure serves as the concrete implementation. Our approach varies in that all procedures that match a DSC are enumerated and included in all possible models that call for that functionality. It is only later, during model validation, that a model inclusive of a selected procedure is decided upon based on system context and the overall cost of the model. The application of design patterns in model driven development is discussed by Stahl et al. [55].

An intent model is a dependency tree that contains procedures as nodes, with the initial executing procedure as the root of that tree. A procedure with stated dependencies form the root of any subtree, and its children are procedures that match said dependencies. The leaves of the tree are procedures that have no stated dependencies. An intent model is built by matching the DSC of a command to the DSC of all available procedures. For a given procedure, we recursively walk its dependencies until no additional dependencies are required. If any dependency cannot be met, that model is eliminated from consideration.

Intent models incorporate a model-centric approach to the Facade design pattern [25] where the DSC that matches the command being executed inherits the

Algorithm 1 Intent model (IM) generation.

```
1: function generateIMs(initDSC, procList)
2: /* initDSC - DSC associated with the control script command being executed
3:    procList - list of procedures in middleware repository */
4:     matchingIMs ← null /* Collection of IMs that will be returned */
5:     matchingProcs ← procList.getMatchingProcs(initDSC) /* Get procedures for
6:     current DSC */
7:     if matchingProcs.isEmpty() then
8:         return null;
9:     for all proc ∈ matchingProcs do
10:        tempMatchingIMs ← null /*Temporary collection of IMs for current level */
11:        dependDSC_List ← proc.getDependency()
12:        if dependDSC_List.isEmpty() then
13:          /*If no dependencies, return IM with current procedure */
14:            matchingIMs.add(new IM(proc))
15:        else
16:            subIMs ← null /*Stores the sub IMs returned from the recursive call*/
17:            countDSC ← 0
18:            for all dependDSC ∈ dependDSC_List do
19:                countDSC ← countDSC + 1
20:              /*Recursive call to generate new IMs for each DSC in the dependency list */
21:                subIMs ← generateIMs(dependDSC, procList)
22:                if subIMs ≠ null then
23:                    if countDSCs = 1 then
24:                    /*For the first DSC in the list create a new IM */
25:                        singleIM.add(new IM(proc))
26:                    /*Merge the new single IM with the sub-IMs */
27:                        tempMatchingIMs ← mergeIMs(singleIM, subIMs)
28:                    else
29:                    /*Merge the temporary matching IMs with the sub-IMs */
30:                        tempMatchingIMs ← mergeIMs(tempMatchingIMs, subIMs)
31:                else
32:                    tempMatchingIMs.clear();
33:            matchingIMs.addAll(tempMatchingIMs)
34:    return matchingIMs
```

role of the interface to a subsystem. Instead of a subsystem however, the interface facilitates the execution of a sequence of procedures all of which are unexposed to the command. They are instead dependent on the context and cost-aware selection of an executable model.

Algorithm 2 Merging parent intent models and children intent models.

```
1:  function mergeIMs(parentIM_List, subIM_List)
2:  /* parentIM_List - list of parent IMs
3:     subIM_List - list of intent models for the children */
4:      newIM_List ← null /* Collection of IMs that will be returned */
5:      for all parentIM ∈ parentIM_List do
6:          /* Loops through subIM list and creates a new IM for each subIM */
7:          for all subIM ∈ subIM_List do
8:              /* Performs a deep clone of the parent IM to the new IM */
9:              newIM ← deepClone(parentIM)
10:             /* Adds the subIM procedure list to the root of newIM */
11:             newIM.addSubtree(subIM)
12:             newIM_List.add(newIM)
13:     return newIM_List
```

Maximum Product Over Partition

A consequence of our method of defining and dynamically composing models based on types is model space explosion. In theory, an idealized set of procedures may produce an excessively large number of intent models that match a particular command. While this may not prove to be a limiting factor in practice, as a typical command may only relate to a small subset of the available procedures, it is still a motivating factor for addressing optimization strategies. The issue arises due to the Maximum Product Partition problem [20], where for a given set of procedures P, there exists a partitioning based on DSCs that creates a maximal number of models.

The problem can be stated as: For a given number of procedures n, what is the maximum value of the product of parts over all partitions of n into distinct parts? Analysis shows that for a given number of procedures n, the product over its partitions φ are as follows:

$$1 * 1 * 1 * 1 = 1$$
$$1 * 1 * 2 = 2$$
$$1 * 3 = 3$$
$$2 * 2 = \mathbf{4}$$
$$4 = \mathbf{4}$$

We note the following observations:

1. It is obvious that maximizing a product negates having 1 as any of the partitions, except in the case where there is only a single partition with element 1.

2. We know from the theorem of unique factorization that every integer greater than 1 is the product of primes, or is a prime itself.

3. For $n \geq 5$, there is some partition of n such that its product is greater than the single partition n

Our third observation states that a maximal partition exists that is greater than the single partition n if $n \geq 5$. For $n < 5$, our second observation handles $n == 4$, and our first observation handles $n == 3$ by ensuring that we do not create the sub partitions $1 * 2$.

We therefore refine our analysis by looking at the product over n's partitions where no partition is 1, and n cannot be the only partition. We increase n to 8 present more useful partitions.

$$2 * 2 * 2 * 2 = 16$$
$$2 * 2 * 4 = 16$$
$$2 * 8 = 16$$
$$4 * 4 = 16$$
$$3 * 3 * 2 = \mathbf{18}$$

These partition sums show us that, for a given partition $n > 5$, we can achieve the maximum product by partitioning our set into as many subsets of 3s as possible until the remaining procedures number 4 or less. At that point we follow:

$$n == 4 \rightarrow 2 * 2$$
$$n == 3 \rightarrow 3$$
$$n == 2 \rightarrow 2$$

Therefore, the upper bound, φ, of our intent model generation process is determined by the function $f(n)$ such that:

$$f(n) = \begin{cases} n & \text{if } n < 3 \\ 3^{n/3} & \text{if } n\%3 == 0 \\ 3^{(n/3)-1} + 2^{g(n)} & \text{if } n\%3 == 1 \\ 3^{n/3} + 2^{g(n)} & \text{if } n\%3 == 2 \end{cases}$$

and

$$g(n) = \begin{cases} 2 & \text{if } n\%3 == 1 \\ 1 & \text{if } n\%3 > 1 \end{cases}$$

Where n is the number of available procedures. The proof for this method is presented in [20].

5.2 Model of Execution

Our model of execution serves as the domain independent aspect of our architecture. It must not inherently posses any domain-specific knowledge, but must be able to accommodate the domain knowledge that is provided and realize user intent through its operations.

5.2.1 Middleware Execution Model

From our design we present the execution model that allows for the instantiation of a domain specific middleware. Once we have completed the model generation process, we must be able to execute the selected intent model.

Stack Machine

The execution of an intent model is achieved through the use of a stack machine as depicted in Figure 5.8. The stack machine processes the execution units of the procedure currently on the top of the stack. A new procedure is placed on the stack when an inter-procedure call is made (whether by name or by type), and a procedure is popped from the stack whenever a null call is made, indicating that the particular procedure has completed execution. When an execution unit registers an event listener via an EventWaitCall, the entire model as well as the current state of the stack are persisted, and are subsequently reloaded upon receipt of the aforementioned event.

Execution Semantics

As execution units provide the programmatic artifacts in our middleware, they must be provided with the facilities for proper execution that allows progress through procedures, models and the ultimate realization of user intent. The facilities of our execution model are depicted below:

Memory Executing units store information as attributes in the middleware's State Manager. Although a unit may or may not utilize the standard variable management scheme available by the implementing architecture for its internal storage during execution, the execution model requires utilizing the State Manager for making information accessible to other units and procedures.

Unlike imperative programming, where variables are scoped locally within a method, our facility allows any level of scoping to be declared, as the manager exists external to the procedure's execution. As a result of this, registered attributes extend beyond the life of an executing procedure, as the necessary information may be relevant to future execution units, procedures and intent models. This is a desirable feature and is therefore the default behavior of the middleware. As a result of this, a procedure, through the applicable execution unit, must explicitly de-register attributes from the State Manager when they are no longer needed.

Calls An execution unit may effectively hand off execution to another unit based on its own executing parameters. There are three methods of making a call upon the completion of an execution unit's run.

- *EUCall* - Used for intra-procedure calls between execution units. This serves as the basic mechanism for progressing through a procedure.
- *DSCCall* - Used to make calls between procedures. An execution unit makes a call via a DSC instructing the middleware to execute the de-

pendent procedure that matched the DSC during the model generation process. This call is only successful if the called DSC was initially listed as a dependency of the parent of the calling execution unit. The call is accompanied by an execution unit that should be executed on return to the procedure once the called procedure has finished executing. There are no limitations placed on how often a procedure may call upon a dependent procedure or in what context it may do so.

- *EventWaitCall* - Used to register an execution unit of the parent procedure to respond to an event received by the middleware. Upon this call, the entire model is persisted to the state manager and is re-instantiated upon receipt of the specified event. The calling procedure is then loaded and the registered execution unit is executed.

Additionally, an execution unit may return a NULL value indicating that a procedure has completed execution. If a NULL value is returned from a previously called procedure (that is, the current executing procedure has a parent procedure in the intent model), the parent procedure is re-entered and the previously identified re-entry execution unit executed. If there is no parent procedure (that is, this is the root procedure in the intent model), then this is an indication that the model has completed execution and the Model Executor discards the intent model.

While intra-procedure calls are supported via EUCalls, an execution unit may not call another unit located in another procedure by name. Inter-procedure calls must be named or type calls and the initial execution unit of the called procedure is the unit that is executed first.

Message Passing Messages are passed from one execution unit to another, and from one procedure to another, using state information that is managed by

the middleware's *State Manager*. Messages passed between execution units of a single procedure would utilize a naming convention and semantics inherent to that procedure. Therefore, a unit y that is called by a unit x would know, by design, what information to expect from x (or any other previously executed unit) and where to find it. It then reads this information during its execution and, prior to completion, may itself save information to the State Manager before calling some unit z.

Information is passed between procedures in the same way, however in this case, there is a previously agreed upon naming convention that is described by a DSC and inherent in its semantics as discussed in Section 4.1. This is to facilitate the binding of various procedures during model generation that match a stated dependency while ensuring that they able to communicate. This agreed upon set of attributes between procedures could be likened to parameters of a publicly scoped method in an object-oriented programming model. As an example, a procedure that depends on an *Encrypt* procedure may have a set of predefined attributes: *plainTextFileURI*, which is the path and name of the file to be encrypted, and *encryptedFileURI*, which is the path and name of the now encrypted file. The inherent semantics of the *Encrypt* DSC requires these attributes, and all procedures of that type, or procedures that depend on that type, must be aware of them.

Events Our middleware allows an execution unit to respond to predefined events through registration with the *Event Register*. This facility maintains a registry of execution units and the events they are to respond to. In the case of an event arriving for which there is no registered unit to be executed, the event system triggers the model generation process in a way similar to the way it is done upon receipt of a command.

Exceptions Exceptions are a special type of event that indicate a deviation from expected middleware behavior. Exceptions generated by the middleware are passed to the Synthesis Engine (SE) to be processed or handed off to the user interface layer, generally to provide notification. For example, if the middleware is unable to generate an intent model to realize the user's intent because of a restrictive policy, the middleware's failure is reported to the SE as an exception. This exception would include relevant information such the specific policy on which the middleware failed to find a valid intent model. At this point the SE may report this exception to the user or, based on its capabilities, relax the policy and compel the middleware to reattempt the script's execution.

Concurrent Operation We view concurrent operations in two varying but similar contexts. The first is the simultaneous execution of independent models (similar to execution of multiple processes within a system), and the second, multiple executing instances of a single model (similar to multi-threaded execution of a program).

To address the first issue, we ensure that all generated intent models are assigned a universally unique identifier (UUID). This identifier is used to prefix all state and event information. By doing so, we are able to manage concurrent operations of separate intent models regardless of the model's procedural composition. That is important to ensure that if a single procedure exists in multiple executing models, there is no concern of cross manipulation of state information.

The second issue requires further consideration. Due to the event driven nature of our middleware, it is possible for multiple execution units to be triggered within the procedures in an executing intent model. Because this

behavior may be advantageous in many applications, it has not been suppressed. This however requires special considerations be taken when writing execution units. If a synchronous operation within a procedure is required, then care must be taken in execution unit design. Sanity checks should be added to ensure that a procedure is not already executing (this could be done by interrogating the State Manager) and execution unit designers should be mindful of the units they register for event responses. Registering multiple execution units for multiple events that may fire with an indeterminate order can result in multiple threads of execution for a single procedure.

Language Facilities Computational considerations are not addressed explicitly by the middleware, and are instead based on the implementation language architecture. A middleware instance build on top of Java may provide execution units with the full power of the language, whereas an implementation that utilizes a template language may provide a minimal set of execution facilities.

5.3 DSVM Middleware Design

The DSVM middleware design describes the main artifacts of the MoE that are needed to interpret a control script command, including IM generation and selection, and the execution of the intent model to realize the management and delivery of services in a specific domain. We start by presenting the structural design of the middleware that includes a package diagram of the components of the middleware, followed by other class diagrams showing the refinement of the interpreter component. We end by describing the behavioral design for the interpreter component, focusing on intent model generation and selection, and its subsequent execution.

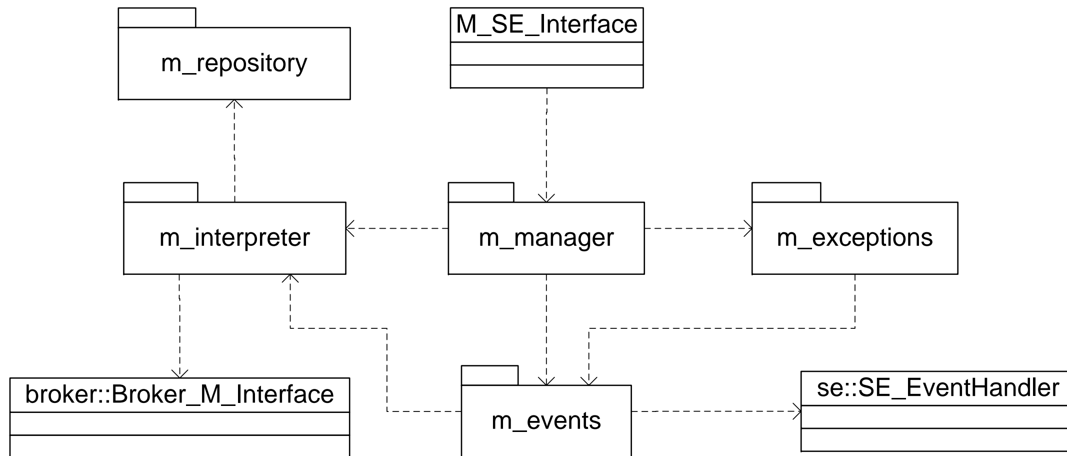

Figure 5.2: Package diagram showing the main components in the DSVM middleware.

5.3.1 Structural Design

The DSVM middleware consists of five main components as shown in Figure 5.2. These components include: (1) `m_manager` - coordinates the activities of the middleware; (2) `m_interpreter` - responsible for interpreting control scripts from the DSVM synthesis engine and the events from the broker; (3) `m_repository` - contains the DSCs, procedures and the execution units; (4) `m_events` - handles events from the DSVM broker and generates events to be handled by the synthesis engine; and (5) `m_exceptions` - processes local exceptions for the middleware and converts the exceptions to be handled by the synthesis engine into events. Figure 5.2 shows the dependencies between the five main components in the middleware.

There are three classes shown in Figure 5.2 including (1) `M_SE_Interface` - exposes the interface to the synthesis engine that receives control scripts;
(2) `Broker::Broker_M_Interface` - exposes the API of the broker to the middleware, the operations in this API are invoked by the middleware interpreter; and (3) `SE::SE_EventHandler` - is the synthesis engine handler for events raised in the

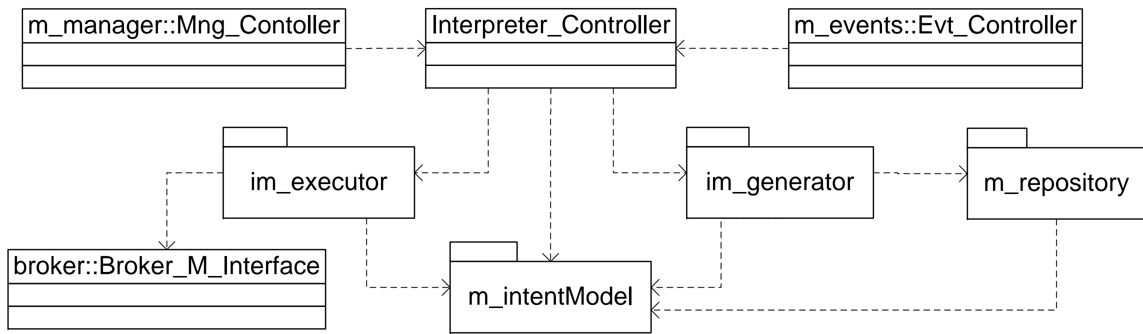

Figure 5.3: Class diagram for the `m_interpreter` package in the DSVM middleware.

middleware. In this section we will focus mainly on the generation, selection and execution of intent models, which is done in the `m_interpreter`.

Figure 5.3 shows the structure of the middleware interpreter. The activities of the interpreter are coordinated by the `Interpreter_Controller`, and it processes calls from the controllers in the middleware manger package, `m_manager::Mng_Controller`, and events package, `m_events::Evt_Controller`, respectively. The two main functions of the interpreter are to (1) generate and select intent models in the `im_generator` package, and (2) execute the selected intent models in the `im_executor` package. The `im_generator` package depends on the `m_repository` package since the definitions for the procedures, DSCs and execution units are stored in the repository. The class diagram for the intent model structure was described in Section 4.2.2.

For each control script command the `im_generator` package is responsible for generating a set of applicable intent models and selecting a single intent model, which is stored in the `Interpreter_Controller`. The `im_executor` package executes the intent model in order to realize the user's intent specified in the control script command being processed.

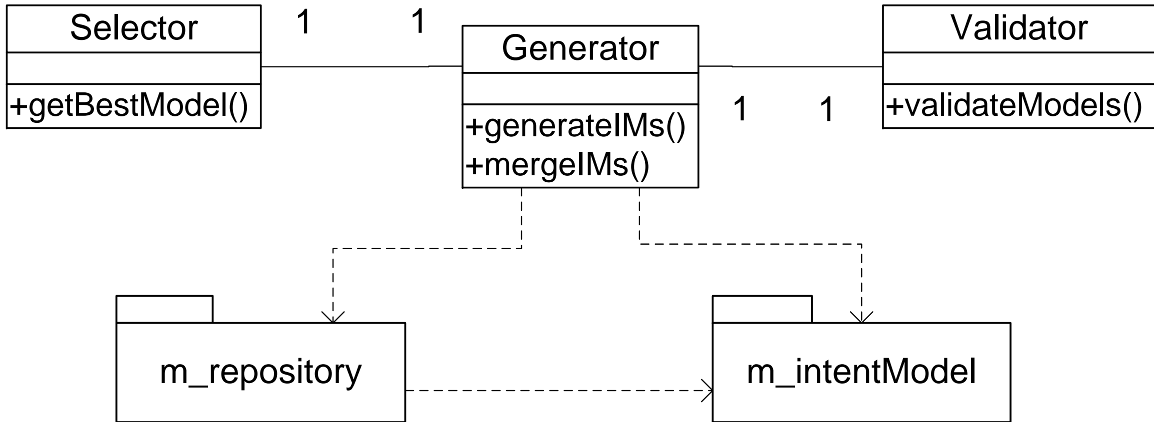

Figure 5.4: Class diagram for `im_generator` package.

Figure 5.4 shows the components used in the generation, selection and validation of intent models. The `Generator` class is responsible for generating the intent model to be used by the `im_executor` in the interpreter. Section 5 describes the process and algorithms for generating intent models. The `Generator` class accesses the list of persistent DSCs, procedures and execution units from the middleware repository, and uses the package `m_intentModel` to build the intent models. After all the intent models have been built the `Generator` class invokes the `Validator` class to ensure that each intent model satisfies the constraints placed on the intent model by user-defined or system policies. Currently, our approach to intent model validation involves walking the graph of the intent model to determine the presence or absence of a procedure classified by a specified DSC. We plan to develop a more robust method of validation in future work.

After building the set of intent models based on the control script command received by the middleware, the `Selector` class identifies the most appropriate intent model for execution through a domain-specific cost analysis mechanism. For example, in out prototype we define the function $f(M)$ such that

$$f(M) = \sum_{i=0}^{n-1} cost(M_i)$$

where n is the number of procedural nodes in the model and $cost()$ takes a procedure M_i and returns a deterministic cost associated with it.

5.3.2 Behavioral Design

In Section 4.1 we gave an overview of first-class operations of our middleware. The model generation process populates the model space with all possible intent models that can perform an operation based on a command or event. We detailed this process in Section 5.1. The selection process ensures that we select a model for execution that fully complies with policies currently active within the system. Our domain independent platform, which provides our generic model of execution, facilitates these operations. Once an intent model is selected, the execution process is initialized.

This section details the domain independent platform, which is the part of the middleware that performs generation, selection and execution of intent models. Through the use of DSCs, procedures, and their execution units, we are able to abstract the DSK from our architecture. By removing the DSK, what remains is a domain independent platform that serves as the execution mechanism for procedures in response to commands (see Figure 5.5) and events (see Figure 5.6). Figure 5.5 shows the behavior of the middleware when a control script is received and Figure 5.6 the behavior when an event form the lower layer in the DSVM, the broker, is received. The platform provides a framework in which we specify a set of classifiers, and provide the procedures that perform the operations these classifiers describe.

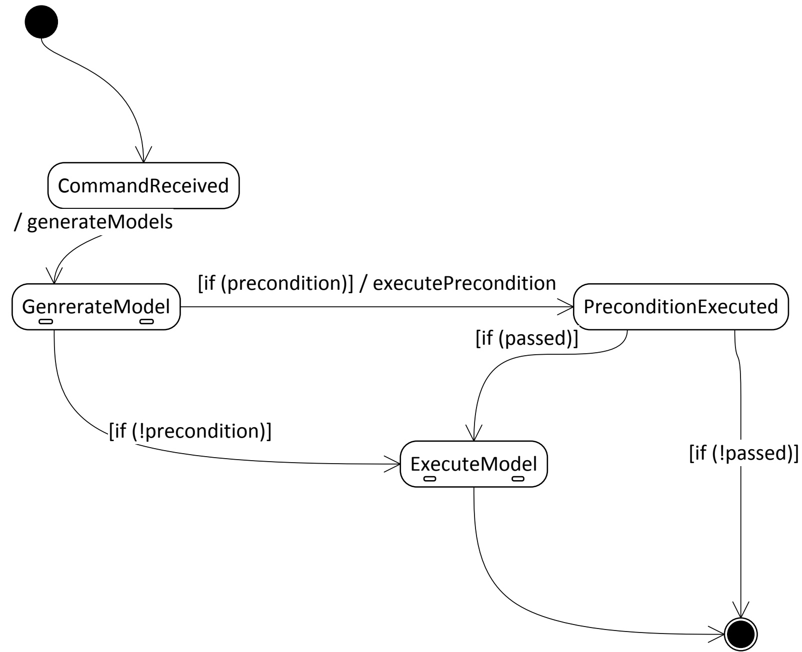

Figure 5.5: Command Received Statechart

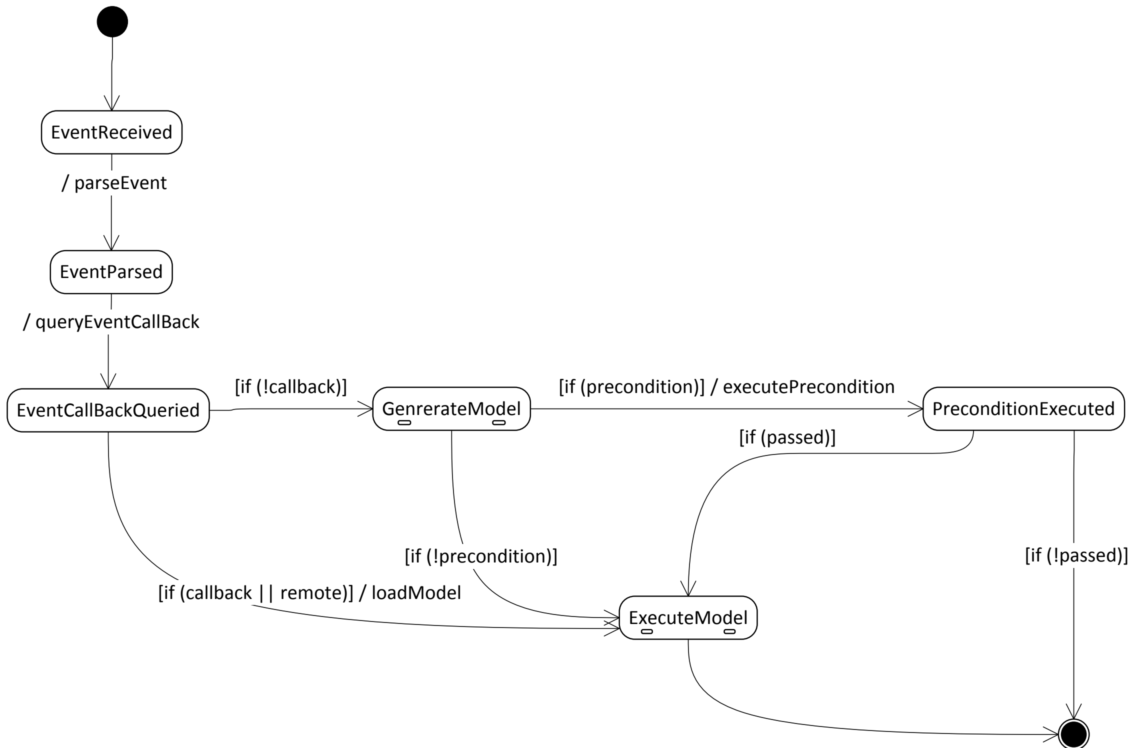

Figure 5.6: Event Received Statechart

Candidate Selection

Following the generation of a list of intent models suitable for the control script command being executed, the middleware must reduce this list to a subset of models that respect the current set of policies in place. It then further reduces this list and selects a single model for execution. The process of model validation is discussed in detail in Section 6.1. Here we discuss the selection process that follows model validation. There are a multitude of techniques for selecting the most viable intent model through the implementation of a cost analysis mechanism. We define the function $f(M)$ such that

$$f(M) = \sum_{i=0}^{n-1} cost(M_i)$$

where n is the number of procedural nodes in the model and $cost()$ takes a procedure M_i and returns a deterministic cost associated with it. We run this function on all candidate models and select the model with the minimal total cost. A specific mechanism for determining the cost of a given procedure is not defined by our architecture. This ensures that we do not constrain domains that may have varied analysis requirements. These requirements could vary both in terms of the data needed to analyze a procedure's operation, as well as the context or makeup of a model. For instance, a particular procedure may have a higher or lower operational cost based on some other procedure found in the model. We therefore define an abstract class to perform cost analysis, see the class `Selector` in Figure 5.1.

Model Execution

Figure 5.7 depicts the classes in our architecture that facilitate the execution of our models. These classes include: the `Executor` that manages control flow; the `StateManager` and `Attribute` classes that manage state information; the `EventRegis-`

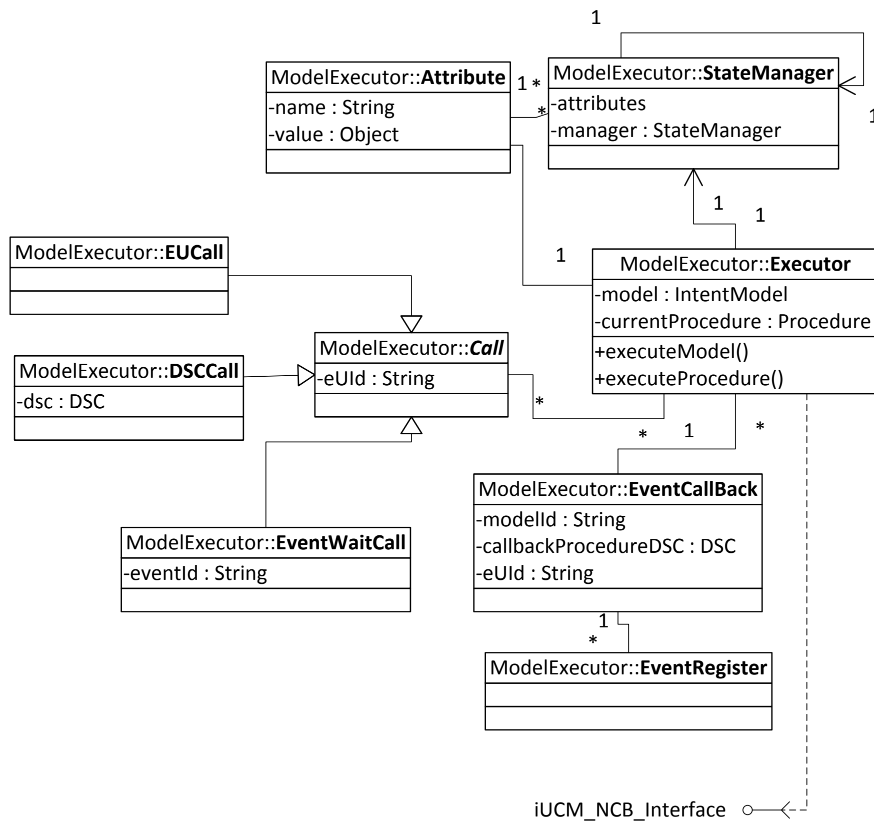

Figure 5.7: Model Executor Class Diagram

ter that handles event management; and the `EventCallBack` that handles various calls that allow execution units to progress through a model. The `Executor` is provided access to the underlying Broker layer's interface so as to facilitate API calls made by execution units.

Our Model Executor initializes execution of a model by loading the initial execution unit of the root procedure. Each unit has the responsibility of ensuring progression through a procedure by directly executing, or registering for execution, the next unit in sequence. Execution units may also call on a dependent procedure if its capabilities are required. This call to a dependent procedure may be done indirectly (via a DSC), or directly (via the procedure's ID). If a call is made via a DSC, the middleware will execute the starting execution unit of the procedure pre-

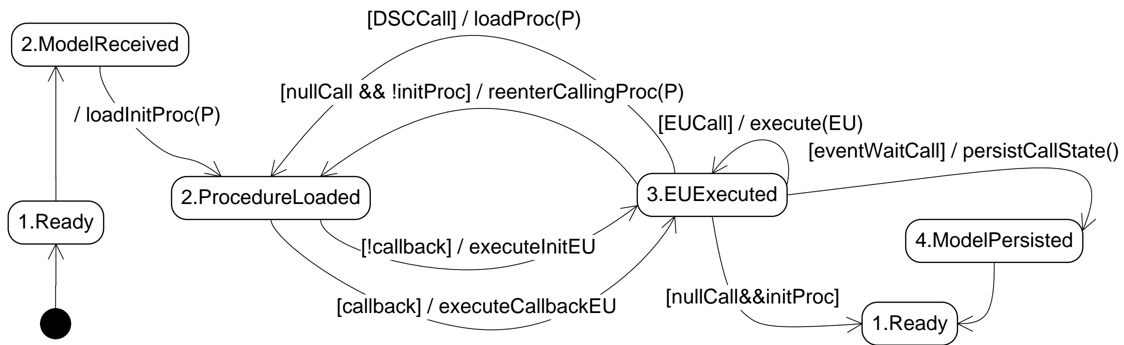

Figure 5.8: Model Execution Statechart Diagram

viously identified during the model generation process. The dynamic semantics of this process is shown in Figure 5.8.

Our architecture is able to facilitate distributed execution via special control messages passed via middleware specific events that are supported by the underlying layer. These messages contain information on the executing procedure, and the current state information on the executing node. Unlike the work of Al-Jaroodi et al. [1], our control messages are not predefined, but are instead dependent on a particular procedure's implementation. The packing and unpacking of control messages are handled by our middleware, while marshalling and dispatch are handled by the underlying broker layer. This relieves the middleware of concerns relating to guaranteeing message delivery as external communication with remote DSVMs instances fall to the broker layer.

Upon receipt of these control messages by a remote instance of the middleware, the provided data is first unpacked and registered in the state manager; subsequently, the procedure matching the UUID is loaded into a single-procedure model and executed. A procedure that is written for distributed execution would, upon initial execution, check whether it is being executed on the initial local host, or on a remote host by the presence of absence of its predefined control data.

State Information and Access Control. The middleware must manage state information that is used to coordinate inter and intra-procedural operations. It accomplishes this through the use of the *State Manager* that manages key/value pairs that can be DSC attribute values as well as data generated by executing procedures.

EU Register. In order to facilitate distributed procedure execution (a procedure executing in response to continuous events from a remote instance of the middleware, e.g. during negotiation), the middleware must allow a procedure, or more specifically its execution units, to respond to events. Our architecture facilitates this by maintaining an register where execution units are registered to respond to specific events, including those generated in response to the action of remote middleware instances.

Repository. The repository of the platform houses the procedures, their execution units, and the DSCs that describe them. It is possible to populate the repository offline or at runtime. Runtime population may come as a result of incorporating an Advertise and Discovery mechanisms that would allow a middleware instance to obtain new procedures from other instances of the middleware or from an online repository accessible by the platform [66].

5.4 Chapter Summary

In this chapter we detailed the facilities of our Generic Model of Execution, which facilitates the realization of user intent in a particular domain via it's specialization through the provision of domain-specific artifacts. We showed the various subsystems of the platform that undertook the generation, validation, selection and execution of intent models, as well as the programming model, which allows procedure writers to exploit the platforms resources. Additionally, we detailed the interfaces present within the current design that allow for the extensibility of the

aforementioned first-class operations based on present design considerations. The next chapter explores a method of validating generated intent models for compliance with DSVM policies.

CHAPTER 6

VALIDATION OF INTENT MODELS

In this chapter we present our work towards runtime validation of intent models to guarantee correctness and policy compliance. Our method utilizes the Alloy Analyzer and specification language to facilitate formal checking of intent models, which provides robustness, efficiency, and automation.

6.1 Model Validation

Our approach to intent model validation requires the translation of a model's representation in our middleware architecture to that required by the Alloy Analyzer. The analyzer uses the Alloy Specification Language [30]. Table 6.1 gives the mapping used in translating middleware intent models into Alloy models. The translation process is outlined in Algorithm 3.

Translation Method

We define our DSCs as abstract signatures in our model specification. This allows us to express our procedures as extensions of their DSC base classes, as well as allowing us to describe our policies utilizing type abstractions in Alloy in much the same way as we do in our middleware architecture.

To express middleware policies, we utilize Alloy *assertions*, which allow us to *claim* that a property or set of properties hold for the specified model. Assertions in Alloy utilize a representation of first-order logic that facilitates references to artifacts in the defined model. Each policy can be expressed as its own Alloy assertion, or they may be joined together using logical conjunctions into a single assertion.

Once the set of assertions are described, they are verified using Alloy's *check* command. If the assertions hold, Alloy will be unable to generate any configu-

Algorithm 3 Model Translation

IN: Intent Model (IM)
OUT: Alloy Model
1: **for all** Procedure p in IM **do**
2: $new\ abstract\ Sig : t = p.DSC$
3: $new\ Sig :\ s\ extends\ t = p.Name$
4: **for all** DSC dsc in p.Depencencies **do**
5: $newFact : s.contains \longleftarrow sig(dsc)$
6: /*
7: Alternatively, may utilize properties for simple relationships
8: */

rations that contradict them. If our assertions do not hold, Alloy will generate a counterexamples that meets the given specifications, but contradict the assertions. Since the specification is restricted to only generate the single model under analysis, the existence of a counterexample is indicative of that model being invalid. Such a model can then be discarded from our candidate list for the current command as its execution would not respect middleware policies.

It merits stating that this behavior in Alloy is a direct result of the restrictions placed on our model translation process. In the general case, while the existence of a single counterexample is evidence of a model's invalidity, Alloy will generate all possible counterexamples within scope to demonstrate the various invalid aspects of the model.

The definition of a well formed intent model states that all DSCs, and their implementing procedures, must be unique within the model. This reduces the complexity involved in ensuring liveness when generating and executing these models. This restriction also possesses the side effect of rendering the choice of quantifiers as immaterial in the process of translation policies into assertions. The uniqueness of DSCs ensures that $\exists \equiv \forall$ since the assertion's validity is unaffected by the choice.

We restrict the generation of multiple and superfluous configurations by utilizing the *one* keyword when defining signatures. This ensures that only a single instance of a signature will be present within the alloy model instance. This guar-

antees that the signature can only be associated with a single parent, and/or a set of children in accordance with the definition of the associated intent model procedure.

In Algorithm 3, we accept an intent model and step though its member procedures. As Alloy's specification language is declarative, the order in which we translate an intent model's procedures is irrelevant. We then follow a set of predefined translation rules for the DSCs, procedures and dependencies, generating a set of Alloy signatures, properties and facts that describe the various aspects of our intent model. The result is a semantically equivalent Alloy model specification.

Algorithm 4 augments our specification by generating a set of assertions from our ECA policies. The *event* in our ECA policies are the commands being executed by the middleware. When responding to and event, we are interested in two forms of conditions - inclusive and exclusive - to determine the presence or absence of a particular action in response to an event E. Actions are defined by their transitive closure (denoted by ^). Use of the reflexive transitive closure (denoted by *) is unnecessary as our intent models, with each procedure being singly classified by a unique DSC, have no facility for the procedure denoted by the action to be equivalent to the root procedure.

For completeness, we ensure that a signature is created for each action specified in the Action clause of our policy. This results in disjoint objects within our generated Alloy models. Failing to perform this step however, results in the generation of errors in Alloy whenever it attempts to evaluate as assertion that references a non-existent signature. Since a signature that is not present in an associated intent model, but created to satisfy an assertion definition will, by necessity, be disjoint from the main structure that possesses the intent model's root procedure as a member, there are no side effects to their presence in our validation process. Additionally, whether or not these assertion-satisfying signatures are declared as *abstract*

Algorithm 4 Policy Translation

IN: ECA Policy Set (ECA_Set)
OUT: Alloy Assertion

```
1: new Assertion a
2: for all Policy p in ECA_Set do
3:     if ! exists(sig(pol.A.DSC)) then
4:         new Sig : t = p.A.DSC
5:         /*
6:         May be declared as abstract.
7:         Alloy creates a single instance
8:         if not extended by another sig
9:         */
10:    if pol.C is InclusiveCondition then
11:        a ⟵ ∀s : sig(p.E.DSC)|sig(p.A.DSC) ∈ s.^dependency
12:        /* May be repeated for all dependencies */
13:    else
14:        a ⟵ ∀s : sig(p.E.DSC)|sig(p.A.DSC) ∉ s.^dependency
15:        /* May be repeated for all dependencies */
```

Table 6.1: Artifact Mapping

Middleware Artifact	Alloy Artifact
DSCs	Abstract Signatures
Procedures	Signatures
Dependencies	Facts and Properties
Policies	Assertions

have no bearing on the resulting model as Alloy will only disallow the instantiation of abstract signatures if there is at least one signature that extends it. The result of our translation process is a full Alloy model specification that can be checked for validity and, in the case of an invalid intent model, generate counterexamples that can be visualized.

Model Validation

Having converted our intent models into an Alloy specification, and translating all relevant ECA policies into their equivalent assertions, the process of validating an intent model is reduced to checking a model for satisfiability with respect to the assertion or set of assertions.

This process, of course, takes place at runtime in response to commands and events received by the middleware, unlike conventional approaches that are generally undertaken as an offline process due to the properties that are traditionally checked such as safety and fairness [10]. The runtime validation of intent models is necessary as the validity of a model M in response to a event or command C, may change over time. This can be as a result of changing environmental variables that render a particular policy inapplicable. In [43] we discussed some approaches that could help address efficiency issues resulting from runtime model validation.

6.2 Chapter Summary

This chapter detailed our method of validating generated intent models for compliance with Event Condition Action (ECA) policies present within the DSVM. Our method utilized the Alloy Specification language and Alloy analyzer to facilitate formal model checking of intent models. This was achieved through the translation of intent models and ECA policies into appropriate Alloy artifacts. Once completed, the validity of an intent model could be determined by checking for the satisfiability of the model's Alloy representation against the requisite policies.

CHAPTER 7

EXPERIMENTATION

In this chapter we present a set of experiments and demonstrations that validate our stated research goals and methodologies. We have segmented the chapter into sections based on our major contributions, with overlap where necessary to show the correctness or applicability of a particular contribution.

7.1 Generic Model of Execution

This section presents some comparative results between the current UCM implementation and a prototype of our new architectural design. Our initial tests were designed to measure the potential runtime overhead of our model generation process in a realistic scenario.

Objective: The objective of the first experiment is to determine the execution time overhead for the intent model generation process in our middleware for a standard domain deployment.

Method: We designed this experiment to measure the runtime overhead of our intent model generation process in a realistic scenario. To do this, we curated a test environment where we initialized a prototype with a set of 100 procedures to simulate a typical middleware layer implementation based on our analysis of the user-centric communication domain. This analysis was performed by Wu et al. [64] and details the operations of the communication domain. Of the provided procedures, 10 were designed to either match the operation of the test command, or meet a dependency such that they would realize the *maximum partition sum bound* [20]. This meant we would generate the largest possible set of well-formed intent models from this set of 10 procedures. Based on Wu et al. [64], we claim that this workload

supersedes the basic requirements of the communication domain, and therefore provides a valid analysis of the expected overhead generated by our approach.

The entire process was performed using procedure descriptors (meta-data), which can be loaded independently of the execution units. These descriptors contain information on the classification of procedures, as well as their dependencies.

Setup: Experiments were performed on a 64-bit Fedora Linux based machine with 4.00 GB of memory and a quad core Intel i7 4-core CPU clocked at 1.8 GHz per core. Our system ran the OpenJDK JRE version 1.7.

Process: As detailed in Listing 7.1, we first created a set of 10 procedures and associated DSCs from the communication domain that would result in the desired set of generated intent models:

Listing 7.1: Scenario Artifacts

```
// Set up DSCs
DSC sendDSC = new DSC("Send", Type.OPER);
DSC encryptDSC = new DSC("Encrypt", Type.OPER);
DSC partDSC = new DSC("MultiPart", Type.OPER);
DSC compressDSC = new DSC("Compression", Type.OPER);

// Dependency definitions
ArrayList<DSC> dependencies1 = new ArrayList<DSC>();
dependencies1.add(encryptDSC);
ArrayList<DSC> dependencies2 = new ArrayList<DSC>();
dependencies2.add(partDSC);
ArrayList<DSC> dependencies3 = new ArrayList<DSC>();
dependencies3.add(compressDSC);

// Procedures
Procedure procedure1 = new Procedure("0001"
 , "Send1", sendDSC, dependencies1);
Procedure procedure2 = new Procedure("0002"
 , "Send2", sendDSC, dependencies1);
Procedure procedure3 = new Procedure("0003"
 , "Send3", sendDSC, dependencies1);
Procedure procedure4 = new Procedure("0004"
 , "Encrypt1", encryptDSC, dependencies2);
Procedure procedure5 = new Procedure("0005"
 , "Encrypt2", encryptDSC, dependencies2);
Procedure procedure6 = new Procedure("0006"
 , "Encrypt3", encryptDSC, dependencies2);
Procedure procedure7 = new Procedure("0007"
 , "Part1", partDSC, dependencies3);
```

```
Procedure procedure8 = new Procedure("0008"
 , "Part2", partDSC, dependencies3);
Procedure procedure9 = new Procedure("0009"
 , "Compress1", compressDSC);
Procedure procedure10 = new Procedure("0010"
 , "Compress2", compressDSC);

Repository repo = Repository.getInstance();
repo.addProcedure(procedure1);
[...]
repo.addProcedure(procedure10);
```

In Listing 7.2 we generated a set of dummy procedures to simulate the existence of resources not associated with the current command being executed:

Listing 7.2: Dummy Artifacts

```
DSC testDSC = new DSC("Test", Type.OPER);
for (int i = 0; i < 90; i++){
  Procedure procedure = new Procedure(String.valueOf(i)
        , "TestProcedure", testDSC);
  repo.addProcedure(procedure);
}
```

Finally, in Listing 7.3 we perform the model generation, validation and selection steps detailed in Section 5.3.2:

Listing 7.3: Scenario Execution

```
for (int i = 0; i < cycles; i++){
  // Find all models that match command
  ArrayList<IntentModel> matchingModels =
  (new NaiveGenerator()).generateModels(initialDSC);
  // Find valid models based on user preferences
  ArrayList<IntentModel> validModels = (new NaiveValidator()).
              validateModels(matchingModels, encryptDSC);
  // Find the best model based on cost
  IntentModel bestModel =
  (new NaiveSelector()).getBestModel(validModels);
}
```

Results: Our set of 100 procedures resulted in the generation of 36 intent models ($3 \times 3 \times 2 \times 2$) that are detailed below. We validated our model against a DSC known to be present in all intent models, therefore ensuring that no models were eliminated from the candidate list, and passing all 36 models to the model selection phase.

	Milliseconds			
Cycles	Min	Max	Average	Total
1	114	144	114	114
10	10	105	36	366
100	3	108	8	817
1000	1	111	2	2205
10000	1	112	1	13632
100000	1	118	1	125899

Table 7.1: Model Generation Timing

Finally our prototype selected a model for execution based on procedure count, this selection was done using our naive cost analysis implementation that uses the number of procedures as the cost factor. The evaluation results are presented in Table 7.1.

Discussion: The values in Table 7.1 provide us some guidance on how the execution times involved in generating, validating and selecting intent models in response to commands and events.

Recall we used the procedure descriptors to generate the intent models that can be loaded independently of the execution units. This reduces the overall memory footprint of a procedure to the bare minimum required for the generation of an intent model. Execution units need only be fetched into memory at the time of execution. As the results of our prototype evaluation show, we are able to perform the full generation, validation and selection process in the order of micro seconds. We believe that these results demonstrate the appropriateness of our architecture for many domains, even those that may have a very high responsiveness requirement such as smart electrical grid management [2].

As depicted in Table 7.1 and illustrated in Figure 7.1, we are not only able to generate intent models within the requirements of high-response domains, but we are also able to generate these intent models with increasing efficiency as we increase

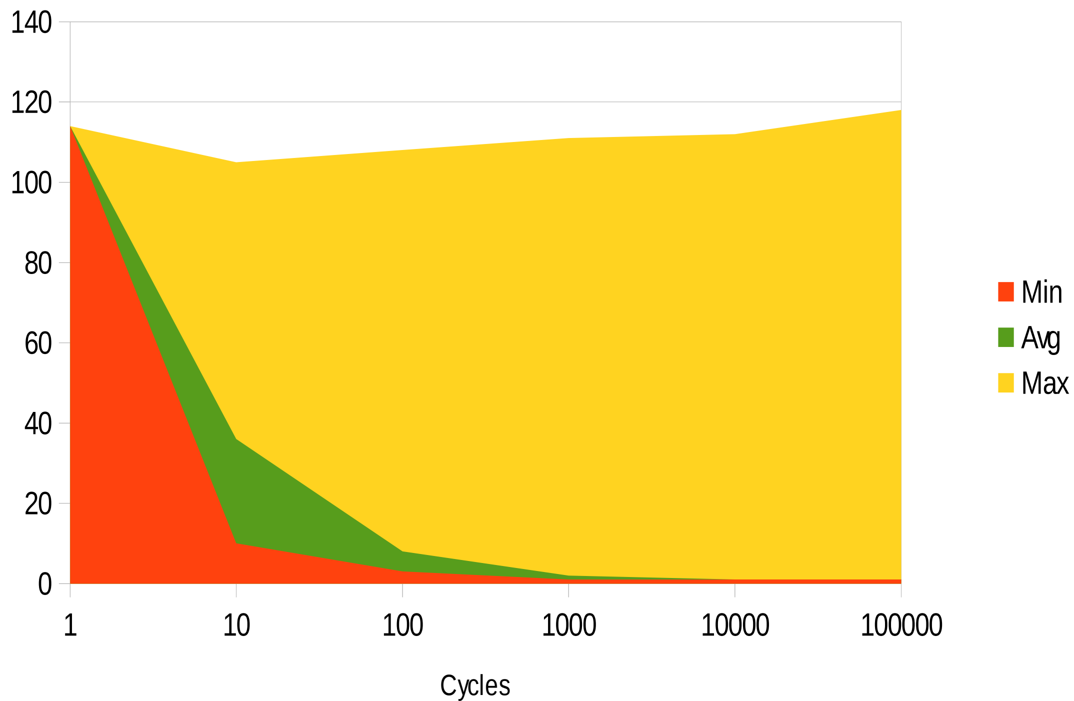

Figure 7.1: Minimum, maximum and average model generation times

cycles. This is due in part to our implementation that can more efficiently manage procedures that are already resident in memory.

Threats to Validity: Although the results in Table 7.1 provide us an estimate of the cost of generating, validating and selecting intent models in a given domain, we acknowledge that the operating times depend heavily on the definition of domain operations and the resulting granularity of procedures and DSCs. The time complexity of model generation is exponential with respect to N, where N is the number of available procedures, and model execution time increases proportionally with the granularity of procedures and execution units. The space complexity of our model generation process is also exponential, which is salient in our analysis as it may have a measurable effect on memory access times for a non-trivial amount of procedures. Additionally, our prototyping environment did not incorporate changing

system or environmental contexts, which could result in reduced operational time, or incur additional memory requirements.

We attempted to normalize the inherent non-determinacy of Java's JIT as presented by Georges et al. [26] through large sets of cycles to determine timing, as well as presenting various statistical measures, however we acknowledge that some aspects of achieving full analysis of the Java Virtual Machine remain unaddressed.

We then went on to evaluate the execution of the real world scenario of transferring a single 10 megabyte file in both an encrypted and non-encrypted form. This operation was performed on the current version of the CVM and our proposed architecture. Our measurements are broken down into 3 areas:

1. The time to generate a model (if needed) including access to the repository to load the necessary metadata
2. The time to load the needed components for adaptation
3. The time to perform encryption (if needed) using an external tool called by our middleware

Experiment 2 Setup: Both systems were executed on Java version 1.6 using Eclipse Helios Service Release 2. They both incorporated the Janino Java compiler for runtime class loading on a Windows 7 Enterprise based machine with 4.00 GB of memory and a dual core Intel CPU clocked at 2.40 GHz per core.

Our experiments were done using an unoptimized approach where components were not preloaded into memory prior to model generation. This was done to (1) present a less than ideal case to demonstrate that our approach still performed reasonably well despite the added overhead, and (2) we found that when optimization strategies were applied, Java did not provide us with the necessary timing resolution to make any conclusive claim about the differences.

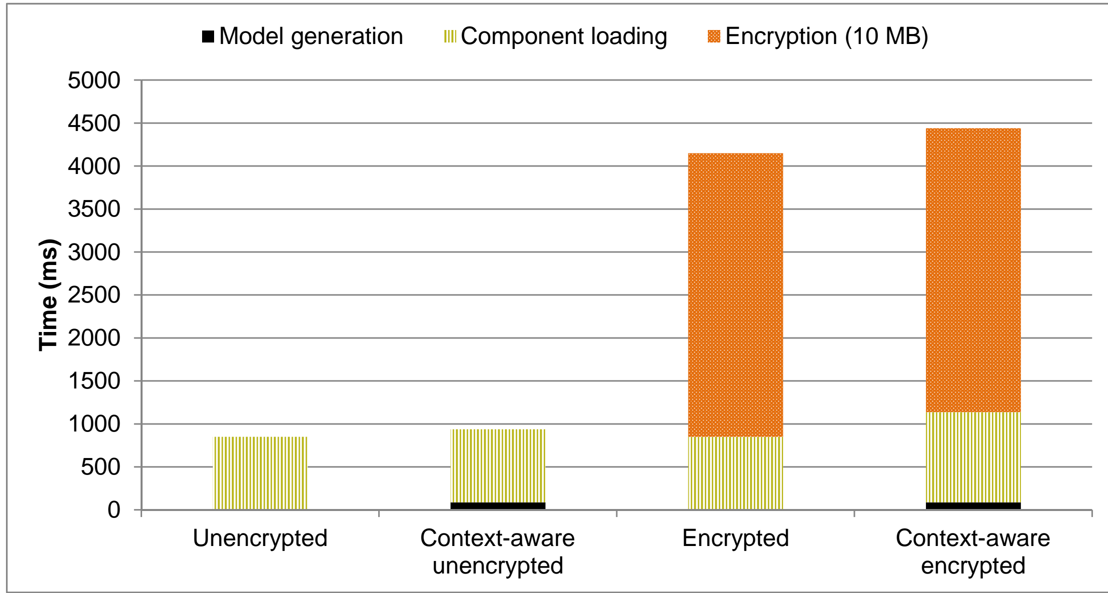

Figure 7.2: Variable File Transfer Operation

For our adaptive system, we curated a set of 10 procedures that would meet the Maximum Product Over Partition bound. This resulted in the generation of 36 models that were able to perform the requested operation. Additionally, the runtime procedure list was populated with the meta data of 90 dummy procedures for a total of 100. This far outnumbers the required procedure set for the average instance of the two domains we have investigated so far, namely the CVM and MGridVM.

Experiment 2 Results: The results of the experiments as shown in Figure 7.2. As both systems incorporate reflection through runtime code compilation and loading, they both experience degraded performance performance compared to direct execution of an operation in a non-adaptable system. However, as the current UCM does not incorporate any context aware adaptation, and because it does not compose its operational facilities from multiple components, it has a slight advantage in arriving

at the cusp of execution compared to our proposed solution. Acknowledging this disparity, we make the following claims:

1. Although slower on arriving at execution plan due to the model generation, our approach can achieve much lower execution times by composing its intent models of more efficient procedures based on system context. For example, to remove the requirement of including an encryption procedure when performing a file transfer if the current connection or network is already secure.

2. Due to the lack of policy awareness with the UCM, the current CVM must expose operational variability at the language level. This will invariably (1) increase the complexity of of designing a communication schema as their are more elements in the language, (2) increase the complexity of synthesizing the CML model provided by to the SE by the UI, and (3) decrease the level of assurance in the execution of an operation as a CML model may incorporate elements that do not comply with current system policies.

3. To reduce the cases of schemas not meeting needed business policies (such as HIPAA [59]), and to mitigate the necessity of modifying those schemas by users that may lack the necessary technical knowledge, schemas designed for the current UCM may choose the most strict operations (such as encrypted file transfers instead of plan text). This design choice, which addresses the issues listed, may result in longer execution times than is required as the current system context may not have required that a file be encrypted prior to transfer.

Our claims are demonstrated in Figure 7.3 where we simulate a set of file transfer operations in the cases where encryption may or may not be required. As you can see, to reduce complexity the current implementation of CVM utilizes encryption for all transfers, which greatly increases the overall operational time for responding to the file transfer request. The context aware approach however, which

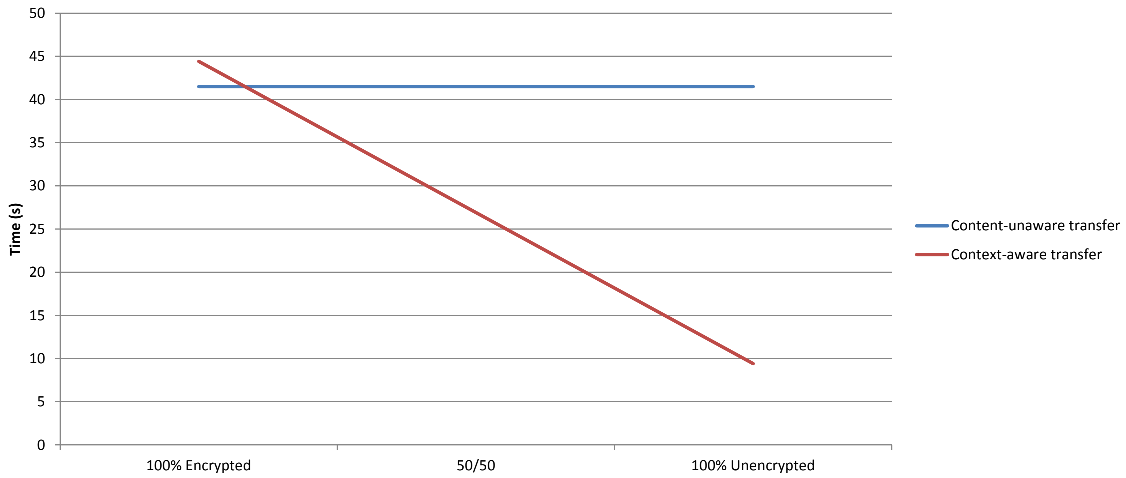

Figure 7.3: Simulated Transfer of 10 files

has the ability to decide on whether or not file encryption is necessary, is able to achieve greatly reduced operational time even when considering the time taken for the model generation process.

7.2 Model Validation

Here we present a demonstrative scenario of our validation process. We will show a set of intent models generated for a specific control script command, the Alloy model representations resulting from our translation process, as well as the translation and application of policies to validate the functionality of the intent models through assertions.

7.2.1 Middleware Artifacts

In this scenario we instantiated the middleware for a DSVM, CVM - Communication Virtual Machine [1], responsible for managing user-centric communication between individuals within an organization. It has available a list of appropriate procedures for the communication domain. Table 7.2 presents the subset of available

[1]http://cml.cs.fiu.edu/cvm/cvm.html

Table 7.2: Subset of Middleware procedures

Name	DSC	Deps
SendBasic	Send	
SendSecure	Send	{Encrypt}
SendCompress	Send	{Compress}
SendSecComp	Send	{Encrypt, Compress}
PKIEncrypt	Encrypt	
GZipCompress	Compress	
...	...	...

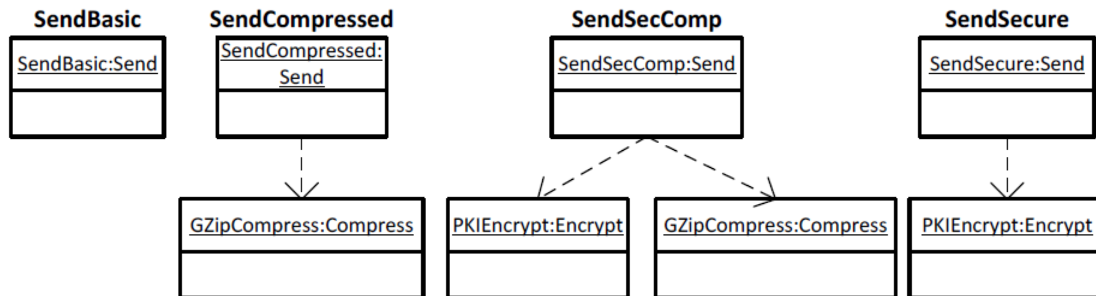

Figure 7.4: Scenario Intent Models: Send command

procedures that factor into our example. A communication has been established between two individuals over an unsecured connection, and the initiator wishes to send a file to the other connected party. The middleware receives a control script that includes the command to perform a file transfer from one party to the other. The model generation process is triggered and a candidate list of intent models is created that are capable of performing the requested action. The models along with their composite procedures and associated type definitions are presented in Figure 7.4.

Governing the operation of the middleware with regards to file transfers over unsecured networks are two policies derived from business rules. The first stipulates that due to the sensitive nature of data housed in the system, all file transfers must be performed securely. This is interpreted as requiring that all files be

encrypted prior to transfer. The second states that all files should be transferred uncompressed due to the burden involved in managing compression when security is required on resource constrained devices. These restrictions are presented as Event Condition Action policies shown in Figure 7.5

7.2.2 Translation

Following the steps described in algorithms 3 and 4, we are able to translate the intent models depicted in Figure 7.4.The resulting Alloy model specifications for the final two intent models in Figure 7.4 are shown in listings 7.4 and 7.5. The two selected intent models and their resulting model specifications depict the main conditions we wish to demonstrate; the validity and invalidity of an intent model based on the presence or absence of a procedure per user and system policies. Based on the policies listed in Figure 7.5, we expect a model to be valid ***iff*** it includes a procedure classified by the *Send* DSC, and excludes any procedure of the type *Compress*.

7.2.3 Validation

When Alloy checks the send_policy assertion of the Alloy specification in Listing 7.4, it results in the generation of a counterexample (Figure 7.6), which is, of course, exactly our described model. This is to be expected as the model clearly violates our second policy (See Figure 7.5), which prohibits file compression in any valid intent model. As our assertion does not hold for the described model specification, the associated intent model *SendSecComp* from Figure 7.4 can be removed from the candidate list of generated intent models.

(1) if (Send) {Send(FileURI) $\rightarrow$ Encrypt(FileURI)}
(2) if (Send) {Send(FileURI) $\rightarrow$ Compress(FileURI)}

Figure 7.5: Middleware Polices using DSCs

Listing 7.4: SendSecComp translated to Alloy

```
/*
An Allow specification for
the SendSecComp intent model.
It cointains 3 abstract
signatures used to represent
DSCs, and 3 signatures that
extend the abstract signatures
and represent procedures.
Signature names are inherited
from the associated procedures
and DSCs in the analogous
intent model
*/

abstract sig SEND {}
one sig SendSecComp extends SEND {
  d1: one ENCRYPT,
  d2: one COMPRESS
}
abstract sig COMPRESS{}
one sig GZipCompress extends COMPRESS{}
abstract sig ENCRYPT {}
one sig PKIEncrypt extends ENCRYPT {}

/*
Assertion for active policies
*/
assert send_policy {
  ∀ p: SEND • ((ENCRYPT in p.d1)
        or (ENCRYPT in p.d2) )
  and ((COMPRESS not in p.d1)
        and (COMPRESS not in p.d2))
}

check send_policy
```

Checking the equivalent policy against the model specification in Listing 7.5 however, results in the absence of counterexamples, which gives us a high degree of certainty that our model is valid based on Alloy's *"complete up to scope"* property [31]. The validity of the model can be trivially observed as a dependency can be seen between the root procedure and the required functionality while the unwanted function is simultaneously excluded. This model therefore remains in our candidate list and moves on to the selection step in out model generation process shown in Figure 4.1.

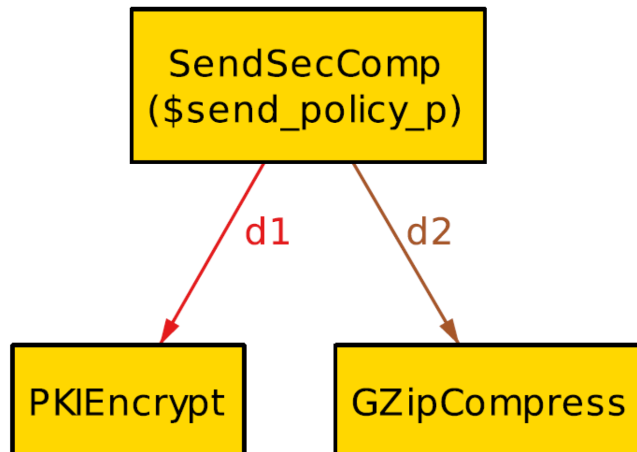

Figure 7.6: Alloy Counterexample

Listing 7.5: SendSecure translated to Alloy

```
/*
An Allow specification for the
SendSecure intent model. It
cointains 2 abstract signatures
used to represent DSCs,
2 signatures that inherit from
abstract signatures, and another
signture that represents an
action from the active policy.
Signature names are inherited
from the associated procedures
and DSCs in the analogous
intent model
*/

abstract sig SEND {}
one sig SendSecure extends SEND {
  d1: one ENCRYPT
}
abstract sig ENCRYPT {}
one sig PKIEncrypt extends ENCRYPT {}
one sig COMPRESS{}

/*
Assertion for active policies
*/
assert send_policy {
  ∀ p: SEND • ((ENCRYPT in p.d1)
          and (COMPRESS not in p.d1))
}

check send_policy
```

7.2.4 Comparative Analysis

The aforementioned approach to functionality validation enhances our ability to constrain intent model execution based on policies. Previously, our abilities to an-

alyze the behavior of an intent model was limited to ensuring that a procedure of a type specified in a relevant policy was present in the intent model (See Policy 1 in Figure 7.5). This limitation not only prevented us from more complex modes of analysis, such as checking for the absence of a procedure (See Policy 2 in Figure 7.5), but we were also unable to verify that the right dependencies were being met, relegating the responsibility to the feature analysis phase of the domain-specific knowledge encapsulation, and subsequent design of procedures and their dependencies. For example, in our previous approach it would be possible for a model M to contain a procedure of type X with a dependency on type Y.

If a policy dictated that for some event with a matching procedure of type E there should be a function of type Y present in the resulting intent model, the intuitive interpretation is that Y should be a dependency of E, however our validation process did not take into account that Y was not a direct dependency, but instead simply verified its presence (Figure 7.7). In this scenario, we are unable to verify if the resulting execution of Y would result in policy compliance. The described behavior required that the procedure writer ensure that Y would only be present in M for an event E ***iff*** Y was a dependent of E. This increased the complexity of domain analysis and procedure development.

7.3 MicrogridVM

We will demonstrate the multi-domain applicability of our approach by outlining a relevant scenario in the smart microgrid domain, and detailing the set of artifacts and processes resulting in the realization of the user's intent.

Scenario

An MGridVM instantiation is responsible for managing the electrical power needs of a small home. The VM has in place a policy that specifies that if the system is

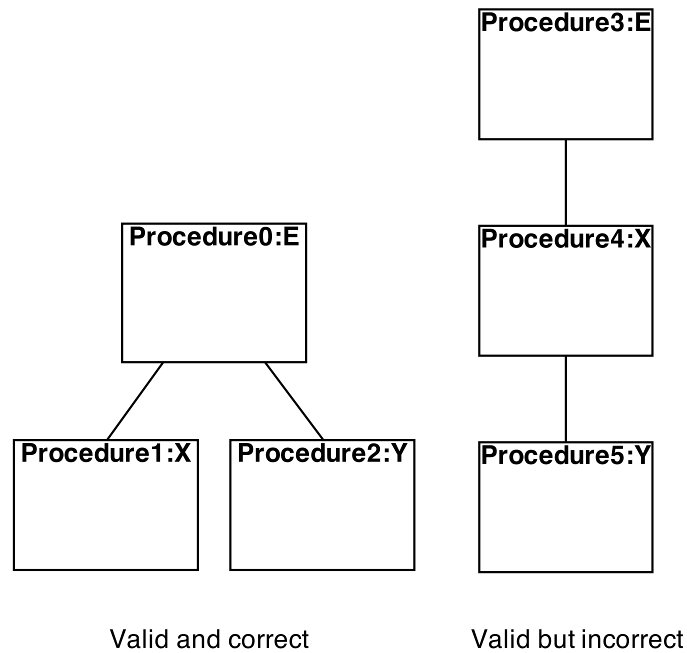

Figure 7.7: Inconsistent Model Validation

in *islanding mode* (that is, it is not connected to the main utility, but instead receives power from a series of local power sources), then before any new load can become operational, the current power consumption must be checked to ensure that there is enough energy to meet the requirements of the new device. If sufficient energy is unavailable, then enough low-priority loads must be shed to allow the new load to become operational. To restate it simply, we want to ensure that we are generating enough power to handle any new load, and if we are not generating enough power, then shed old loads until we have the remaining power is sufficient.

The homeowner attempts to power on a light bulb, listed as a load in the MGridVM UI, by toggling its state from *off* to *on*. This results in a new user model being passed to the synthesis engine, which, after synthesis, results in the generation of a control script containing the command to power on the light bulb.

	Name	Kind
1	Device	attr
2	Controller	attr
3	PropertyName	attr
4	PropertyValue	attr
5	IslandMode	attr
6	AddLoad(Controller, Device)	oper
7	RemoveLoad(Load)	oper
8	AddSource(Controller, Device)	oper
9	RemoveSource(Source)	oper
10	SetDeviceProperty(Device, PropertyName, PropertyValue)	oper
11	ShedLoad	oper
12	CheckWattage	oper

Table 7.3: A subset of DSCs for the microgrid domain.

MGridVM Instance

Currently, the system is in islanding mode, which means it is disconnected from the main utility and is instead being powered by local sources. As a result of this, we have a policy in place determining how to handle new loads becoming operational as seen in Figure 7.8.

Table 7.3 lists a subset of DSCs currently present in our DSVM. DSCs 1 - 9 share an association with the Control Script commands shown in Figure B.1, while the remainder are only related to actions internal to the middleware with no external relationships. Table 7.4 details a subset of procedures available for use in our scenario.

(1) if (IslandMode) {AddLoad() $\rightarrow$ ShedLoad()}

Figure 7.8: MGrid Policy

Name	DSC	Deps
EnableLoad	AddLoad	
DisableLoad	RemoveLoad	
PrepareAndEnableLoad	AddLoad	{ShedLoad}
ReservePower	ShedLoad	
...	...	...

Table 7.4: Subset of MGrid Middleware procedures

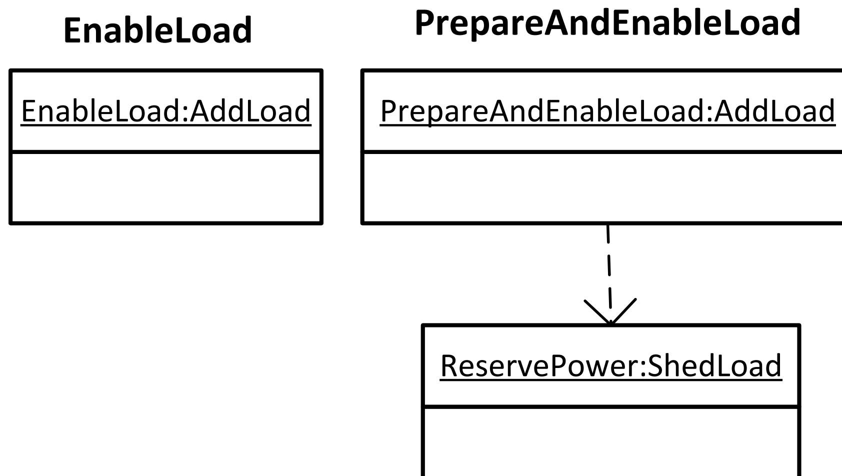

Figure 7.9: Scenario Intent Models: Send command

Model Generation

Upon receipt of the control script containing the AddLoadDeviceCmd command, our middleware enumerates a set of candidate models using Algorithms 1 and 2. This results in the set of candidate models seen in Figure 7.9.

Once generated, our candidate list is then validated to cull all intent models that are not in compliance with the policy in Figure 7.8. As detailed in Chapter 6, the validation of intent models is accomplished through a transformation to their representative Alloy specifications and subsequent analysis by the Allow Analyzer. The Alloy specifications, along with the translated policy, for the EnableLoad and Pre-

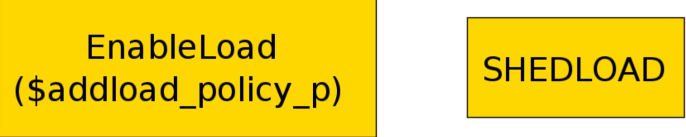

Figure 7.10: Alloy Counterexample

pareAndEnableLoad intent models are shown in Listings D.1 and D.2 respectively. The subsequent analysis of the aforementioned specifications results in the counterexample shown in Figure 7.10, which matched the EnableLoad intent model, as this model does not comply with the relevant policy.

The remaining model, PrepareAndEnableLoad, is then passed to the Model Executor to realize the user intent.

CHAPTER 8

CONCLUSION

This chapter presents a summary of the contributions put forward in our dissertation in attempting to address the issues detailed in Chapter 1. We also introduce directions for future research based on our findings and conclusions.

8.1 Summary of Research

In this dissertation we presented an adaptive middleware design for domain-specific virtual machines (DSVM). This design dynamically integrates decoupled domain-specific knowledge (DSK) that has been captured in a set of artifacts that describe the relevant state and behavior, with a model of execution (MoE) to support the delivery of domain-specific services. The DSK is captured through the use of procedures that perform operations relevant to the domain, and a set of domain specific classifiers (DSCs) that categorize them. DSCs perform this categorization by acting as a labeling system that describes the type of a procedure, as well as by describing first-class state information. Once a middleware instance has been specialized for a particular domain, the MoE provides a platform where procedures are dynamically composed into intent models based on their type classification, and executed based on the context in which a request is made by the layers surrounding the DSVM middleware. An intent model is selected for execution following the full enumeration of all possible intent models based on the types and dependencies of available procedures.

Our architecture performs validation of intent models for execution against Event Condition Action policies at runtime through the use of first-order logic and the Alloy Analyzer. This was accomplished through the development of a methodology for the translation of middleware artifacts into Alloy model specifications and ECA

policies into Alloy assertions. This reduced the process of intent model validation to satisfiability checking in Alloy through the attempted generation of counterexamples. This approach provides robustness in our validation process, which in turn extends the capabilities of our DSVM with respect to the expressiveness of user and system policies, as well as the overall correctness of dependency validation. We demonstrated the efficacy of our approach using case studies of conventional use cases within the communication and microgrid domains. The analysis performed resulted in the generation of counter examples for invalid models, and provable correctness for valid models.

Experiments were performed to determine the overhead required by the proposed design and the results show that the increased execution times are acceptable for the domains under investigation given the functionality of the middleware. Our approach to dynamically composing intent models result in quadratic space and time complexity, which is considered manageable overhead in the model generation process when compared to the operations of the original middleware design. Our approach also provides benefits in overall operation execution time due to its ability to determine at runtime an optimum execution path.

8.2 Future Work

Our work on the DSVM middleware has identified several questions that require further investigation in order to fully realize the true potential of our architectural design. Some of these questions focus on optimization of the MoE, such as, the pre-generation of models representing the procedures and domain-specific classifiers, and a caching mechanism that provides a smarter approach to generating models. Other questions will focus on the selection of the most appropriate model when multiple models apply based on the context of the currently executing command, and dynamic validation of the models to reduce execution times. Additionally, ex-

panding on the current approach to the intent model generation process by investigating methods of potentially amalgamating model generation, validation, and selection to provide a more robust mechanism for evaluating the adequacy of executing a model given current system context. Additionally, we acknowledge the potential applicability of our approach of intent model generation and autonomic execution path selection to other domains that incorporate intent realization.

We plan to expand on this research in various directions. Our main areas of focus are 1) improved efficiency and robustness, 2) behavioral adaptation, and 3) applicability across platforms.

8.2.1 Improved Efficiency and Robustness

The operations of our architecture fall into 4 major categories: generation, validation, selection, and execution. Our major focus thus far has been to realize sufficient generation and execution operations that are able to meet the requirements of the domains under investigation for instantiating a DSVM. We intend to broaden our focus to achieve more efficient and robust model validation and selection. This will include investigating additional formal model checking methods, as well the potential amalgamation of the validation and selection steps. This would require a formalization of a generic cost specification for procedure execution. We hope to use this mechanism to potentially allow a procedure developer to detail the cost of execution regardless of the method or stage at which the cost analysis is performed.

8.2.2 Behavioral Adaptation

Our architecture currently addresses structural runtime adaptation. That is, we determine the semantics of domain-specific operations at runtime by composing disparate executable components, allowing for the steps involved in performing a given task to vary over time based on context. We intend to augment this func-

tionality by facilitating the adaptation of our first-class domain-independent operations - model generation, execution, validation and selection - based on additional context, such as system resources. These adaptations should be transparent to all domain-specific concerns.

Offline adaptation

We intent to investigate the use of offline adaptation to allow a middleware to be instantiated with varying implementations of first-class operations. For instance, we would allow for a light weight model generation process if the middleware is being instantiated on a mobile device whereas a more robust implementation may be provided for a desktop computer.

Runtime adaptation

Additionally, we will investigate runtime behavioral adaptation, allowing first-class operations to be automatically *hot-swapped* as runtime based on environmental context, such as reduced battery or processing power.

8.2.3 Cross platform applicability

Finally, we intend to investigate the applicability of our runtime composition approach to adaptation for current and emerging paradigms, such as mobile and web, as the classification of operations and the use of delegation for performing these operations gain traction.

BIBLIOGRAPHY

[1] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson. Middleware infrastructure for parallel and distributed programming models in heterogeneous systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14(11):1100–1111, 2003.

[2] M. Allison, A. A. Allen, Z. Yang, and P. J. Clarke. A software engineering approach to user-driven control of the microgrid. *Software Engineering and Knowledge Engineering*, 2011.

[3] M. Allison, K. Morris, Z. Yang, P. Clarke, and F. Costa. Towards reliable smart microgrid behavior using runtime model synthesis. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 185–192, 2012.

[4] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425, 1990.

[5] U. Bellur and N. Narendra. Towards a programming model and middleware architecture for self-configuring systems. In *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, pages 1 –6, 0-0 2006.

[6] N. Bencomo. On the use of software models during software execution. In *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, pages 62 –67, may 2009.

[7] P. A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, Feb. 1996. http://doi.acm.org/10.1145/230798.230809.

[8] P. Boettner, M. Gupta, Y. Wu, and A. A. Allen. Towards policy driven self-configuration of user-centric communication. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 35:1–35:6, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1566445.1566493.

[9] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of push-down automata: Application to model-checking. In A. Mazurkiewicz and

J. Winkowski, editors, *CONCUR '97: Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer Berlin Heidelberg, 1997. http://dx.doi.org/10.1007/3-540-63141-0_10.

[10] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[11] M. R. Center". What is middleware?, 2005. http://web.archive.org/web/20120629211518/http://www.middleware.org/whatis.html.

[12] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic anchoring with model transformations. In *Model Driven Architecture–Foundations and Applications*, pages 115–129. Springer, 2005.

[13] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009. http://dx.doi.org/10.1007/978-3-642-02161-9_1.

[14] S. Chiba. Load-time structural reflection in java. In *ECOOP 2000—Object-Oriented Programming*, pages 313–336. Springer, 2000.

[15] S. Cho, H. Kim, S. Cha, and D.-H. Bae. Specification and validation of dynamic systems using temporal logic. *Software, IEE Proceedings -*, 148(4):135–140, Aug 2001.

[16] E. Clarke, O. Grumberg, and D. Peled. *Model Cheking*. Mit Press, 1999. http://books.google.com/books?id=Nmc4wEaLXFEC.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986. http://doi.acm.org/10.1145/5397.5399.

[18] P. J. Clarke, Y. Wu, A. A. Allen, F. Hernandez, M. Allison, and R. France. Towards dynamic semantics for synthesizing domain-specific models. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.*, chapter 9, pages 242 – 269. IGI Global, 2012.

[19] Y. Deng, S. M. Sadjadi, P. J. Clarke, V. Hristidis, R. Rangaswami, and Y. Wang. Cvm – a communication virtual machine. *Journal of Systems and Software*, 81(10):1640 – 1662, 2008. http://www.sciencedirect.com/science/article/pii/S016412120800037X.

[20] T. Došlić. Maximum product over partitions into distinct parts. *Journal of Integer Sequences*, 2005.

[21] F. Eliassen, A. Andersen, G. Blair, F. Costa, G. Coulson, V. Goebel, O. Hansen, T. Kristensen, T. Plagemann, H. Rafaelsen, K. Saikoski, and W. Yu. Next generation middleware: requirements, architecture, and prototypes. In *Distributed Computing Systems, 1999. Proceedings. 7th IEEE Workshop on Future Trends of*, pages 60 –65, 1999.

[22] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[23] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering, 2007. FOSE '07*, pages 37 –54, may 2007.

[24] M. Frappier and A. Mammar. An assertions-based approach to verifying the absence property pattern. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 361–370, Nov 2012.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In M. Broy and E. Denert, editors, *Pioneers and Their Contributions to Software Engineering*, pages 361–388. Springer Berlin Heidelberg, 2001.

[26] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, Oct. 2007.

[27] A. Ghosh, S. wei Li, C. Chiang, R. Chadha, K. Moeltner, S. Ali, Y. Kumar, and R. Bauer. Qos-aware adaptive middleware (qam) for tactical manet applications. In *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, pages 178–183, 2010.

[28] P. Gluck and G. Holzmann. Using spin model checking for flight software verification. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 1, pages 1–105–1–113 vol.1, 2002.

[29] V. H. Hieu and H. D. Hai. An application-aware adaptive middleware architecture for distributed multimedia systems. In *Communications and Electronics, 2006. ICCE '06. First International Conference on*, pages 141–146, 2006.

[30] D. Jackson. *Software abstractions*. MIT press Cambridge, 2006.

[31] D. Jackson. Alloy: a language and tool for relational models. http://alloy.mit.edu/alloy/documentation.html, 2012.

[32] T. Jiang, X. Wang, and Y. Yu. A formal definition of the structural semantics of domain-specific modeling languages. In *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pages 1696–1699, 2010.

[33] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis. Technical Report CMU/SEI-90-TR-21, CMU, Nov 1990.

[34] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, Mar. 2008.

[35] P. Kelsen. A simple static model for understanding the dynamic behavior of programs. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 46–51, 2004.

[36] A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.

[37] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE '07*, pages 259 –268, may 2007.

[38] J. Ma, D. Zhang, G. Xu, and Y. Yang. Model checking based security policy verification and validation. In *Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on*, pages 1–4, May 2010.

[39] G. Madl, S. Abdelwahed, and D. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real-Time Systems*, 33(1-3):77–100, 2006.

[40] P. McKinley, F. Samimi, J. Shapiro, and C. Tang. Service clouds: A distributed infrastructure for constructing autonomic communication services. In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 341–348, 2006.

[41] S. J. Mellor and M. J. Balcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley Professional, 2002.

[42] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 782–796. Springer Berlin Heidelberg, 2008. http://dx.doi.org/10.1007/978-3-540-87875-9_54.

[43] K. Morris, J. Wei, P. Clarke, and F. Costa. Towards adaptable middleware to support service delivery validation in i-dsml execution engines. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 82 –89, Oct. 2012.

[44] S. Nakajima. Verification of web service flows with model-checking techniques. In *Cyber Worlds, 2002. Proceedings. First International Symposium on*, pages 378–385, 2002.

[45] I. A. Niaz and J. Tanaka. An object-oriented approach to generate java code from uml statecharts. *International Journal of Computer & Information Science*, 6(2):315–321, 2005.

[46] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[47] S. Ramanathan, I. Rodriguez, and K. Drira. Adaptive middleware architecture for group communication activities. In *New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on*, pages 1–7, 2011.

[48] V. Raychoudhury, J. Cao, M. Kumar, and D. Zhang. Middleware for pervasive computing: A survey. *Pervasive Mob. Comput.*, 9(2):177–200, Apr. 2013.

[49] R. Romeikat and B. Bauer. Formal specification of domain-specific ECA policy models. In *Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Symposium on*, pages 209 –212, aug. 2011.

[50] S. M. Sadjadi. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Michigan State University, 2003.

[51] R. E. Schantz and D. C. Schmidt. Middleware for distributed systems: Evolving the common structure for network-centric applications. *Encyclopedia of Software Engineering*, 1, 2002.

[52] D. C. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: Their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.

[53] J. Shao, H. Wei, Q. Wang, and H. Mei. A runtime model based monitoring approach for cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 313–320, 2010.

[54] J. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[55] T. Stahl, M. Voelter, J. Bettin, A. Haase, S. Helsen, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, first edition, 2006.

[56] S.-W. Suthon, G. M. Ong, and H. Pung. An adaptive end-to-end qos management with dynamic protocol configurations. In *Networks, 2002. ICON 2002. 10th IEEE International Conference on*, pages 106–111, 2002.

[57] J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, D. Cheung-Foo-Wo, E. Callegari, and M. Riveill. Wcomp middleware for ubiquitous computing: Aspects and composite event-based web services. *Annals of Telecommunications - Annales des Télécommunications*, 64(3-4):197–214, 2009.

[58] A. Tsutsui, H. Maeomiti, R. Kawamura, and K. Yata. An adaptive communication middleware for network service coordination. In *Consumer Communications and Networking Conference, 2004. CCNC 2004. First IEEE*, pages 406–411, 2004.

[59] United States Congress. Health insurance portability and accountability act. U.S. Department of Health & Human Services.

[60] P. Veríssimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser. Cortex: Towards supporting autonomous and cooperating sentient entities. In *Proceedings of European Wireless 2002*, pages 595–601, Florence, Italy, Feb. 2002.

[61] S. Vinoski. An overview of middleware. In A. Llamosí and A. Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 35–51. Springer Berlin Heidelberg, 2004. http://dx.doi.org/10.1007/978-3-540-24841-5_3.

[62] X. Wang, M. Chen, H.-M. Huang, V. Subramonian, C. Lu, and C. Gill. Control-based adaptive middleware for real-time image transmission over bandwidth-constrained networks. *Parallel and Distributed Systems, IEEE Transactions on*, 19(6):779–793, 2008.

[63] Y. Wu, A. Allan, Y. Wang, F. Hernandez, P. J. Clarke, and Y. Deng. A user-centric communication middleware for CVM. *Software Engineering and Applications*, 2008.

[64] Y. Wu, A. A. Allen, F. Hernandez, R. France, and P. J. Clarke. A domain-specific modeling approach to realizing user-centric communication. *Software: Practice and Experience*, 42(3):357–390, 2012. http://dx.doi.org/10.1002/spe.1081.

[65] Y. Wu, F. Hernandez, P. Clarke, and R. France. A DSML for coordinating user-centric communication services. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 93 –102, july 2011.

[66] L. Yan. An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. *Distributed Systems Online, IEEE*, 8(7):1–1, July.

[67] S. Zachariadis, C. Mascolo, and W. Emmerich. The SATIN component system-a metamodel for engineering adaptable mobile systems. *Software Engineering, IEEE Transactions on*, 32(11):910 –927, nov. 2006.

[68] C. Zhang, S. M. Sadjadi, W. Sun, R. Rangaswami, and Y. Deng. User-centric communication middleware. Technical Report FIU-SCIS-2005-11-01, Florida International University, 2005.

APPENDIX A

CVM CONTROL SCRIPTS

1. *controlScript* ::= *command {command}*
2. *command* := *createConnectionCmd | closeConnectionCmd | addParticipantCmd | removeParticipantCmd | sendSchemaCmd | enableMediaInitiatorCmd | enableMediaReceiverCmd | disableMediaInitiatorCmd | disableMediaReceiverCmd | sendMediaCmd | sendFormCmd | declineConnectionCmd | requestFormCmd | requestMediaCmd | sendNegTokenCmd | requestNegTokenCmd*
3. *createConnectionCmd* ::= **createConnection** $connectionID_A$
4. *closeConnectionCmd* ::= **closeConnection** $connectionID_A$
5. *addParticipantCmd* ::= **addParticipant** $connectionID_A$ $personID_A$ {$personID_A$}
6. *removeParticipantCmd* ::= **removeParticipant** $connectionID_A$ $personID_A$ {$personID_A$ }
7. *sendSchemaCmd* ::= **sendSchema** $connectionID_A$ sender-$personID_A$ receiver-$personID_A$ {receiver-$personID_A$} $schema_A$
8. *enableMediaInitiatorCmd* ::= **enableInitiatorMedia** $connectionID_A$ $mediaName_A$
9. *enableMediaReceiverCmd* ::= **enableReceiverMedia** $connectionID_A$ $mediaName_A$
10. dis*ableMediaInitiatorCmd* ::= **disableInitiatorMedia** $connectionID_A$ $mediaName_A$
11. dis*ableMediaReceiverCmd* ::= **disableReceiverMedia** $connectionID_A$ $mediaName_A$
12. *sendMediaCmd* ::= **sendMedia** $connectionID_A$ $mediaName_A$ $mediumURL_A$
13. *sendFormCmd* ::= **sendForm** $connectionID_A$ $formID_A$ $mediumURL_A$ {$mediumURL_A$ } $action_A$
14. *declineConnectionCmd* ::= **declineConnection** sender-$personID_A$ receiver-$personID_A$ {receiver-$personID_A$}
15. *requestFormCmd* ::= **requestForm** $connectionID_A$ $formID_A$ $mediumURL_A$ {$mediumURL_A$ } $requestAction_A$
16. *requestMediaCmd* ::= **requestMedia** $connectionID_A$ $mediaName_A$ $requestAction_A$
17. *sendNegTokenCmd* ::= **sendNegToken** $personID_A$
18. *requestNegTokenCmd* ::= **requestNegToken** $connectionID_A$

Figure A.1: CVM Control Scripts

APPENDIX B

MGRID CONTROL SCRIPTS

1. *controlScript* := *command {command}*
2. *command* := *initializeMGridCmd | addGroupControllerCmd | removeControllerGroupCmd | addLoadControllerCmd | addStorageControllerCmd | addSourceControllerCmd | addPCCCmd | removeControllerCmd | addLoadDeviceTypeCmd | addStorageDeviceTypeCmd | addSourceTypeCmd | addMeterTypeCmd | removeTypeCmd | addLoadDeviceCmd | addStorageDeviceCmd | addSourceCmd | addSmartMeterCmd | addLegacyMeterCmd | removeEntityCmd | setPropertyCmd | requestPropertyCmd*
3. *initializeMGridCmd* := **initializeMGrid** mgridID$_A$
4. *addGroupControllerCmd* := **addGroupController** contGroupID$_A$ controllerID$_A$ {controllerID$_A$}
5. *removeGroupControllerCmd* := **removeGroupController** contGroupID$_A$
6. *addLoadControllerCmd* := **addLoadController** controllerID$_A$ name$_A$ cardinality$_A$ critical$_A$ groupAction$_A$ lowerWattage$_A$ upperWattage$_A$ {typeID$_A$ }
7. *addStorageControllerCmd* := **addStorageController** controllerID$_A$ name$_A$ cardinality$_A$ chargeStatus$_A$ {typeID$_A$ }
8. *addSourceControllerCmd* := **addSourceController** controllerID$_A$ name$_A$ cardinality$_A$ critical$_A$ groupAction$_A$ {typeID$_A$ }
9. *addPCCControllerCmd* := **addPCCController** controllerID$_A$ name$_A$ cardinality$_A$ critical$_A$ connected$_A$ typeID$_A$
10. *removeControllerCmd* := **removeController** controllerID$_A$
11. *addLoadDeviceTypeCmd* := **addLoadDeviceType** deviceTypeID$_A$ typename$_A$ critical$_A$ usage$_A$ controllerID$_A$
12. *addStorageDeviceTypeCmd* := **addStorageDeviceType** deviceTypeID$_A$ typename$_A$ lowerThres$_A$ upperThres$_A$ controllerID$_A$
13. *addSourceTypeCmd* := **addSourceType** sourceTypeID$_A$ typename$_A$ sourceC$_A$ priority$_A$ controllerID$_A$
14. *addMeterTypeCmd* := **addMeterType** meterTypeID$_A$ typename$_A$ controllerID$_A$
15. *removeTypeCmd* := **removeType** typeID$_A$
16. *addLoadDeviceCmd* := **addLoadDevice** deviceID$_A$ deviceTypeID$_A$ wattage$_A$ control$_A$ critical$_A$ { (attribute$_A$, value$_A$)}
17. *addStorageDeviceCmd* := **addStorageDevice** deviceID$_A$ deviceTypeID$_A$ wattage$_A$ capacity$_A$ charging$_A$ chargeT$_A$ {(attribute$_A$, value$_A$)}
18. *addSourceCmd* := **addSource** sourceID$_A$ sourceTypeID$_A$ wattage$_A$ onDemand$_A$ charging$_A$ chargeT$_A$ {(attribute$_A$, value$_A$)}
19. *addSmartMeterCmd* := **addSmartMeter** meterID$_A$ meterTypeID$_A$ tarriff$_A$ usage$_A$
20. *addLegacyMeterCmd* := **addLegacyMeter** meterID$_A$ meterTypeID$_A$
21. *removeEntityCmd* := **removeDevice** entityID$_A$
22. setLCPropertyCmd := **setLCProperty** deviceID$_A$ attribute$_A$ value$_A$
23. *setDevicePropertyCmd* := **setDeviceProperty** deviceID$_A$ attribute$_A$ value$_A$
24. *requestPropertyCmd* := **requestProperty** deviceID$_A$ attribute$_A$

Figure B.1: MGridVM Control Scripts

APPENDIX C

NETWORK COMMUNICATION BROKER API

<table>
<tr><td><code>addParty(java.lang.String sessionID, java.lang.String participantID)</code>
This function adds the participants specified to the specific session</td></tr>
<tr><td><code>createSession(java.lang.String sessionID)</code>
This function creates a session with the specific session ID</td></tr>
<tr><td><code>createUserProfile(UserObject usr, java.lang.Object schema)</code>
This method generates a user profile for the given user</td></tr>
<tr><td><code>disableMedium(java.lang.String connectionID, java.lang.String mediumName)</code>
This command will stop sending the specified medium to all the participants during the connection</td></tr>
<tr><td><code>enableMedium(java.lang.String connectionID, java.lang.String mediumName)</code>
Enables the media steam</td></tr>
<tr><td><code>isCreatedSession(java.lang.String sessionID)</code>
This method returns whether the session was created or not</td></tr>
<tr><td><code>login(java.lang.String userName, java.lang.String password)</code>
This method will attempt login the given user</td></tr>
<tr><td><code>logout(java.lang.String userName)</code>
Logs the user out</td></tr>
<tr><td><code>mapConnToSession(java.lang.String connectionID, java.lang.String sessionID)</code>
This function maps a connection to a session</td></tr>
<tr><td><code>removeParty(java.lang.String sID, java.lang.String participant)</code>
This method adds the list of participants to the given session</td></tr>
<tr><td><code>resetNCB()</code>
Resets the ncb instance</td></tr>
<tr><td><code>retrieveSchemas(java.lang.String userName, java.lang.String password)</code>
This method returns the schemas for the given user</td></tr>
<tr><td><code>saveSchema(java.lang.Object schema)</code>
Saves the given schema</td></tr>
<tr><td><code>sendMedia(java.lang.String sID, java.lang.String medium, java.lang.String mediumURL)</code>
This command will send the specified medium to all the participants during the connection</td></tr>
<tr><td><code>sendSchema(java.lang.String sID, java.lang.String senderID, java.lang.String listReceiver, java.lang.Object control_xcml)</code>
This method will send the schema to all participants in the specified connection</td></tr>
<tr><td><code>sendSchema(java.lang.String sID, java.lang.String senderID, java.lang.String listReceiver, java.lang.String control_xcml, java.lang.String data_xcml)</code>
This method sends a schema to a given user in a given session</td></tr>
</table>

Table C.1: Network Communication Broker API

APPENDIX D

ALLOY SPECIFICATIONS

Listing D.1: EnableLoad Alloy Specification

```
/*
An Allow specification for the
PrepareAndEnableLoad intent model. It
cointains 1 abstract signature1
used to represent DSCs,
and 1 signature that inherits from
the abstract signature.
Signature names are inherited
from the associated procedures
and DSCs in the analogous
intent model
*/

abstract sig ADDLOAD {}

one sig EnableLoad extends ADDLOAD {}

abstract sig SHEDLOAD {}

/*
Assertion for active policies
*/
assert addload_policy {
  all p: ADDLOAD | (SHEDLOAD in p.d1)
}

check addload_policy
```

Listing D.2: PrepareAndEnableLoad Alloy Specification

```
/*
An Allow specification for the
PrepareAndEnableLoad intent model. It
cointains 2 abstract signatures
used to represent DSCs,
and 2 signatures that inherit from
abstract signatures.
Signature names are inherited
from the associated procedures
and DSCs in the analogous
intent model
*/

abstract sig ADDLOAD {}

one sig PrepareAndEnableLoad extends ADDLOAD {
  d1: one SHEDLOAD
}

abstract sig SHEDLOAD {}

one sig ReservePower extends SHEDLOAD {}

/*
Assertion for active policies
*/
assert addload_policy {
  all p: ADDLOAD | (SHEDLOAD in p.d1)
}

check addload_policy
```

VITA

KARL MORRIS

July 2012 - Present	Co-founder & CEO Colada Studios LLC
April 2012 - Present	Member Upsilon Pi Epsilon
Fall 2009 - Summer 2014	Ph.D., Computer Science School of Computer and Information Systems Florida International University
Spring 2011	Instructor CGS 4854, Website Construction and Management
Fall 2010	Government Assistantship in Areas of National Need (GAANN)
March 2006 - July 2009	Project Officer - Web & Standards Central Information Technology Office Kingston, Jamaica
June 2001 - June 2006	Database Administrator Office of the Contractor-General Kingston, Jamaica
Jan 2001 - May 2001	Programmer Pioneer Software Development Kingston, Jamaica

PUBLICATIONS AND PRESENTATIONS

Morris, K.A., He, X., Costa, F., Clarke, P.J. *An Approach to Dynamic Validation of Intent Model Behavior in DSVMs* (Under review CASCON 2014)

Morris, K.A., Allison, M., Wei, J., Costa, F., Clarke, P *Towards a Middleware for Domain-Specific Virtual Machines* Submission to the Journal of Information and Software Technology – Elsevier . Impact Factor: 1.522 (Under review)

Allison, M., Morris, K.A., Costa, F., Clarke, P. S*Synthesizing Interpreted Domain-Specific Models to Manage Smart Microgrids* The Journal of Systems and Software – Elsevier. Impact Factor: 1.135

Morris, K.A., Costa, F.M. , Wei, J., & Clarke, P.J. 2012. *Towards Adaptable Middleware to Support Service Delivery Validation in i-DSML Execution Engines* IEEE 14th International Symposium on High Assurance Software Engineering (HASE) Nebraska, USA

Allison, M., Morris, K.A., Yang, Z., Clarke, P.J. & Costa, F.M. 2012. *Managing Smart Microgrid Behavior by Synthesizing Domain-Specific Models.* IEEE 14th International Symposium on High Assurance Software Engineering (HASE) Nebraska, USA