3-19-2014

A Generic Model of Execution for Synthesizing Domain-Specific Models

Mark Allison
Florida International University, malli002@cis.fiu.edu

DOI: 10.25148/etd.FI14071187

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A GENERIC MODEL OF EXECUTION FOR SYNTHESIZING DOMAIN-SPECIFIC MODELS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Mark Allison

2014

To: Dean Amir Mirmiran.
College of Engineering and Computing

This dissertation, written by Mark Allison, and entitled A Generic Model of Execution for Synthesizing Domain-Specific Models, 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.

Xudong He

Deng Pan

Jinpeng Wei

Armando Barreto

Peter J. Clarke, Major Professor

Date of Defense: March 21, 2014

The dissertation of Mark Allison is approved.

Dean Amir Mirmiran.
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2014

DEDICATION

To my family and loved ones who have sacrificed through this journey.

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to all those who contributed to the success of my PhD study and the completion of this thesis. Special thanks to the members of my committee, Drs. Xudong He, Deng Pan, Jinpeng Wei and Armando Barretto for their insights and rigorous questioning that has resulted in the focus that resulted in this work. I especially would like to express my foremost gratitude to Dr. Peter J. Clarke who guided me through this process, going above and beyond in due diligence and patience.

ABSTRACT OF THE DISSERTATION

A GENERIC MODEL OF EXECUTION FOR SYNTHESIZING DOMAIN-SPECIFIC MODELS

by

Mark Allison

Florida International University, 2014

Miami, Florida

Professor Peter J. Clarke, Major Professor

Software engineering researchers are challenged to provide increasingly more powerful levels of abstractions to address the rising complexity inherent in software solutions. One new development paradigm that places models as abstraction at the forefront of the development process is Model-Driven Software Development (MDSD). MDSD considers models as first class artifacts, extending the capability for engineers to use concepts from the problem domain of discourse to specify apropos solutions. A key component in MDSD is domain-specific modeling languages (DSMLs) which are languages with focused expressiveness, targeting a specific taxonomy of problems. The de facto approach used is to first transform DSML models to an intermediate artifact in a HLL e.g., Java or C++, then execute that resulting code.

Our research group has developed a class of DSMLs, referred to as interpreted DSMLs (i-DSMLs), where models are directly interpreted by a specialized execution engine with semantics based on model changes at runtime. This execution engine uses a layered architecture and is referred to as a domain-specific virtual machine (DSVM). As the domain-specific model being executed descends the layers of the DSVM the semantic gap between the user-defined model and the services being provided by the underlying infrastructure is closed. The focus of this research is the synthesis engine, the layer in the DSVM which transforms i-DSML models into executable scripts for the next lower layer to process.

The appeal of an i-DSML is constrained as it possesses unique semantics contained within the DSVM. Existing DSVMs for i-DSMLs exhibit tight coupling between the implicit model of execution and the semantics of the domain, making it difficult to develop DSVMs for new i-DSMLs without a significant investment in resources.

At the onset of this research only one i-DSML had been created for the user-centric communication domain using the aforementioned approach. This i-DSML is the Communication Modeling Language (CML) and its DSVM is the Communication Virtual machine (CVM). A major problem with the CVM's synthesis engine is that the domain-specific knowledge (DSK) and the model of execution (MoE) are tightly interwoven consequently subsequent DSVMs would need to be developed from inception with no reuse of expertise.

This dissertation investigates how to decouple the DSK from the MoE and subsequently producing a generic model of execution (GMoE) from the remaining application logic. This GMoE can be reused to instantiate synthesis engines for DSVMs in other domains. The generalized approach to developing the model synthesis component of i-DSML interpreters utilizes a reusable framework loosely coupled to DSK as swappable framework extensions.

This approach involves first creating an i-DSML and its DSVM for a second domain, demand-side smartgrid, or microgrid energy management, and designing the synthesis engine so that the DSK and MoE are easily decoupled. To validate the utility of the approach, the SEs are instantiated using the GMoE and DSKs of the two aforementioned domains and an empirical study to support our claim of reduced developmental effort is performed.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

LIST OF ACRONYMS

AC	Alternating Current
DC	Direct Current
CERTS	Consortium for Electric Reliability Technology Solutions
CHP	Combined Heat and Power
CM	CERTS Microgrid
CVM	Communication Virtual Machine
DER	Distributed Energy Resources
DG	Distributed Generation
DS	Distributed Storage
DSML	Domain-Specific Modeling Language
DOE	U.S. Department of Energy
MGridML	Microgrid Modelling Language
MGridVM	Microgrid Virtual Machine
MUI	Microgrid User Interface
MCM	Microgrid Control Middleware
MHB	Microgrid Hardware Brokerage
MDE	Model-Driven Engineering
MSE	Microgrid Synthesis Engine

CHAPTER 1

INTRODUCTION

Increased complexity and pervasiveness of software has resulted in the need for new approaches to develop software applications specific to a given domain. Although there will always be the need to develop software using the conventional software development life-cycle [9, 60], using a development approach that focuses on specific domains and places models at the center of the development process can improve productivity [30, 29]. Model-driven software development and the use of domain-specific modeling languages (DSMLs) are gaining more attention as the tools and techniques to support the development of such applications are becoming more reliable [44, 69].

There are several advantages of using DSMLs when creating domain-specific applications, including: (1) the developer is presented with an abstraction using the concepts from the problem domain; (2) the concrete syntax of the DSML can use graphical symbols and text from the problem domain; and (3) modeling tools continue to improve their code generation capabilities.

Using DSMLs to develop applications involve creating a platform-independent model (PIM) of the application which is resultantly transformed to a platform-specific model (PSM). The PSM is usually in the form of a high-level language artifact which is compiled, linked then executed. This approach is very similar to that used for text-based domain specific languages (DSLs) [30, 55]. Converting DSL models to code in a high-level language may involve a series of model-to-model, model-to-text and text-to-text transformations that is difficult to adapt to changes at runtime. One alternative approach to transforming models into high-level language prior to execution is interpreting the models directly using a specialized execution engine for the specific domain. We refer to DSMLs that support the direct execution of models as *interpreted DSMLs* or *i-DSMLs* [56, chp. 9]. The dynamic semantics of i-DSML models are based on changes to models at runtime (the currently executing model and the new model) and the current state of the running system.

The interpretation of i-DSML models is the obligation of an execution engine, referred to as a *domain-specific virtual machine* (DSVM), which is coupled to the semantics of the domain. The DSVM is designed using a layered architecture which supports the dynamic semantics by separating concerns as regards model refinement. As the domain-specific model being executed descends the layers of the DSVM the semantic gap between the PIM and PSM closes. In this research we focus on the layer of the DSVM that transforms models into executable scripts for the next layer to execute, this layer is known as the *synthesis engine*.

The current methodology entails first defining the i-DSML in terms of its meta-model (abstract syntax and static semantics) and subsequently constructing the DSVM. The serialized approach necessitates detailing the language's execution semantics only after creating the metamodel. The execution semantics of the language, the presence of model elements within the model, their interrelationship, context within successive models and the environment need to be unambiguously defined to limit or remove undesired or non-deterministic behaviors. At the onset only one i-DSML had been created for use within the user-centric communication domain applying the aforementioned approach. This i-DSML is the Communication Modeling Language (CML) [83] and its DSVM is the Communication Virtual Machine (CVM) [17]. The execution semantics for CML remains tightly woven into CVM locking the methodology in a one language, one interpreter mapping. While the semantic domain is an extension of the abstract syntax, current approaches do not allow for the execution semantics to be defined in parallel to the meta-model or in a reusable manner [15].

In the development of this dissertation a second i-DSML, MGridML, and it's DSVM, MGridVM, was created in its entirety. This i-DSML pertains to the energy management domain, specifically addressing an atomic element within the *smart grid* concept called the *microgrid*. During the development of this new i-DSML, several commonalities became apparent between the initial language/interpreter pair, CML/CVM, and the new MGridML/MGridVM.

To further investigate the commonalities within the DSVM we scrutinized the way each i-DSML is executed to establish a *model of execution* (MoE) for each the i-DSML. A MoE implicitly describes how models (or model differences) translate to behavior. This MoE, while unique to each language, possesses mutual execution constituent elements. We sought to understand these elements with an eye to reuse them in building future i-DSMLs.

The focus of this dissertation is to investigate the feasibility and utility of an extensible framework based on a generic model of execution (GMoE) for i-DSMLs. The shortcomings of the existing DSVM development methodologies which lacks support for reuse is the primary motivation for embarking on this research path and is elaborated next.

1.1 Motivation

As software solutions increase in complexity, greater demands are placed on the research community to develop effective tooling support and high level abstract concepts to represent the desired solution. The use of domain-specific approaches has proven its benefits in taming this complexity by allowing the problem to be addressed at a higher level of abstraction. There are however some drawbacks associated with this approach which is manifested in the software engineering community being reluctant to fully embrace the paradigm. Of particular interest to the dissertation is the quality of the language solution and the expertise required within the realm of domain-specific modeling.

One central tenet of Model-Driven Engineering (MDE) is computational completeness of PIMs; the capability of such models to be executed [68]. A major challenge in realizing computational completeness lies in the representation and interpretation of the i-DSML execution semantics. Currently i-DSML execution engines are developed from scratch in a serialized manner after the language's syntax has been specified. The execution semantics is usually embedded in the interpreter with no exploitation of the commonalities which exists in the DSVM framework. DSVMs are also designed separately from their respective i-DSML, with a lack of regard for the critical mapping between syntactic and semantic domains.

The development of a i-DSML requires expertise in not only the problem domain but in metamodeling and interpreter building. The development becomes disproportionately resource intensive, as much of the knowledge gained in actual language development is lost and the solutions remain custom-built. As the execution semantics in the language is tightly interwoven within the interpreter there are high maintenance costs involved when minor changes are made to the language [87]. The lack of methodologies that employ the reuse of interpreter logic and model transformation operations is a primary source for development errors [42].

Currently systematic approaches to assist language designers with interpreter tooling and to define DSML execution semantics are a major challenge of MDE [15]. We posit that the development of a GMoE will reduce redundant developmental effort by reusing interpreter logic. In accord with Fayad et.al. [26], a GMoE reified to an object oriented framework addresses the challenge of homogeneity in architecture and may yield the following benefits:

- Containment of verification and validation efforts.

- Improvement in quality through the reuse of proven designs.

- Preservation of domain knowledge (domain in this context refers to that of i-DSML interpretation).
- Encapsulation of volatile implementation details.

The adaptable runtime model which is an abstract representation of the system under control is a staple of the DSVM methodology as such, encapsulating the transformation process is critical to curtailing a lengthy and arduous developmental process. By first extracting a MoE, we realized the essential skeleton to actualize a framework capable of rapidly realizing interpreters across multiple domains. Defining the language in terms of its metamodel and execution semantics in rough parallel allows language authors to approach the task in a totalistic manner. This also allows for interdependency and traceability concerns to be more apparent and manageable.

The preliminary work on CVM and creating an i-DSML for microgrid energy management, *MGridML* and *MGridVM*, indicated that the generic architecture for DSVMs presented by Clarke et al. [56, chp. 9], shown in Figure 1.1, was feasible and interesting. We directed our investigation into the commonality found within the semantics for the synthesis process of the two domains. Our primary observation revealed that the redundant parts of these MoEs were, (1) a comparison of models at runtime and, (2) a subsequent interpretation of the resultant changes to produce directives to lower layers of the DSVM; however deeper analysis was warranted.

1.2 Problem Definition and Contributions

The research question under investigation is *how to decouple the domain-specific knowledge (DSK) from the model of execution (MoE) in the synthesis engine, and subsequently producing a generic model of execution (GMoE) that can be used to instantiate synthesis engines in other DSVMs.* The scope of this research is confined within the synthesis engine which carries core processes to interpret changes in the

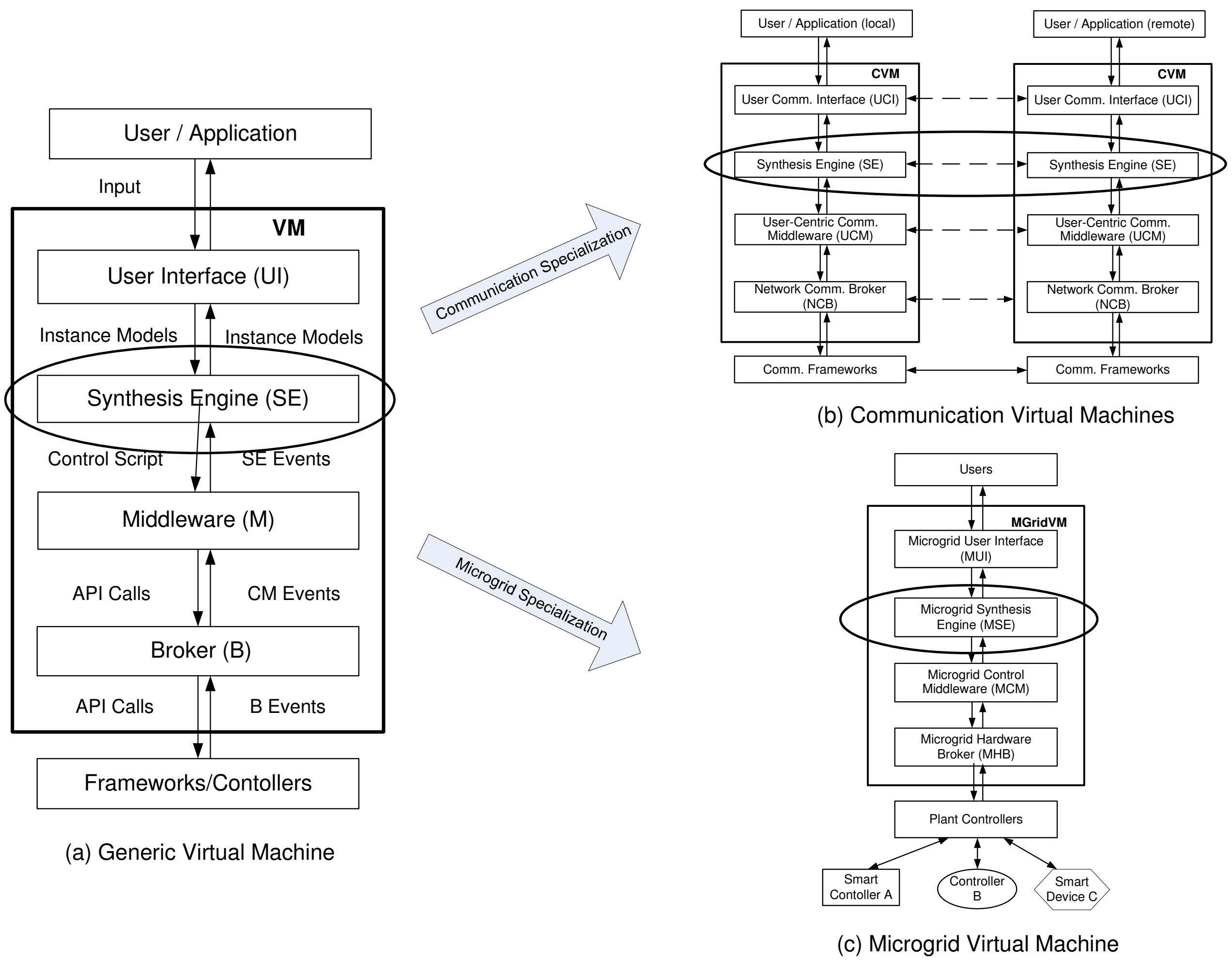

User / Application
Input
VM
User Interface (UI)
Instance Models
Instance Models
Synthesis Engine (SE)
Control Script
SE Events
Middleware (M)
API Calls
CM Events
Broker (B)
API Calls
B Events
Frameworks/Contollers
(a) Generic Virtual Machine
Communication Specialization
Microgrid Specialization
User / Application (local)
User / Application (remote)
CVM
User Comm. Interface (UCI)
Synthesis Engine (SE)
User-Centric Comm. Middleware (UCM)
Network Comm. Broker (NCB)
Comm. Frameworks
(b) Communication Virtual Machines
Users
MGridVM
Microgrid User Interface (MUI)
Microgrid Synthesis Engine (MSE)
Microgrid Control Middleware (MCM)
Microgrid Hardware Broker (MHB)
Plant Controllers
Smart Contoller A
Controller B
Smart Device C
(c) Microgrid Virtual Machine

Figure 1.1: High Level View of Generic Architecture Highlighting the Synthesis Engines

controlled system and user requirements to produce control scripts which in turn dictates behavior of the aforementioned controlled system. To answer this question, the research effort required a division into the following three sub-problems:

1. *How do we formulate a MoE for MGridML model synthesis in MGridVM, based on changes to user-defined models at runtime?* This sub-problem requires an investigation into the various approaches used to represent dynamic semantics and how model changes can be realized in an efficient and correct manner for a new domain.

2. *How can DSK semantics be extracted from the MoE and represented in a persistent manner?* To answer this sub-problem we need to decouple the DSK semantics from the MoE for a given DSVM. In addition, we need to identify techniques that can be used to represent the DSK semantics in a persistent state.

3. *How to formulate a GMoE from the reusable interpreter logic and instantiate the synthesis engine, given a representation of the DSK?* For this sub-problem we need to transform the MoE to a GMoE so that can be used for different domains e.g., user-centric communication, and define a methodology for recombining the DSK semantics and GMoE. This approach needs to be evaluated using at least two domains for different types of systems.

In addressing these questions, this research presents its primary contribution as an integrated methodology and tool support for reusing interpreter logic in the form of a generic model of execution for i-DSML synthesis engines. In particular, this high-level contribution has spawned the following as novel elements:

Contribution 1: The formulation of a detailed description of a new i-DSML for the microgrid domain complete with an unambiguous definition of the semantics for synthesizing instances of its models based on changes at runtime. The design intent of the MoE for this i-DSML is formulated around the lose coupling between the commonalities of DSVMs interpreter logic and DSK concerns.

To evaluate this contribution we described an abstraction of the synthesis process for MGridVM that is unambiguous and traceable with respect to the requirements established by the microgrid domain analysis process. The specification was sufficiently complete to be reified to a prototype capable of generating the correct microgrid control scripts (MCSs) given a test suite from a cross section of models from the microgrid domain.

Contribution 2: A methodology to separate and persistently represent Domain-Specific Knowledge (DSK) using the synthesis engine for the MGridVM. This entails revisiting the initial MoE for the MGridVM synthesis engine and decouple the concerns utilizing aspect-oriented refactoring in such a manner that the DSK can be made persistent. More specifically, we persistently represent DSK as: (1) the i-DSML meta-model; (2) a set of finite state automatas; and (3) a change mapping table whereby changes between the user intent model and the adaptive runtime model is mapped to specific domain actions.

Contribution 3: A representation of a GMoE for i-DSMLs. The GMoE is evaluated via synthesis engine instantiations given DSK artifacts and the derived GMoE as proof of concept. We present a prototype and use appropriate metrics to measure the change in the code base, effort, coupling and performance, compared to earlier instantiated synthesis engines to determine utility and the overhead required during the development process.

1.3 Dissertation Roadmap

Chapter 2 situates the dissertation by providing background in model-driven engineering, domain-specific modeling languages, aspect-oriented software engineering, and energy management as it relates to the microgrid. In addition, we provide a review of the related literature in the areas of model operations, model synthesis, model execution, an existing i-DSML, the Communications Modeling Language (CML) and its interpreter, the Communications Virtual Machine(CVM), and alternate approaches to energy management.

Chapter 3 describes the development of a new i-DSML, MGridML and its interpreter MGridVM. From this new i-DSML we will derive a MoE for model synthesis. A MGridVM prototype is presented to prove the concept of the MoE targeting an architectural design with loose coupling of concerns.

Chapter 4 states how the domain specific knowledge is separated and persistently represented while preserving the intended behavior. In this chapter we apply aspect aware techniques to inspect the MoE derived in chapter 3 for crosscutting concerns, refactor the MoE to separate and persist the DSK as a primary concern.

Chapter 5 presents the distillation of the MoE to a GMoE. We present a meta-modeling approach as a gluing mechanism for instantiation. The reification of the GMoE and the domain specific knowledge is presented as a prototype and a demonstration of principle. The prototype is evaluated and comparatively studied as a tool to leverage common functionality in the reduction of i-DSML development effort.

Chapter 6 summarizes the dissertation's contributions and considers promising future directions for this research.

CHAPTER 2

LITERATURE REVIEW

This chapter provides the background essential to understanding the problem under investigation and reviews prior works related to model operations, model synthesis, model execution and microgrid energy management. An overview of key terms and concepts used in this dissertation is also provided to supplement this review.

2.1 BACKGROUND

This section provides background related to understanding the investigated problem. We first overview the model-driven engineering paradigm then define domain-specific modeling languages (DSMLs) and introduce the concept of an interpreted DSML (i-DSML). We will next look at aspect-oriented software development which is the driving methodology to extricate the DSK from the MoE. Finally key concepts in the energy management domain is addressed to transition to chapter 3.

2.1.1 Model-Driven Engineering

Software research has consistently sought to address increasing complexity in the solution domain using ever higher levels of abstractions. The methodologies and toolsets employed mostly addresses abstractions of the solution space. Model-Driven Engineering (MDE) seeks to bridge the conceptual gap between the problem and solution domains by utilizing models as problem-level abstractions [31]. As first-class artifacts, models are elevated from the level of documentation to being integrated within the development process or the implementation itself as in adaptable runtime models. Software engineers who embrace this paradigm are able to specify solutions using concept representations from the problem domain. Models created as such are capable of being systematically transformed to implement behavior, either directly or via some intermediate artifact. Another goal of MDE is to protect software developers from the underlying complexities of the implementation platforms.

To realize these goals there are a myriad of underlying challenges which France et. al. [31] considers as wicked problems. Among the attributes of wicked problems are the lack of stopping rules, the idea that solutions can be considered good or bad not true or false, and they possess no ultimate test of the solution[63]. France et. al. goes on to identify the following as major software engineering research areas that influence MDE success:

1. Reuse of development experience;

2. Systematic software testing

3. Technology for compilation

This dissertation pertains strongly to the first and third items. By developing a reusable framework, experience gained may be easily leveraged by language developers, allowing them to focus on domain specific concerns. One of the contributions of this dissertation is a novel adaptable interpreter technology to reduce developmental effort.

Rivera et. al. [64] speculates that there are a lack of real MDE practices and mature tools which support the automation of design, development and analysis of software systems. Furthermore industry's high interest in MDE could wane should engineering research not be able to deliver measurable engineering methodologies and processes to support predictable development of software systems. Rivera et. al sees three predominant challenges to the MDE vision:

1. The specification of behavioral semantics of metamodels to allow for more rigorous analysis;

2. Support for a temporal dimension for behavioral descriptions;

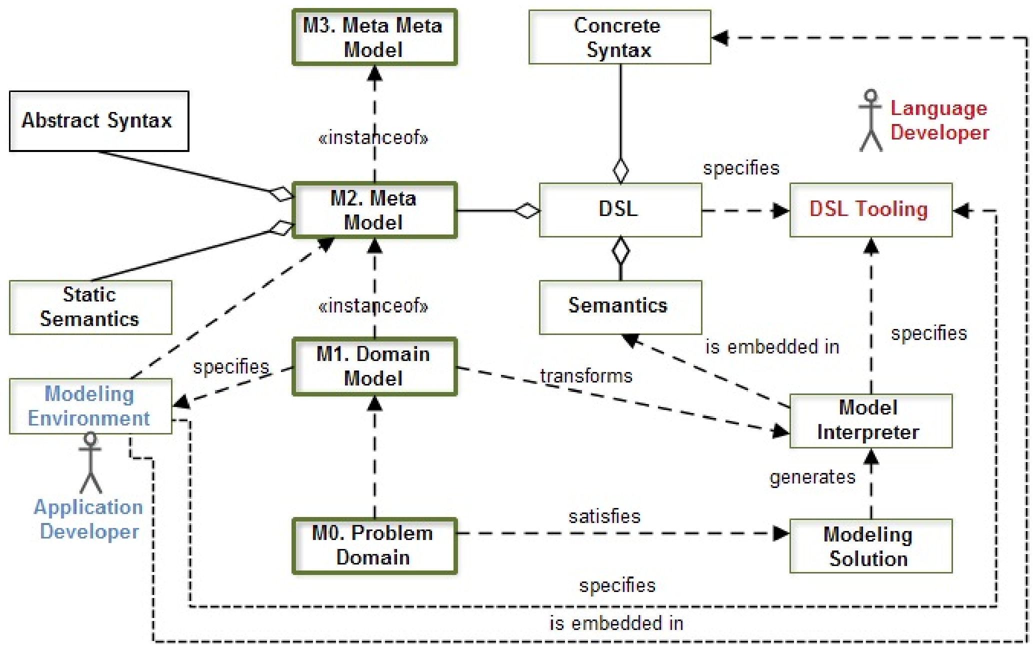


Figure 2.1: DSL Concept Adapted from [78]

3. Tackling essential complexity through the effective use of complementary viewpoints.

This work most directly targets the first challenge as the framework allows for development efforts to be focused on behavioral semantics in the production of DSK extensions and ignore the peripheral commonalities of model interpretation.

This dissertation adopts the view of MDE as a systems creation paradigm related to the design and specification of modeling languages based on a four layer architecture as in [4]. Within the MDE architecture, the lowest or M0 layer, represents *'real world'* objects. At the M1 layer are abstractions of M0 as models. M2 models are metamodels that defines a set of valid M1 models. At the highest level are M3 metametamodels whose concepts describe the M2 metamodels. This architecture allows for the creation of the key construct of the MDE paradigm, namely the domain-specific languages (DSLs). Within this dissertation the distinction between DSLs and Domain-Specific Modeling Languages (DSMLs) are not made and are used interchangeably; the practice of MDE requires viewing textual and graphical language representations as models.

2.1.2 Domain-Specific Modeling Languages

Domain-specific languages also known as little languages are so called as they are comparatively smaller than their general purpose counterparts and possess focused expressive power [76]. Compared to general purpose languages, DSLs may provide a more optimal solution for well defined application domains [77]. Operating within an application domain, the DSL inherits the constraints and assumptions of that domain. Empirical data from [46] suggests that the use of DSLs increases reliability, usability and flexibility. Informally the term *domain-specific modeling languages* DSMLs is used to refer to graphical DSLs [29].

Figure 2.1 shows the interrelationships of the MDE layers and how they currently relate to DSL development and utility. A DSL comprises at least one *Concrete Syntax*, a *Metamodel* and its *Semantics.* The *Metamodel* is further comprised of an *Abstract Syntax*, and its *Static Semantics.* The dynamic or execution semantics of the DSL is embedded within the *Model Interpreter* which is used to generate a *Modeling Solution.*

Of particular relevance to the motivation for the dissertation is the connectedness of the *Language Developer* (Upper-right) and the *Application Developer* (Lower-Left). Although these roles are seemingly separated, in a majority of cases their roles become one in the same. Application developers are often required to assume the role of the language developer, which requires a substantial degree of expertise in interpreter development. The objectives within this dissertation addresses the essence of this problem by capturing some of this expertise within a GMoE framework.

DSLs allow for significant gains with respect to ease of use and productivity when compared to their general purpose counterparts [55]. The ease of use and focus within the scope of the domain lessens the expertise required and translates to the languages widening their usage base to a larger group of developers.

The investment in developing a new DSL is significant. DSLs are usually developed in four discrete stages [78]:

- Analysis - In this stage the domain is inspected and the predominant features, concepts and elements are gathered and represented as artifacts. This is the stage where the language developer gains vital knowledge regarding the domain.
- Design - At this stage the abstracted artifacts are used to produce a specification for the language.
- Implementation - This stage concerns the development of the interpreter responsible for transforming models which comply to the language specification to lower level artifacts or behaviors within a controlled systems as is the focus of this dissertation.
- Usage - This is the stage where that domain users or application developers may construct models and apply them using the tooling to solve some problem.

The challenge in developing DSLs is that along with language development expertise there is a significant amount of domain knowledge required. To compound this issue these requirements may fall on the shoulders of the same person or small team [55]. DSL development decisions are often left postponed resulting in the majority never reaching the implementation phase.

To delve deeper within domain-specific languages a more formal definition is needed. Chen et al. [11] defines a DSL as a five-tuple, $< C, A, M_c, S, M_s >$.

- The concrete syntax (C) - expresses elements in the abstract syntax. It defines the notations used to create the model.

- The abstract syntax, (A) - defines the language concepts, their relationships and related integrity constraints for the language.
- The semantic domain (S) - is a set of expressions with well defined behaviors or meaning in some domain and is usually represented in some formal framework.
- The syntactic mapping ($M_c : C \rightarrow A$) - assigns syntactic constructs to elements in the abstract syntax.
- The semantics mapping ($M_s : A \rightarrow S$) - relates the abstract syntactic concepts to the semantic domain. The abstract syntax and the static semantics together represent the meta-model of a DSML. The meta-model is specified as $< A, C, M_c >$.

The spectrum of DSMLs that are used in the generation of valid software artifacts and applications include those that are translated into an intermediate general-purpose high-level programming language, and those whose models are directly executed by a model execution engine [10]. This dissertation relates to the latter class of DSMLs, which we will refer to as *interpreted DSMLs* (i-DSMLs). i-DSML's are directly interpreted and models may be reconstructed (synthesized) during runtime according to interpreter constraints, system state and the state of the domain elements under control. The advantage of i-DSML's is the capacity for model changes and debugging at runtime without regeneration, retesting and redeployment, [16] . Model execution has become a core interest in model driven engineering (MDE).

2.1.3 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) pertains to a methodology which provides the means by which to separate or modularize cross cutting concerns [66, 28, 14]. A *concern* in this sense pertains to an interest important or otherwise critical to a stakeholder and relates to the development, operation and maintenance of a software system [75, 25]. The weaving of concerns within and across a software's architecture diminishes its maintainability and the reusability of components significantly [66]. First coined by Dijkstra in [18], the term *separation of concerns* refers to a principle which states each concern is best treated in isolation from others [25]. This principle alludes that within most large software systems concerns crosscut each other and are woven within the architecture. The implementation of these concerns are scattered throughout the software itself and requires aspect aware refactoring to modularize concerns.

AOSD seeks to modularize the concern to develop software with enhanced maintainability and reuasability, which leads to better software. To develop a system with separated yet cooperating modules, the paradigm relies on aspects and join points as its core concepts. An *aspect* is a module capable of encapsulating and implementing a concern and refers to join points [36]. *Join points* are well defined points within a programs control flow where the an aspect is incorporated. Join points may be in the form of function calls variable reference or even an assignment statement. A set of join points which signify where an aspects advice is invoked is referred to as a *pointcut*.

In our research we augment our model driven approach with aspect-oriented refactoring whereby we treat the DSK as concern to be made distinct and isolated.

2.1.4 Microgrid Energy Management

The changes in energy consumption patterns are being dictated by rising energy costs and higher demand. These changes manifest themselves as additional requirements of the legacy electrical grid. The United States Department of Energy and similar entities across the globe have been tasked to upgrade the single largest interconnected machine on the planet, the electrical grid. The existing grid, heavily reliant on fossil fuels, has effectively remained unchanged since the early twentieth century and is reaching its functional limits. The smart grid is seen as the successor to the legacy grid. A core component central to realizing the smart grid concept is the microgrid.

Microgrids are atomic self contained energy grids which conceptually should monitor their consumption and co-generate their own power [37]. As such, microgrids may operate in grid connected or in off grid mode. The ability to operate in isolation of the macrogrid requires Distributed Energy Resources (DERs) which constitutes Distributed Storage(DS) and Distributed Generation (DG). DERs provide the ability to leverage renewable energy efficient technologies to flatten peaks in consumption by infusing DERs at intervals of high consumption. With the advent of this conceptual model, consumers may set energy use policies leading to an increased awareness, consumer participation, and eventually a limitation on peak usage.

The impact of microgrids is a key aspect to realizing the goals of the smartgrid [37].There has been significant effort in the research community to address energy management within the microgrid domain[20, 86, 43]. A prominent definition and the one our work is based on is the CERTS microgrid concept (CM)[50]. Distributed energy management is a paramount concern increasing robustness, normalizing quality of service and reduce communication traffic with the smartgrid.

Lasseter et al. [50] state that the structure of the CM is based on an aggregation of loads and microsources operating as a single system providing both power and heat. Figure 2.2 is a diagram of the CM structure adapted from Marnay and Bailey [53]. On the left of the figure there is a point of common coupling that interfaces with the macrogrid e.g., utility company.

The larger grid will see the microgrid as a collection of loads and sources aggregated as a single controllable unit. When connected to the larger grid, the microgrid may expand the usual role of consumer, to that of a producer through its DER components. In off grid or islanded mode the microgrid needs to balance demand and supply which it accomplishes through reducing controllable loads (load shedding) or

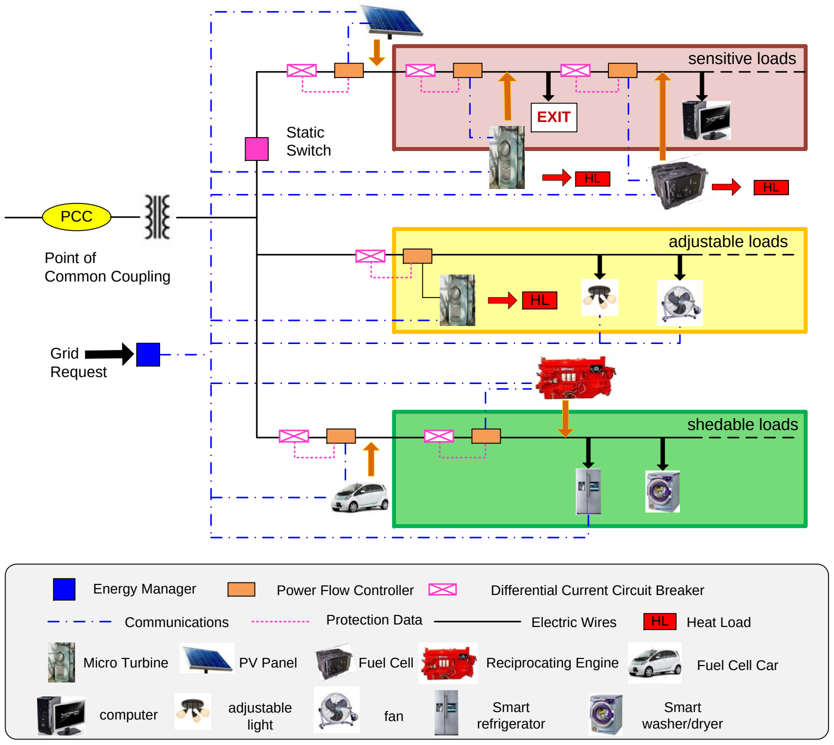


Figure 2.2: The CERTS Microgrid Concept.

by bringing more power sources (DERs) online. The dynamism of this system relies on capable energy management systems.

The pertinent components of the CM which are abstracted in the development of MGridML are:

PCC - or Point of Common Coupling, represents that point where the microgrid may connect or disconnect from the distribution grid.

Loads - are devices that consume electrical energy. Dependent on their perceived role these devices may be classified as controllable or non controllable. Aside from controllability there is also a need for further grouping into (1) *sensitive loads* - loads that must be met if at all possible with distinct quality of service requirements, (2) *adjustable loads* - can set the amount of load shed, and (3) *sheddable loads* - loads that are abandoned first if there is a power shortage.

Sources - Distributed Generation (DG) elements include generation resources which may rely on renewable or non renewable sources. We categorize DG elements as dispatchable or nondispatchable based on controllability. Consider wind power as nondispatchable as it is not an on-demand technology as are fuel cells.

Storage - Distributed Storage (DS) serves to stabilize sources in the MG in the event of load fluctuations and bridges the gap caused by the intermittent nature of sunlight and wind when used as an energy resource. DS elements may facilitate operational transitions from a black start. A microgrid black start is the process of recovering from a partial or total loss of power. DS elements include super capacitors, battery banks and flywheels [48].

Smartmeter - The smartmeter constitutes the portal to the advanced metering infrastructure where data is exchanged with the smartgrid via radio frequency or over powerline communication methods.

Physical Controllers - Plant elements may contain integrated or non integrated controllers which are capable of changing the operational state of said device. Physical controllers may be composed into logical controllers at a higher level of abstraction to simplify management of incongruous and dispersed devices. The user therefore may define groupings of devices to be managed in concert without any knowledge of the physical controllers or which devices they control. As a case in point all outdoor lights may be grouped within a logical controller which may have the directive to switch all its devices on at 6pm. The interpreters middleware would make the required mapping and execute the task.

MGCC - Microgrid Central Controller is responsible for coordinating the operation of the microgrid. This controller orchestrates the delicate demand supply balance.

2.2 RELATED WORK

This dissertation bears upon a myriad of MDE concerns. It is apparent that this dissertation cannot capture all the extensive subfields and the rich research within. In that light, this section we address five specific areas of related work that most directly influences and relates to the GMoE: (1) operations essential to manipulating models at runtime; (2) the synthesis as it relates to programs and models; (3) approaches that support runtime model execution; (4) the first i-DSML, the Communication Modeling Language (CML) and the Communication Virtual Machine (CVM), the execution engine for CML is described next. CVM plays a crucial role in the development and evaluation of the GMoE approach at the heart of this dissertation; and (5) approaches to managing energy within the microgrid.

2.2.1 Model Operations

Being grounded in MDE, our approach treats models as first class artifacts. We extensively utilize model operations as a basis of our operational semantics; the main operation being model comparison. The model comparison algorithm we employ yields changes which fall within three distinct categories *add, remove* or *modify.* These changes are based on the difference of an existing model with a new proposed model; recall i-DSML semantics are based on changes to models at runtime. These changes which occur may in turn drive changes to the existing model dependent on the state of the system.

There have been several approaches proposed for the comparing, composing and transformation of models. Alanen et. al. [2] describes a method of model composition through the use of generic difference and union operations. It identifies a set of primitive operations which allow model differences from various models to be identified from a base model, and then to have these changes composed into a new model. The primitive operations include those used in this work and we respect the approach that model differences are not models.

Yuan et al.[79] describe a method for change detection of XML documents. This method would allow for the detection of unordered elements in a model. While our models are easily represented as XML documents (for example, the use of XMI for persistence) as they are indeed graphs, models resident in memory may utilize more appropriate representation mechanisms. Additionally, our approach incorporates an event driven mechanism to initiate change notifications. The composition of our various models requires a fundamentally different approach to finding unions and differences. As pointed out in Kelter et.al.[45], although XMI and other XML based models have primarily tree-like structures, they are not exactly trees as they may contain cross references to other elements in the model, or in external models. Wolfe et al. [82] utilizes an approach of graph rewriting rules to maintain consistency between models, when inconsistency occurs though the failure of a component which results in its removal from the model. Our approach differs in that changes may occur from various events, which may result in the addition or removal of elements from our model, and model consistency must still be maintained.

To the best of our knowledge there is no work pertaining to changes to models at runtime to define the semantics of a DSML. Stanek et al. [70] provides groundwork for the theory of labeled attributed graphs and the graph differencing problem. Kelter et al. [45] presented work regarding the development of a generic difference algorithm for UML models. Their approach compute the differences between UML models encoded as XMI files and reported satisfactory performance.

Xing et al. [84] describes an algorithm, *UMLDiff*, that automatically detects structural changes between designs in different versions OO of the software. UMDiff traverses two class models identifying corresponding entities based their name and structure similarity. Unlike the approach by Xing et al. that compare UML models based on the UML metamodel, Lin et al. [51] present metamodel-independent algorithms and tools for detecting mappings and differences between DSMs. Lin et al. define their models as a typed attributed hierarchical graph. The approach we used to identify model changes is based on a labeled attributed graph, as a result if a change is made to an attribute we consider the node in the graph as being replaced. The work by Xing et al. [84] and Lin et al. [51] tend to focus more on model evolution than on using changes to support the dynamic semantics of a DSML.

2.2.2 Model Synthesis

The term *model synthesis* has been used in several contexts in the domain-specific modeling community. The most common use of the term model synthesis is the translation of a higher-level more abstract model, into a low-level more concrete model [54]. An example of synthesis is the (semi)automatic generation of source code from DSML models. Since models are based on graphs, many of the operations used to perform model-to-model transformations are based on graph transformations [1]. Our work uses some of the basic concepts used in graph transformations such as comparing graphs to identify model differences [2, 51, 70, 84], which are the changes used in our model comparison. In addition, unlike the traditional approaches we transform an MGridML model, a more abstract model, into control scripts, a lower-level model, while the application is running, i.e., we perform runtime model synthesis.

Bencomo et al. [6] describes how models at runtime may be used to synthesize software artifacts, particularly mediators, during the execution of a system to solve the interoperability problem in the networking domain. Their approach uses discovery and learning methods to capture and refine knowledge of the context and environment of the running system to create a runtime model. The behavioral semantics for the runtime model are captured using labeled transition systems (LTSs), which model the interaction protocol. Using the knowledge of the runtime models, including the current structure and behavior, software artifacts (mediators) are synthesized on the fly. In the broad sense the approach by Bencomo et al. is similar to ours in that it manipulates models at runtime, it keeps track of the current state of the system, and the models are causally connected to the running system. However, unlike our approach their runtime models are created without user involvement and do not represent the user requirements of the running application.

Mannadiar et al. [52] describes an approach for synthesizing artifacts from domain-specific models (DSM) using layered model transformations. The authors use the *PhoneApps* DSML to develop a DSM to build a phone application, then apply a set of rule-based transformations to compile the PhoneApps model into increasingly low-level code until the complete Google Android application is created. The DSM consists of three models, a statechart for the behavior, android screens for the UI, and phone features that will be used. Unlike our approach, Mannadiar et al. captures the behavior of the phone application by transforming PhoneApp containers into a statechart using several predefined rules. In our approach we assume the LTSs for the controllers have already been created by an expert in the microgrid domain and used during the runtime synthesis process. Their approach also exploits existing features in the phone to create the application, similarly we use the services provided by the plant controllers to realize the microgrid solution (see Figure 3.10).

Edwards et al. [21] describes an approach that automatically synthesizes configurations from models for flexible model analysis and code generation; these models are mainly in the software architecture space. Their approach enhances the domain-specific language (DSL) metamodel with additional semantics that enable the generation of configuration files and plug-ins. The premise of their work is that model editors and model interpreters are isomorphic therefore rendering models in an editor is just another form of model interpretation. The semantics are applied to the objects in the metamodel thereby supporting the creation of a metainterpreter and model interpreter framework that together perform the synthesis of various tools. Unlike the approach by Edwards et al., we do not generate any code for tools to support analysis of the software for the application. Our model synthesis directly executes application models at runtime and generates controls scripts to be interpreted by the next layer in the DSVM.

Automated program synthesis, or proof theoretic synthesis, introduced by Sumit and Gulwani[33, 34, 35] is a methodology to generate executable programs from user intent expressed in terms of constraints. This research direction, as with ours, is considered enabling technology for non-expert programmers; effectively broadening the spectrum of users by providing the tooling support through abstraction to generate complex applications via synthesis. Users of program synthesis may declaratively stipulate their intent (functional specification) in terms of examples; logical relation between *inputs* and *outputs*. This is analogous to our user specifying a model of intent M_{user}, *output*, and viewing the current runtime model $M_{runtime}$ as *input*.

These approaches diverge in how the researchers synthesize programs from the analysis of ($input \rightarrow output$), and how we synthesize new models and control scripts from model changes,($M_{runtime} \rightarrow M_{user}$). Our approach utilizes labeled transition systems specified by the i-DSML author(s) to interpret the model changes. Program synthesis utilizes searching techniques based on exhaustive search, version space algebras, machine learning or logical reasoning techniques applied to a search space over imperative/functional programs, or restricted models of computation.

Wu et.al. [83] presented an earlier approach to model synthesis for an i-DSML, CML, developed within our research group. This approach presented the initial approach to define the behavioral specification for an i-DSML which relies on the dynamic synthesis of models to produce control scripts. This approach was targeted solely to the user centric domain, however we were able to draw upon its core interpreter logic to define the next generation i-DSML (MGridML). Our current approach refines this earlier approach and extends the synthesis process to be generalized across both i-DSMLs. More precisely, the approach to defining model synthesis relies on four main processes:

- A *SE Controller* - responsible for coordinating the transformation of incoming model instances and updating the environment.
- A *Schema Analyser* - responsible for identifying the changes to the models then submitting such changes to the *Connection* process.
- A *Connection* - responsible for maintaining the subprocesses for (re) negotiating a connection or transferring media during a session. This process is specific to the communications domain.
- A *SE Dispatcher* - responsible for updating the upper and lower layers of the DSVM and sending action requests to the *SE Controller* for environment updates.

The more significant distinction between the two approaches concern revisiting these processes to ensure greater modularity and aptness to be extended and generalized. The functionality of the *SE Controller* and *Schema Analyser* were revised to spawn separate modules for model comparison, change interpretation and runtime update. This revision required an explicit runtime model which causally represents the controlled system. We further encapsulated model changes (add, delete, modify) within a *change list* as the means of communication between the model comparator and change interpreter. The domain specific *Connection* was separated into the DSK semantics and made persistent by using labeled transition systems based on the state machines presented by Wu. This dissertation will present the systematic refinement of this early architecture, making it more supportive of generalization and allowing for the separation of DSK and GMoE within the execution semantics of an i-DSML.

2.2.3 Model Execution

Model execution is a central concern of MDE. DSMLs are a key feature to realize MDE objectives. There are two distinct approaches to realizing behavior from DSMLs, code generation and direct interpretation. DSMLs which are analyzed by transformation engines to synthesize intermediate artifacts such as high level languages [44]. Utilizing intermediate artifact do however have its drawbacks as changes to models at runtime require regeneration, retesting and redeployment. Edwards et.al. [22, 23] presents a model interpreter framework to automate DSML development at the language and interpreter level by employing an abstract component technology. Our work differs in two critical aspects. The approach used by Edwards et al. concerns the transformation of models to an intermediate high level language. Secondly, the semantics of our approach are derived from changes to models at runtime.

Several approaches utilize action languages to thread executability within metamodels such as Kermeta [58], xOCL [12] and using abstract state machines [62]. These approaches allow for intuitive development of models, however they do not specify how the models are to be interpreted or a generalized methodology as outlined in our work. Combemale and Pantel have however proposed a design pattern called the executable DSML pattern [15] targeted at the development of a reusable model of computation similar to our model of execution. Sadilek and Wachsmuth presented a similar approach, *EPROVIDE* [65], providing executablity to DSMLs along with interpreter specification support, prototyped using petri-nets. The fundamental difference in approaches is that our semantics is based on model changes at runtime.

2.2.4 The Communication Modeling Language

The increased user demand for communication solutions to leverage technological advancements, has led to the search for highly customizable solutions integrating voice video and data. In addition, many users of communication solutions require a level of abstraction that allows them to specify their requirements using terminology from the user-centric communication domain. The scope of the term communication is limited in this dissertation to designate electronic media over a data network.

As a result of this need the first i-DSML, *Communication Modeling Language* (CML) was created by Clarke et al. [13] and its DSVM the *Communication Virtual Machine* (CVM) by Deng et al. [17]. There are two versions of CML: a graphic based (G-CML) and a textual X-CML which is a XML derivative. The base operations of CML are:

- Data transfer
- Add/Remove participants
- Establish connection
- Data transfer specification
- Dynamic structuring of data for transfer

Figure 2.3 shows the abstract syntax for CML. The language has `CommunicationSchema` as its root construct which may be either a `ControlSchema` or a `DataSchema`.

A `ControlSchema` defines the configuration of the communication instance. it comprises one or mode `Connections`, one or more `attachedParty`, and one or more `DataTypes`. In turn each `Connection` has one or more `Devices` and one or more `DataTypeRefs`. Each AttachedParty has one or more `IsAttached` and one or more `Persons`. Datatypes are abstract representations of a `FormType` or `MediumType`; with a `FormType` capable of being composed of `FormTypes` or `MediumTypes`. This rich metaclass relationship gives CML the capacity to describe the users communication requirements intuitively.

The `DataSchema` carries the actual data used in the communication session. A `DataSchema` may be a `Request` or `DataContent`. A `Request` can be either a `MediaRequest`

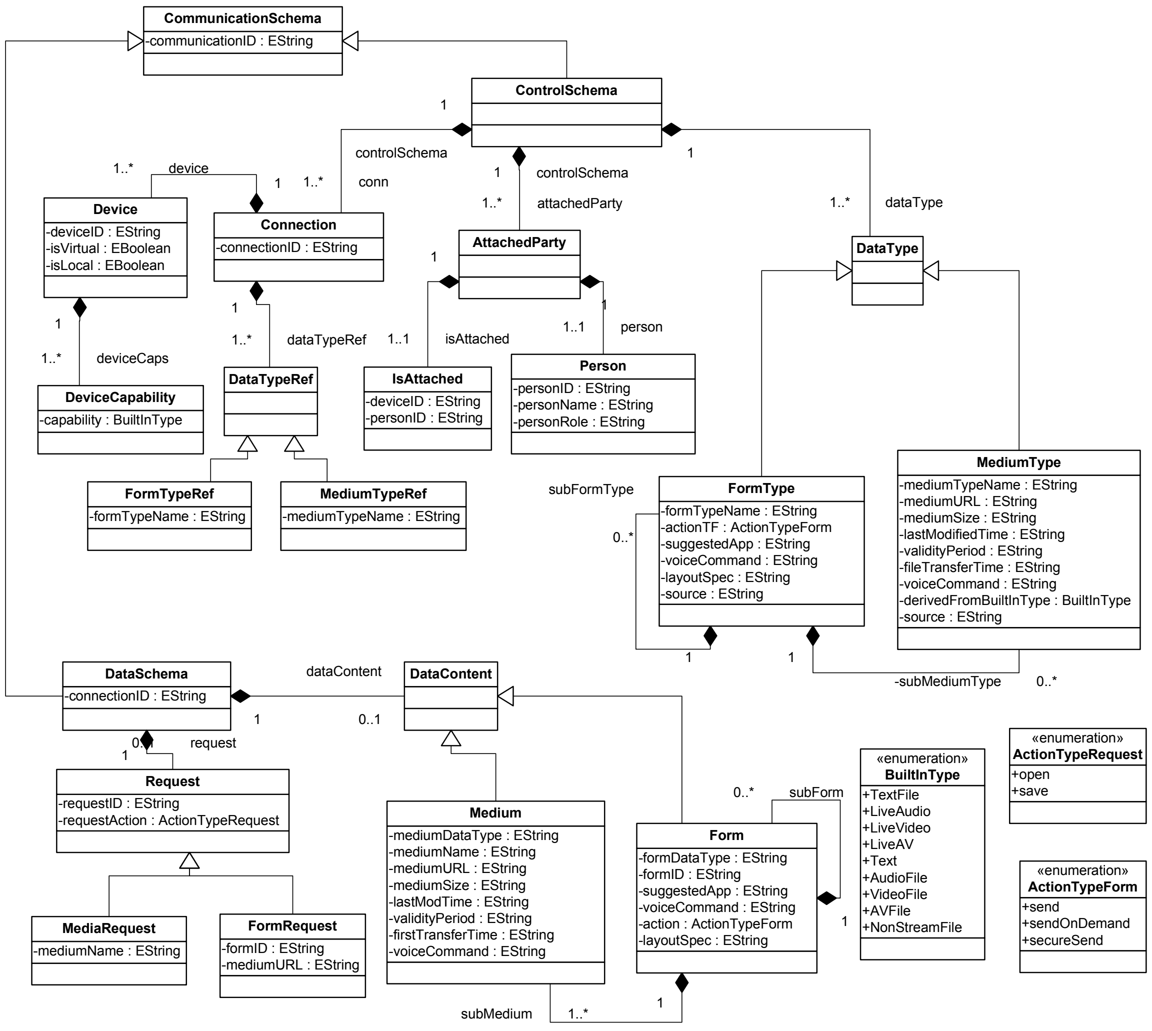


Figure 2.3: CML Abstract Syntax

or a `FormRequest`. A `DataContent` can either be a `Medium` or a `Form`. `Forms` may have `subForms`.

Figure 2.4 presents the four layered architecture of CVM which realizes the user-defined communication preferences captured by CML model instances. The four layers of the architecture are:

- A User Communication Interface (UCI) provides the environment for the users to specify their requirements.

- A Synthesis Engine (SE) transforms an adaptive runtime model according to user preference models, platform capabilities and environmental events to coordinate a communication session by the generation of communications control scripts (CCS).
- A User-centric Communication Middleware (UCM) executes the control scripts to coordinate the communications service.
- A Network Communication Broker (NCB) furnishes a network independent interface with the underlying communications frameworks to actualize the requests.

CVM is a distributed system whereby each communication hub requires a running instance. Later in this disserattion we will show how we have enriched and refined the earlier ad-hoc approach used in CVM. Using CVM as a basis the exploration of inherent commonalities within the interpreters was launched and has provided the basis for the specification of the execution semantics presented within this work.

We now overview the model synthesis approach used in CVM. The approach represents that of the first i-DSML and as such the prototype, which while functional is rudimentary with little or no separation of somain specific concerns. Revisiting this early system is however essential as we will later show how it is refined and its operational semantics captured using persistently represented DSK artifacts.

Figure 2.5 shows a high level representation of the synthesis engine of the CVM. The process begins when the user submits a validated model consisting of a control schema (CS) and data schema (DS) pair via the user interface (UCI). The SE controller directs the analysis of (CS,DS) via Schema Analysis which does a comparison of the model to the current runtime model and generates the schema changes. Dependent on the CS event type a renegotiation is initiated or the current negotiation scheme is updated. The type of DS event may trigger a new media transfer or update

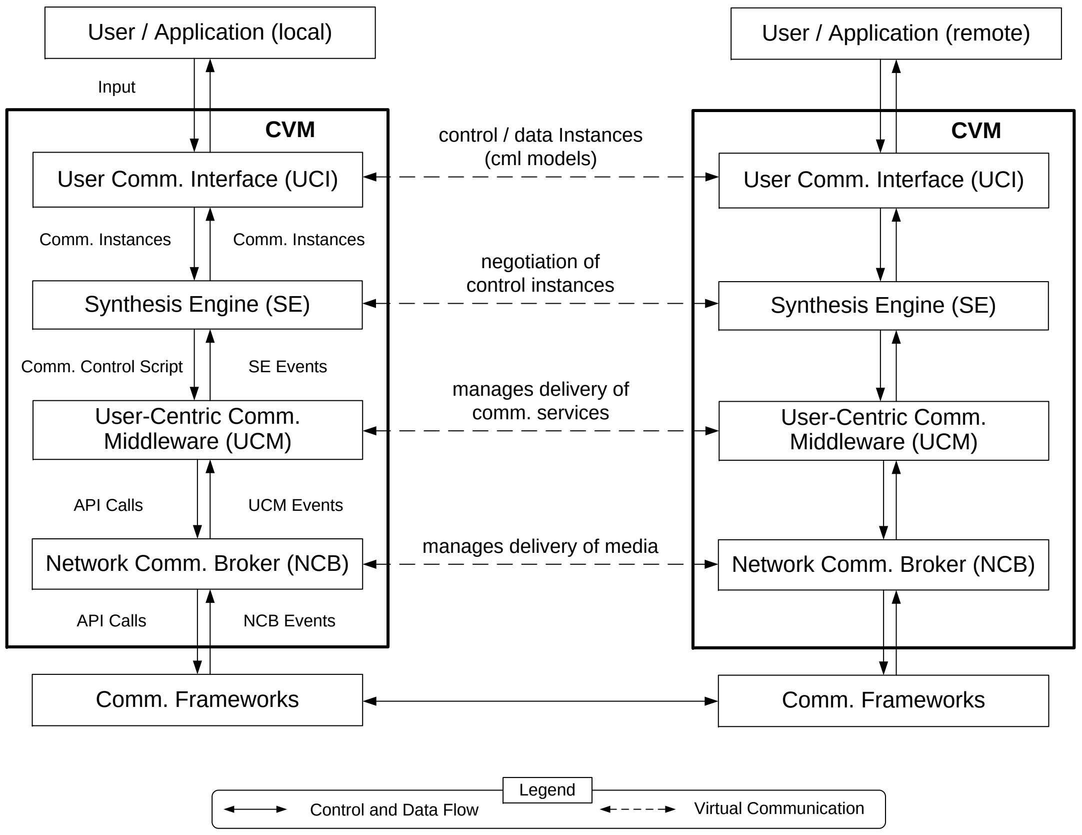


Figure 2.4: The CVM Layered Architecture

an earlier one. A byproduct of the media transfer and (re)negotiation processes are control scripts to the middleware and a model to update the UCI. The SE Dispatcher is responsible for the submission of these artifacts to the upper and lower layer of CVM.

The CVM SE views a connection as a link between two or more participants (persons) within the same communication space. The initial runtime model is (CS_0, DS_0) which is the null model; indicating there is no connection present. The initial model represented by (CS_1, DS_1) carries the initial communication requirement. DS_1 will be null as media transfer requires first a connection to be established.

We may view the process more formally as:

$$((CS_{i+1}, DS_{i+1}), Env_i) \rightarrow ((CS_{out}, DS_{out}), Sc_{i+1}, Env_{i+1}) \tag{2.1}$$

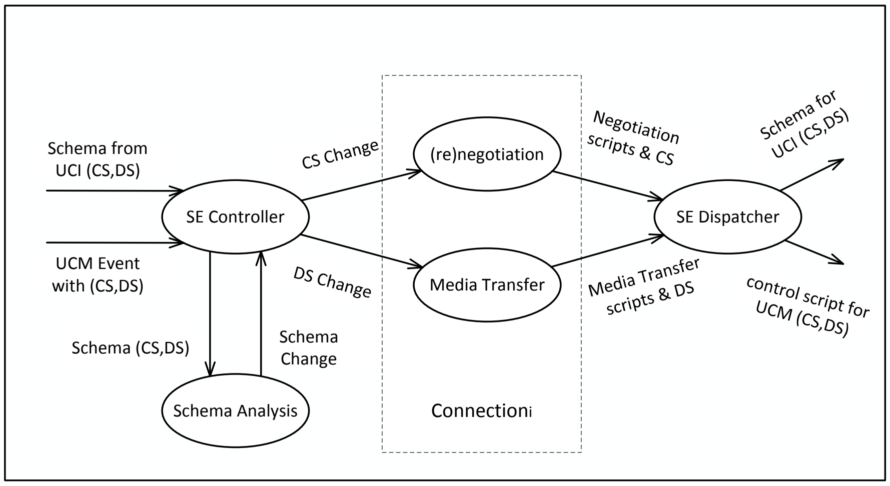


Figure 2.5: Overview of the CVM Synthesis Engine. CS - Control Schema; DS - Data Schema

where: $(CS_{i=1}, DS_{i+1})$ is the next model to be processed;

Env_i is the current executing environment which consists of:

(CS_i, DS_i) is the current runtime model. $CS_i \in \{CS_{exe}, CS_{neg}\}$

The elements of the set $\{CS_{exe}, CS_{neg}\}$ *are a Control Schema either being executed or being negotiated.*

(CS_{out}, DS_{out}) is the new model generated by the synthesis process;

Sc_{i+1} is the control script for the UCM layer; and Env_{i+1} is the new environment subsequent to synthesis.

2.2.5 Microgrid Energy Management Approaches

With respect to our approach to energy management, much of the work in the area of microgrids tend to focus on the electrical aspects such as efficient designs and hierarchical integration [39, 43, 86] of generation and load into existing electric power distribution infrastructure. At a high level, this work compliments much of the aforementioned research by providing orthogonally, a simplified software engineering approach to support the management of the microgrid through models.

Several approaches to energy management utilize software multi-agent systems (MAS) performing in concert to automate equilibrium between sources to loads. Pipattanasomporn et.al. [61] presents a MAS for managing the microgrid whereby agents communicate to work in concert to detect and island the dynamic system. Dimeas et.al. [19, 20] describes an approach which utilizes MASs to control devices in a marketing environment. Approaches using MAS are typically distributed by nature, however our approach is hierarchical centralized control which lends itself to a more resilient architecture.

The DSL approach is utilized in Habitation [40], a domain specific language for home automation system design. Similar to our work, a model-driven paradigm is employed, providing a higher level of abstraction to the user of the tool. The Habitation language however, targets the representation and manipulation of loads alone as the aim is home automation. MGridML is designed to address the complete energy system with algorithms concerned with the balancing of energy between loads and sources. Additionally, Habitation uses a code generation methodology while MGridVM uses a runtime model interpretation technique to support dynamic reconfiguration of the microgrid.

CHAPTER 3

DEFINITION OF SEMANTICS FOR SYNTHESIZING MGRIDML MODELS

In this chapter we outline the development of the language designed to manage demand side smart grid elements, MGridML, and its associated DSVM, MGridVM. An illustrative scenario will thread this chapter. The focus of the discourse is the MoE developed within the synthesis engine. A model of execution (MoE) for the synthesis process that takes place during the realization of user-defined MGridML models is defined. This definition is based on changes to control and data instances at runtime. An overview of the synthesis process is provided, then a description of how the control and data instances are analyzed and the resulting model changes interpreted. The chapter concludes by illustrating how the synthesis process is applied and incorporate the evaluation of the implemented prototype. This prototype has high significance within this dissertation as it is used as a baseline for our comparative studies for the GMoE inspired prototype in Chapter 5.

3.1 Microgrid Modeling Language

In this section the metamodel for MGridML, which consists of the abstract syntax and the static semantics, is defined. The metamodel is represented using a UML class diagram (Abstract Syntax) and OCL statements (Static Semantics). Prior to creating the metamodel, a domain analysis was performed on the microgrid domain to identify the mandatory and variable features of a microgrid. To illustrate an application of MGridML, we create models for the Winter and Spring configurations using a generic concrete syntax for the scenario in Section 3.1.2.

3.1.1 Domain Analysis

Undertaking the task of developing a language for the application in a particular domain is nontrivial as significant effort is required [55]. According to Simos [67], candidate domains should be reasonably stable where it is worth studying, mature, and economically viable to do so. We contend that energy management within the microgrid is an ideal candidate for implementing a DSL solution.

Domain analysis is performed using the Feature Oriented Domain Analysis (FODA) approach [41] which identifies pertinent entities within the domain, yielding a taxonomy model for microgrids. Analysis of the application domain requires a thorough grasp of the specifics to capture mandatory and variable features [55]. The analysis centered around the CERTS microgrid concept(CM) [50] (See Figure 2.2) whereby we determined the scope, terminology, concept descriptions and feature model for the domain. Figure 3.1 shows the feature decomposition of the domain in the form of a feature diagram. In the lower right you will see the key which explains how the various features of the domain are constituted. As an example, the diagram indicates that a mandatory feature of the smart microgrid is power supply which may include a combination of external and internal resources; the latter of which may in turn be composed combinations of power sources and storage elements. Here we see the smart microgrid requiring the main features of:

- A Power Infrastructure;
- Energy Management;
- Privacy;
- Tolerance;
- Scalability

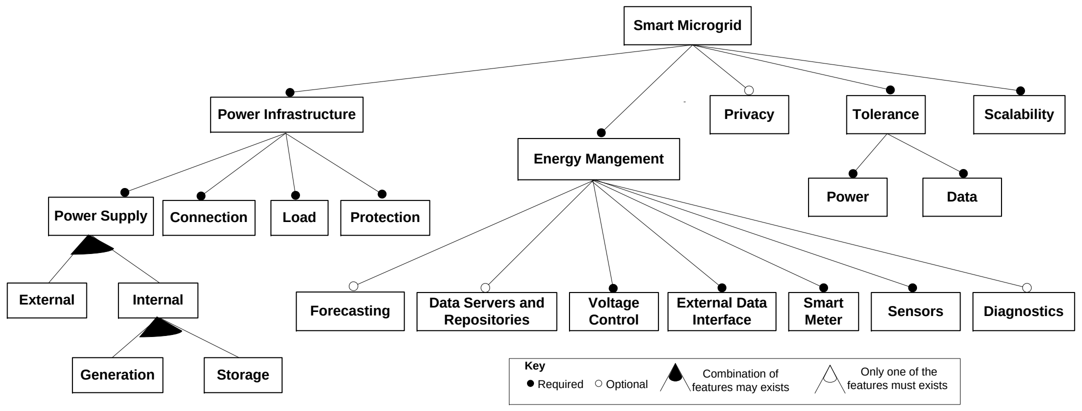


Figure 3.1: The Microgrid Feature Diagram.

The language's metamodel attempts to capture the required features of the *Power infrastructure* in terms of its possible configurations while respecting the concerns of *Tolerance* and *Scalability*.

Not all features of the domain can be represented solely by the language. Mandatory features such as tolerance which is excluded from the language is incorporated within the virtual machine as an embedded cross-cutting concern.

3.1.2 Illustrative Scenario

We present a scenario from the domain of microgrid energy management that will thread this chapter. The scenario will be used to illustrate how MGridML models are used to infer the semantics during the synthesis process. The primary actor in our scenario is Dana, a homeowner in northeast United States where there are distinct changes in seasonal temperature and accordingly dissimilar energy profiles. Her home contains a cooling system (AC), a heating system (heater), several sensitive electronic devices and a pool filtration system (pool). The home is connected to the utility via a smartmeter, and uses a battery bank as a secondary source of energy.

Change Season Scenario. Dana needs to reconfigure her devices to adjust for seasonal changes. As the seasons change from winter to spring, anticipated climate changes necessitates plant reconfiguration. She will need to activate the AC and pool, while turning off the heating system. If the power from the utility is interrupted, the energy for the home is provided by the battery (*islanding*) until it is depleted or utility power is restored. □

One option Dana has is to reconfigure the elements in her plant manually. However, in manually configuring the plant elements, Dana is faced with the issues of: (1) managing devices on a live system without knowledge of the active or current state of these devices; (2) estimating the rate of energy usage during islanding; and (3) reconfiguring the microgrid to replenish the battery when the microgrid is reconnected to the utility. Alternatively, she can access a mobile device and select an i-DSML model aptly named *Spring Configuration* and press the submit button. An appropriate DSVM can interpret this i-DSML model and her microgrid is seamlessly and safely reconfigured in seconds and she does not have to handle the aforementioned issues.

3.1.3 Metamodel

MGridML is defined in terms of its metamodel comprising its abstract syntax and static semantics. Figure 3.2 shows the abstract syntax and Appendix A contains the static semantics for MGridML. A characteristic of i-DSMLs is the production of two types of models; a control model signifying configurations and control structures, and a data model, which as its name suggests, carries the data properties of entities within the domain. This separation, presented in Section 2.1.2, is appropriate since the data portion is expected to change more frequently than the control portion. This leads to faster interpretation as the entire model of the system does not need to be processed.

The abstract syntax describes a MGridML model (or *schema*, terminology adopted from CML) as either a *control* schema (`MGridControlSchema`), shown on the right side of Figure 3.2, or a *data* schema (`MGridDataSchema`), left side of the figure. A

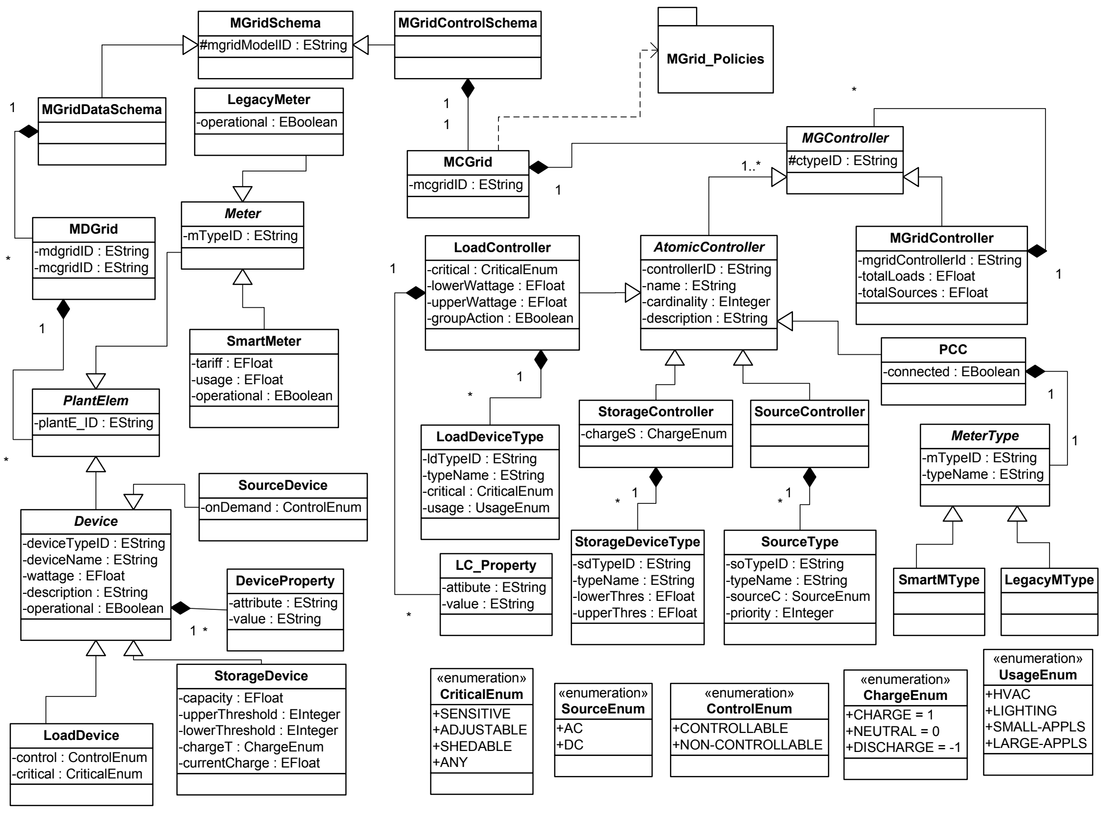


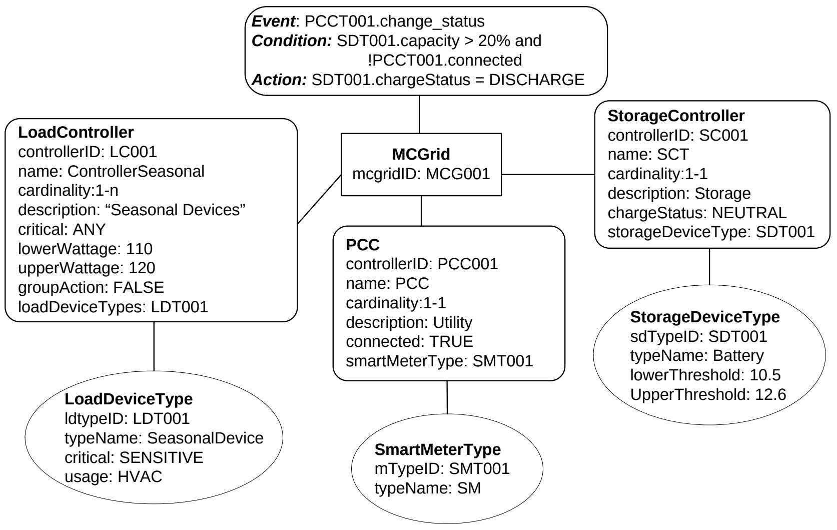
Figure 3.2: MGridML Abstract Syntax Diagram.

control schema (CS) specifies the logical configuration of a microgrid energy management plant. A CS contains a central and singular control entity (`MCGrid`) that is composed of zero or more controllers (`MGController`). These controller can either be atomic or composite entities. A composite controller is used to create a hierarchy of logical controllers. The atomic controller represent the different controllers that are connected to the various device types. Our domain devices include sources, loads, storage units and one point of common coupling (PCC). The PCC controller is connected to one meter type which may be a smart or legacy meter type.

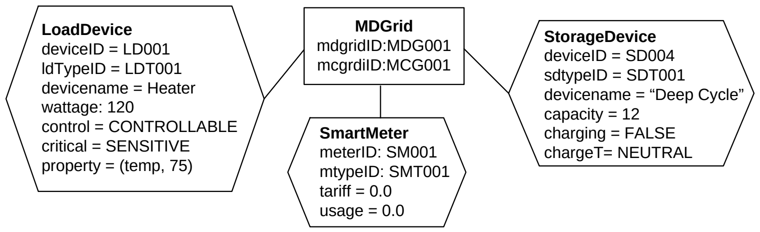
The data schema (DS) contains the actual plant elements, associated with the types defined in the control schema. A DS contains one microgrid data entity (`MDGrid`) associated with a microgrid control entity (`MCGrid`) defined in the CS. The microgrid data entity contains one or more plant elements which may be a meter or a device. The meter may be a smartmeter or a legacy meter. Each device has a set of device properties and may be a storage device, source device or a load device. Device properties provide the flexibility for the user to assign values to various properties, e.g., temperature of the cooling unit is 75 degrees.

There are three concrete notations used to represent models in MGridML: the XML-based (*X-MGridML*), the graphical (*G-MGridML*) and the UI-based (*UI-MGridML*). The X-MGridML form is used internally by the DSVM, the G-MGridML form is used by the expert e.g., the electrician, and the UI-MGridML form is used by the novice or casual user, e.g., Dana in the motivating scenario. Bi-directional model transformations between G-MGridML and X-MGridML, and UI-MGridML and X-MGridML are employed in the DSVM. Before a CS or DS can be interpreted by a DSVM they must be fully instantiated, that is, all attributes must have specific values. We refer to these instances as a *control instance* for a CS and a *data instance* for a DS. Note that in this dissertation we may use schema when we are referring to an instance, particularly during the synthesis process. The languages metamodel comprises not only of the abstract syntax but it's static semantics. A partial list of of the language's static semantics, written in the Object Constraint Language (OCL), is provided in Appendix A.

MGridML Models for the Illustrative Scenario. Figures 3.3 and 3.4 show the MGridML models used to realize the Change Season scenario. The model in Figure 3.3(a) shows the control instance for the Winter season. The central rectangular shape (MCGrid) represents the MGrid controller that coordinates the activities of lower level controllers - *LoadController*, *PCC*, and *StorageController*, shown as rounded rectangles. The controllers are connected to the device types - *LoadDeviceType* with type id LDT001, *SmartMeterType* with type id SMT001, and *StorageDeviceType* with type id SDT001, shown as ovals. Note that each controller has a unique identifier and a



(a) Control instance for the winter season



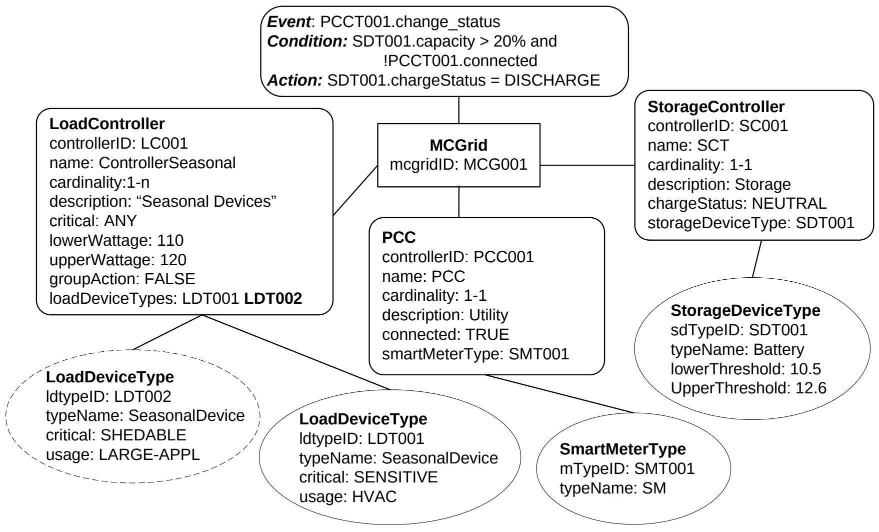
(b) Data instance for the winter season

Figure 3.3: MGridML models for the control and data instances for the *Winter* season.

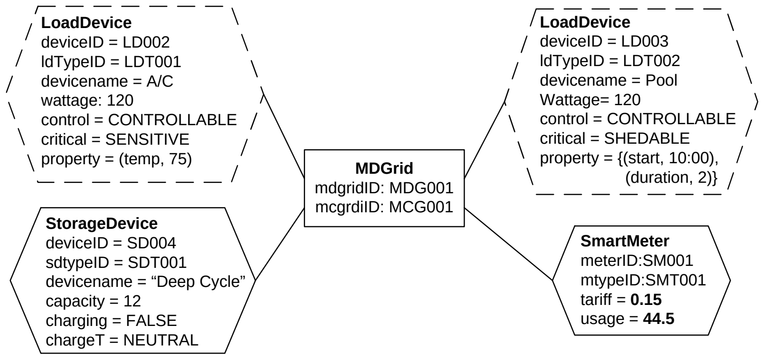
list of the identifiers for the types attached to it, among other properties. For completeness of the scenario we show an event-condition-action policy connected to the MCGrid controller representing when islanding mode is activated.

The model in Figure 3.3(b) shows the data instance for the Winter season. It contains the microgrid data entity connecting: (1) a load device with device id LD001, which connects to type id LDT001 in the control instance, and property *temperature* set to 75 degrees Fahrenheit; (2) a smart meter SM001, mapping to meter type SMT001, tariff = 0.0 and usage = 0.0 - initial values of the meter; and (3) a storage device with id SD004, mapped to type id SDT001.

Figure 3.4 shows the control and data instances for the spring season. The model elements shown with the dotted lines are the additions made to the winter control and



(a) Control instance for the spring season.



(b) Data instance for the spring season

Figure 3.4: MGridML models for the control and data instances for the *Spring* season.

data instances. For example, the *LoadDeviceType* was added to the control instance shown in Figure 3.4(a). Two devices were added to the data instance, shown in Figure 3.4(b), these included a *LoadDevice* with id LD002 for the air-conditioning unit, and a *LoadDevice* with id LD003 for the pool. The storage and smartmeter devices remains unchanged, except for their properties. Note that the plant behavior may be defined through device property changes or addition or deletion of model nodes. In the next section we provide details on how we use the changes in the MGridML models to realize the changing behavior in the plant.

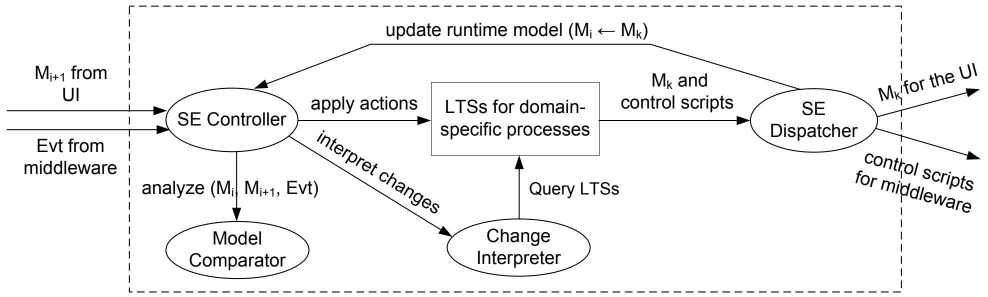

update runtime model ($M_i \leftarrow M_k$)
M_{i+1} from UI
Evt from middleware
SE Controller
apply actions
LTSs for domain-specific processes
M_k and control scripts
SE Dispatcher
M_k for the UI
control scripts for middleware
interpret changes
analyze (M_i, M_{i+1}, Evt)
Query LTSs
Model Comparator
Change Interpreter

Figure 3.5: High level view of the synthesis process

3.2 Synthesizing MGridML Models

To define our model of execution we describe: (1) a formulation of the execution semantics to support the interpretation of MGridML models based on changes to user-defined models at runtime; and (2) a design of the synthesis process that separates domain-specific knowledge (DSK) from the model of execution (MoE). Unlike previous presentations of the i-DSML synthesis process [17, 83], we provide details of the model interpretation, and decouple the DSK from the MoE for the microgrid energy management domain. The expectation is that the MoE can be reused in other domains with minimal changes. The application of the MoE to be modularized and applied to other domains will be done in the next chapter. Now we provide an overview of the synthesis process, then describe how the control and data instances are analyzed, and how the model changes are interpreted. Finally we illustrate how the synthesis process is applied to the models of the Change Season scenario (see Figures 3.3 and 3.4).

3.2.1 Overview of Synthesis Process

Figure 3.5 introduces the synthesis process that occurs in the synthesis engine at a very high level. The *SE Controller* will receive either events from the middleware layer (MCM) or a new model M_{i+1} from the microgrid user interface (MUI). Upon receipt the *SE Controller* will evoke the *Model Comparator* to analyze the new model or event with respect to the current runtime model M_i to produce a list of changes. The list of changes are subsequently interpreted by the *Change Interpreter* using current state of the Labeled Transition systems(LTSs) to produce a new runtime model M_k and a set of control scripts. The *SE Dispatcher* informs the the MUI of the new state of the runtime instance using M_k and submits the control scripts to the middleware for processing. M_k subsequently becomes the new runtime model M_i and the *SE Controller* awaits its new signal. We next look at the process in a more formal manner.

The synthesis process essentially involves comparing two models at runtime and inferring the behavior of the applications based on the changes between the two models. We use Label Transitions Systems (LTSs) [27] to define the domain-specific behavior for the synthesis process. An LTS is defined as a triple $< L, Q, \rightarrow >$, where L is the set of labels, Q the set of states or acceptable configurations, and $\rightarrow$ the transition relation where $\rightarrow \subseteq Q \times L \times Q$.
The synthesis process is formally defined as the function:

$$i_k : I \times S_k \rightarrow O \times S_{k+1} \tag{3.1}$$

where:

I is the input event alphabet that are events (Evt) which may be an event (Evt_{UM}) that signifies the receipt of a new user preference model (U_j) to be synthesized, or an event (Evt_P) representing a change in the microgrid plant.

$U_j = (CI_j, DI_j)$ for $j = 1 \ldots m$, represents the j^{th} instance pair of MGridML models generated by the user. CI_j is the control instance and DI_j is the data instance. This instance pair may be new or one that could not be satisfied in a previous iteration. If a user model cannot be satisfied the instance pair remains immutable and may be reevaluated whenever a change occurs in the plant.

S_k represents the current synthesis environment. This environment includes the current state of the LTSs representing the behavior of the logical controllers and plant elements, and the current runtime model (R_k).

$R_k = (CI_k, DI_k)$ for $k = 0 \ldots n$, represents the k^{th} runtime MGridML instance pair, the control instance (CI_k) and data instance (DI_k) currently being executed by the synthesis process. These models are causally connected to the microgrid plant configuration.

S_{k+1} represents the updated environment after applying the updates to LTSs based on the changes in the models or events from the plant.

O represents the output alphabet including the microgrid energy management control scripts to be executed by the middleware, and an updated runtime model R_{k+1} for the user interface.

3.2.2 Model of Execution

In this section we provide details on the separation of the domain-specific knowledge (DSK) from the model of execution (MoE). Unlike the previous work by Wu et al. [83] in the user-centric communication domain where there is tight coupling between the DSK and MoE in the synthesis process, we separate the DSK from the MoE when defining the dynamic semantics for the synthesis process. Figure 3.6, a refinement of the high-level diagram shown in Figure 3.5, shows the top-level view of the MoE for the synthesis process using a UML state machine. Although there are some

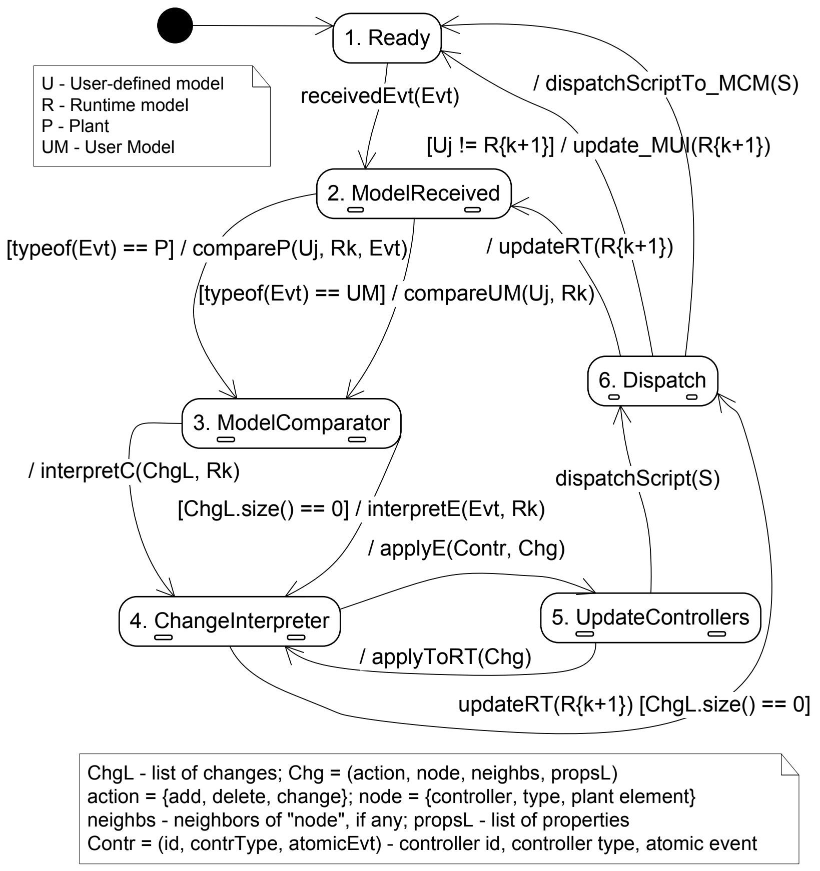

1. Ready
U - User-defined model
R - Runtime model
P - Plant
UM - User Model
receivedEvt(Evt)
/ dispatchScriptTo_MCM(S)
[Uj != R{k+1}] / update_MUI(R{k+1})
2. ModelReceived
[typeof(Evt) == P] / compareP(Uj, Rk, Evt)
/ updateRT(R{k+1})
[typeof(Evt) == UM] / compareUM(Uj, Rk)
6. Dispatch
3. ModelComparator
/ interpretC(ChgL, Rk)
dispatchScript(S)
[ChgL.size() == 0] / interpretE(Evt, Rk)
/ applyE(Contr, Chg)
4. ChangeInterpreter
5. UpdateControllers
/ applyToRT(Chg)
updateRT(R{k+1}) [ChgL.size() == 0]
ChgL - list of changes; Chg = (action, node, neighbs, propsL)
action = {add, delete, change}; node = {controller, type, plant element}
neighbs - neighbors of "node", if any; propsL - list of properties
Contr = (id, contrType, atomicEvt) - controller id, controller type, atomic event

Figure 3.6: High-level state machine of the synthesis process.

submachines that depend on DSK, the main behavior associated with the DSK is captured in submachine 5 *UpdateControllers*, which will be explained in the following text.

After the system is initialized it stays in the *Ready* state until an event (*Evt*) is received. This event may be the arrival of a new user-defined MGridML model (U_j) or an event signifying there is a state change in the plant. Based on the type of event that was received the appropriate method is called in submachine 3 (*ModelComparator*). The task of submachine 3 is the generation of a list of model changes (*ChgL*) to be interpreted by submachine 4 (*ChangeInterpreter*). Each change (*Chg*) in the list is a 4-tuple of the form (*action*, *node*, *neighbs*, *propsL*), where *action* = {*add*, *delete*, *change*}; *node* is the node in the model that changes; *neighbs* - neighbors of *node* in the model, if any; *propsL* - list of changed controller or plant element properties.

Based on the change (*Chg*) being processed, R_k, and the states of the controllers, the appropriate event is identified and applied to a LTS for a given controller. The function *applyE* is called and a transition is made to submachine 5 (*UpdateControllers*) where the appropriate controllers are updated. Assuming the appropriate controllers are successfully updated then function *applyToRT* is called with the change, and R_k is updated to reflect the current runtime state. After all the changes are interpreted and the current runtime model is updated to give the new runtime model (R_{k+1}) the function *updateRT* is invoked. Submachine 6 (*Dispatch*) sends control scripts to the middleware and R_{k+1} to the user interface, and replaces R_k with R_{k+1}.

Figure 3.7 shows the state machine for submachine 5 (*UpdateControllers*) in Figure 3.6 to update the microgrid controllers, the domain-specific behavior for the microgrid domain. Submachine 5.2 (*update*) receives an atomic event contained in *Contr* and the model change (*Chg*) to be applied to a controller. If the type of controller does not exist then the submachine for the controller is created using a fork e.g., *PCC* and *LoadController*, otherwise the event *contrl.atomicEvt* is applied to the controller. After applying the event to the controller submachine, the script generated is passed back to the top-level state machine shown in Figure 3.6. If the change is successfully applied to the controller then the *applyToRT* function is invoked to update the current runtime model. To improve readability we do not show the submachines for *GroupController*, *SourceController* and *StorageController* in the figure since their events are similar to the controllers shown.

Table 3.1 shows the LTS for a load controller, see submachine 5.4 in Figure 3.7. The columns in the table represent the transition number, source state, target state, event, guard and action. The addition of a load controller consist of two transitions (1 and 2). Transition 1 moves from the *Ready* state to the *AddC* state, the event to

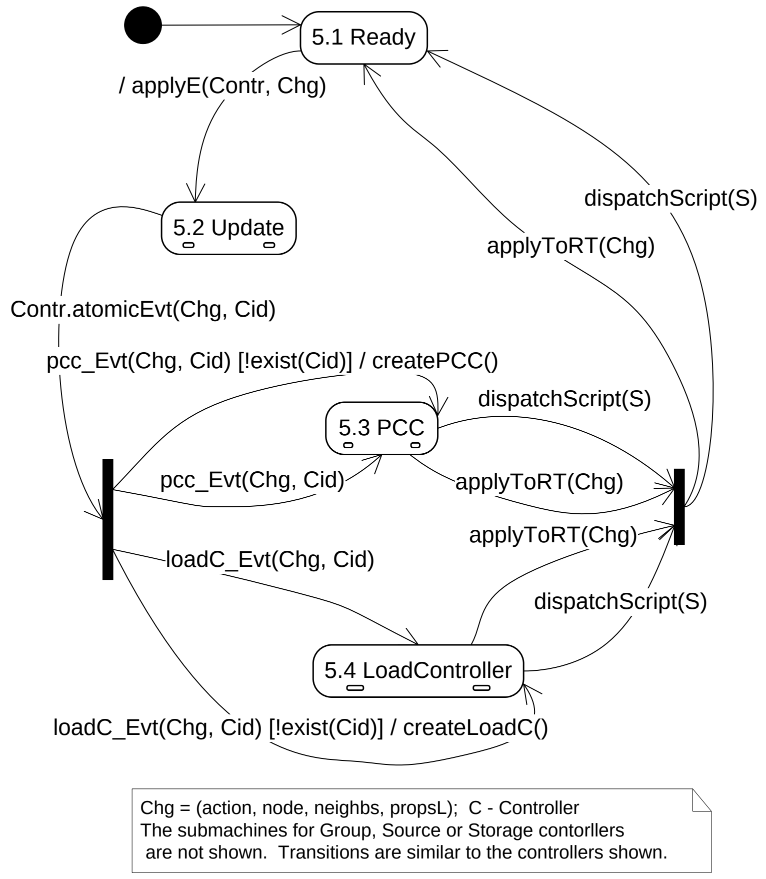

5.1 Ready
/ applyE(Contr, Chg)
5.2 Update
dispatchScript(S)
applyToRT(Chg)
Contr.atomicEvt(Chg, Cid)
pcc_Evt(Chg, Cid) [!exist(Cid)] / createPCC()
dispatchScript(S)
5.3 PCC
pcc_Evt(Chg, Cid)
applyToRT(Chg)
applyToRT(Chg)
loadC_Evt(Chg, Cid)
dispatchScript(S)
5.4 LoadController
loadC_Evt(Chg, Cid) [!exist(Cid)] / createLoadC()
Chg = (action, node, neighbs, propsL); C - Controller
The submachines for Group, Source or Storage contorllers
are not shown. Transitions are similar to the controllers shown.

Figure 3.7: State machine to update controllers.

trigger this transition is *addC* with a parameter *Chg*, the guard checks that the load controller does not exist, and the action generates the add load controller command for the control script. The parameter of the event is a change object (*Chg*) previously described. Transition 2 moves from the *AddC* state to the *Ready* state after the event *addedC* is received with parameter *LC* the id of the load controller being added. Transitions 6 and 13 set the properties for the load controller and device, respectively.

Appendix B shows the list of possible control scripts for the MGridVM that can be generated during the synthesis process dispatched to the middleware. The control scripts are represented here using EBNF-like notation, where "|" symbolizes "or" and

Table 3.1: State machine for a LoadController (Submachine 5.4 in Figure 3.7).

T.	*Source*	*Target*	*Event*	*Guard*	*Action*
Controllers:					
0	**Initial**	Ready		!exist(Chg.node)	create()
1	Ready	AddC	addC(Chg)		gen_addLC_cmd(Chg.node)
2	AddC	Ready	addedC(LC)		applyToRT(Chg)
3	Ready	AddT	addT(Chg)	!exist(Chg.node) && exist(Chg.neighbs)	gen_addLDT_cmd(Chg.node)
4	AddT	Ready	addedT(LDT)		applyToRT(Chg)
5	Ready	Ready	removeT(Chg)	exist(Chg.node) && $\forall$LD typeof(LD) != Chg.node	gen_removeLDT_cmd(Chg.node) applyToRT(Chg)
6	Ready	Ready	setLCProp(Chg)	exist(Chg.node)	gen_setLCProp_cmd(Chg.node,Chg.propsL) applyChange(Chg)
7	Ready	RemC	removeC(Chg)	exist(Chg.node) && $\forall$LDT LDT $\notin$ Chg.neighbs	gen_removeLC_cmd(Chg.node)
8	RemC	Ready	removedC(LC)	!exist(LC) && # LCs > 0	applyToRT(Chg)
9	RemC	**Final**	removed(LC)	!exist(LC)	
Plant Elements:					
10	Ready	AddPE	addPE(Chg)	!exist(Chg.node) && exist(typeof (Chg.node))	gen_addLD_cmd(Chg.node)
11	AddPE	Ready	addedPE(LD)		applyToRT(Chg)
12	Ready	Ready	removePE(Chg)	exist(Chg.node)	gen_removeLD_cmd(Chg.node) applyToRT(Chg)
13	Ready	Ready	setPEProp(Chg)	exist(Chg.node)	gen_setPEProp_cmd(Chg.node,Chg.propsL) applyToRT(Chg)
14	Ready	Ready	notOp(LD)	LD.exist()	Chg $\leftarrow$ (remove, LD, null, null) applyToRT(Chg)
LC - Load Controller; LDT - Load Device Type; LD - Load Device; PE - Plant Element; notOP - not operational; Chg = (action, node, neighbs, propsL) is a model change, where action = {add, delete, change}; neighbs - neighbors nodes in model; propsL - list of (attr, value) pairs					

"{}" zero or more occurrences. Rule 1 states that a control script consists of one or more commands and Rule 2 list the commands. Rule 4 shows the *addController-GroupCmd* consisting of `addControllerGroup` keyword, and attributes *contGroupID* - id of the controller group and a list of one or more controller ids (*controllerID*).

3.2.3 Model Comparison

The user model (U$_j$) and the runtime model (R$_K$) are similar to an attributed graph $G = (N, E, L)$ where N is the set of nodes, $E = N \times N$ a relation representing the set of edges, and $L : N \to A$, a labeling function mapping nodes to attributes [24]. There are a number of basic graph transformations that can be used to change one graph into another. The basic graph transformations we are interested in during model analysis include: *node addition*, *node deletion*, and *attribute change*. Recall that a change to an MGridML model is captured as the tuple (*action*, *node*, *neighbs*, *propsL*), the *action* captures the basic graph transformation e.g., {*add*, *delete*, *change*} where *node* - the unique id of the node; *neighbs* - neighbors of *node* in the model, if any; *propsL* - list of changed controller or plant element properties represented as (*attr*, *value*) pairs. Two of the changes generated when comparing the data instance models in Figure 3.3(b) and 3.4(b) are: (1) `(delete, LD001, {MDG001}, null)` - represents the removal of the node for the heater device, shown on the left of Figure 3.3(b); and (2) `(change, SM001, {MDG001}, {<tariff, 0.15>, <usage, 44.5>})` represents a change in the properties for the smartmeter, where the properties `tariff` and `usage` are updated with the new values, see Figure 3.4(b) bottom right.

Algorithm 3.1 shows the two functions (`compareUM` and `compareP`) for the submachine *ModelComparator* on lines 7 and 17, respectively. Function `compareUM` computes the differences between the control instances for the user model ($U_j.CI$) and the runtime model ($R_k.CI$), shown on line 10, by invoking the function *modelDiff*. A similar approach is used to compute the differences between the data instances $U_j.DI$ and $R_k.DI$, shown on line 11. The *modelDiff* function takes two models and returns the list of changes as described in the previous paragraph. The changes for the control and data instances are stored in the variable *ChgL*. The change list and R$_k$ are passed as parameters when the function *interpretC* is invoked in submachine *ChangeInterpreter*. Function `compareP` checks to see if a plant element in the user-defined model

Algorithm 3.1 *ModelComparator* submachine 3 in Figure 3.6

```
1: modelComparator
   /*ChgL - list of model changes for CI's and DI's. Each change is a tuple of the from
     (action, node, neighbs, propsL) where action = {add, delete, change},
     neighbs - neighbors of "node" in model, propsL - list of changed properties*/
2: compareUM ($U_j$, $R_k$)
3: /* Input: $U_j$ = ($CI_j$, $DI_j$) user model; $R_k$ = ($CI_k$, $DI_k$) runtime model */
4: ChgL.CI ← modelDiff($U_j$.CI, $R_k$.CI)
5: ChgL.DI ← modelDiff($U_j$.DI, $R_k$.DI)    /* If there is at least one CI or DI change */
6: if ChgL.size() > 0 then
7:    interpretChange.interpretC(ChgL, $R_k$)
8: end if
9: EndFunction
10: compareP ($U_j$, $R_k$, Evt )
11: /*Input: $U_j$, $R_k$ same as for analyzeUM; Evt is a plant event */
12: if typeof(Evt) = OPER && Evt.PE ∈ $U_j$.DI then
13:    /* Plant element (PE) becomes operational and is in $U_j$ but not in $R_k$ by default */
14:    compareUM($U_j$, $R_k$)
15: else
16:    interpretChange.interpretE(Evt, $R_k$)
17: end if
18: EndFunction
```

becomes operational. If this event occurs then the function `compareUM` is called with the user model and the current runtime model, otherwise the *interpretE* function is invoked in submachine *ChangeInterpreter*.

3.2.4 Change Interpretation

Algorithm 3.2 defines the functions used in *ChangeInterpreter* submachine in Figure 3.6, including `interpretC`, `interpretE` and `applyToRT`. The four state variables used in the algorithm include: (a) *ChgEvtMap* which is a static table that maps a node type, model change action and controller state to a controller event e.g., $(nodeType, action, state, event)$; (b) *CurrContrStates* a table maintained at runtime with current controllers states, e.g., $(id, state)$; (c) R_{k+1} updated runtime model; and (d) *CurrChgL* the list of model changes that is being updated. An example of an entry in the *ChgEvtMap* table would be (`LoadController, add, Ready, addC`) that is for a load controller type if the model change action is `add` and the current state of the controller is `Ready` then the atomic event returned would be `addC`.

Algorithm 3.2 *ChangeInterpreter* submachine in Figure 3.6

```
1: changeInterpreter
   /* State variables: ChgEvtMap - a static table that maps a node type, model change
     action and controller state to a controller event e.g., (nodeType, action, state, event).
     CurrContrStates - dynamic table with current controllers states, e.g., (id, state).
     $R_{k+1}$ - Updated runtime model; CurrChgL - current changes not handled */
2: interpretC (ChgL, $R_k$ )
3: $R_{k+1} \leftarrow R_k$
4: CurrChgL ← ChgL
5: /* All CI changes are applied before DI changes */
6: for all Chg ∈ CurrChgL.CI do
7:    id ← Chg.node.getContrID()
8:    state ← CurrContrStates.getState(id)
9:    atomicEvt ← ChgEvtMap.getEvt(typeof(Chg.node), Chg.action, state)
10:   Contr ← (id, contrType, atomicEvt)
11:   /* Contr - contains the id of the target controller, the type of controller and the
12:   event for the controller submachine */
13:   UpdateControllers.applyE(Contr, Chg)
14: end for
15: for all Chg ∈ CurrChgL.DI do
16:   id ← Chg.node.getPE_ID()
17:   state ← CurrContrStates.getState(id)
18:   atomicEvt ← ChgEvtMap.getEvt(typeof(Chg.node), Chg.action, state)
19:   Contr ← (id, contrType, atomicEvt)
20:   UpdateControllers.applyE(Contr, Chg)
21: end for
22: EndFunction
23: interpretE (Evt, $R_k$) /* Evt - is generated from the plant */
24: Contr ← (Evt.id, Evt.contrType, Evt.atomicEvt)
25: UpdateControllers.applyE(Contr, null)
26: EndFunction
27: applyToRTChg
28: /* Chg - is a 4 -tuple (action, node, neighbs, propsL) */
29: $R_{k+1} \leftarrow$ applyChange(Chg, $R_{k+1}$) /*$R_{k+1}$ is updated with the change node */
30: CurrChgL.remove(Chg) /*remove change from current change list */
31: if CurrChgL.size() == 0 or timeout() then
32:   Dispatch.updateRT($R_{k+1}$)
33: end if
34: EndFunction
```

The function `interpretC`, line 2, is non-blocking and takes as input the change list `ChngL` and R_k. The first part of the function, lines 6 to 14, iterates through each CI change (*Chg*) and generates the controller atomic event by searching *ChgEvtMap* with the node type, change action and current state of the controller. The atomic event is combined with the controller id and type to create the *Contr* object. The *applyE* function in the *UpdateControllers* submachine is invoked with parameters *Contr* and

R_k. After all the CI changes are processed the DI changes are processed using a similar approach. By non-blocking we mean that after the function call *applyE* function on line 13 *interpretC* continues.

The `interpretE` function, line 23, takes as input an event generated by the plant (`Evt`) and current runtime model ($\texttt{R}_k$), creates the *Contr* object using the fields of the plant event, and calls the *applyE* function in the *UpdateControllers* submachine. The *applyE* function takes as parameters the *Contr* object and the *null* value since there is no change resulting from the processing of a plant event.

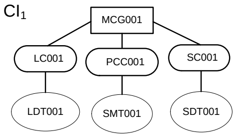
The `applyToRT` function, line 27, takes as input a change that has been successfully processed by the *UpdateControllers* submachine, updates R_{k+1} with the change, removes the change from the current list of changes, and calls the *updateRT* function in the *Dispatch* submachine. Note that if all the changes are not removed from the list of changes and a timeout fires then the *updateRT* function is also invoked.

3.2.5 Synthesis of the Illustrative Scenario

The table in Figures 3.8 illustrate the execution trace of the synthesis process for the change season scenario. The columns in the table from left to right are R_k - the runtime model, U_j - the user model, the changes generated by comparing the R_k and U_j models, the events for the controllers, and the control scripts generated. Due to space limitations the changes, events and control scripts in the tables are abbreviated. The Winter and Spring configurations in the scenario are demarcated using double lines.

The first row of the table in Figure 3.8 shows the comparison between models $R_0.CI$ ($\texttt{CI}_0$ shown on the left of the first row) and $U_1.CI$ ($\texttt{CI}_1$) resulting in a change list containing seven *add* nodes. The change list shown in the table is abbreviated due to space restrictions. This change list results in several events being generated, the first being `initialMCG` which results in the control script `initializeMGrid(‘‘MCG001’’)`

Runtime model = R_k	User Model = U_j	Changes	Events	Control Script Generated

CI_0
null
	CI_1  MCG001
	LC001
	PCC001
	SC001
	LDT001
	SMT001
	SDT001
		added nodes(mgrid controller (MCG001), load controller (LC001), pcc controller (PCC001), …, source data type (SDT001))	initialMCG
			createLC
			createPCC
			createSC
			addLC
			addPCC
			addSC
				initializeMGrid("MCG001")
				addLoadController("LC001",..)
				addPCCController("PCC001",..)
				addStorageController("LC001", ..)

			LC_added PCC_added SC_added	

			addLDT
			addSMT
			addSDT
				addLoadDeviceType("LDT001", …,"LC001")
				addMeterType("SMT001", …, "PCC001")
				addStorageDeviceType("LC001", …,"SC001")

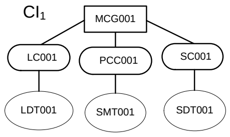
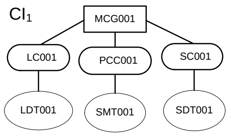
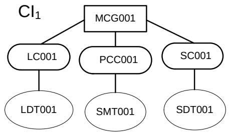
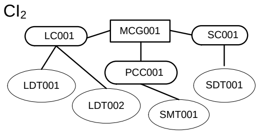
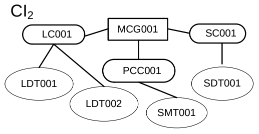
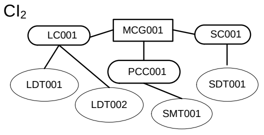
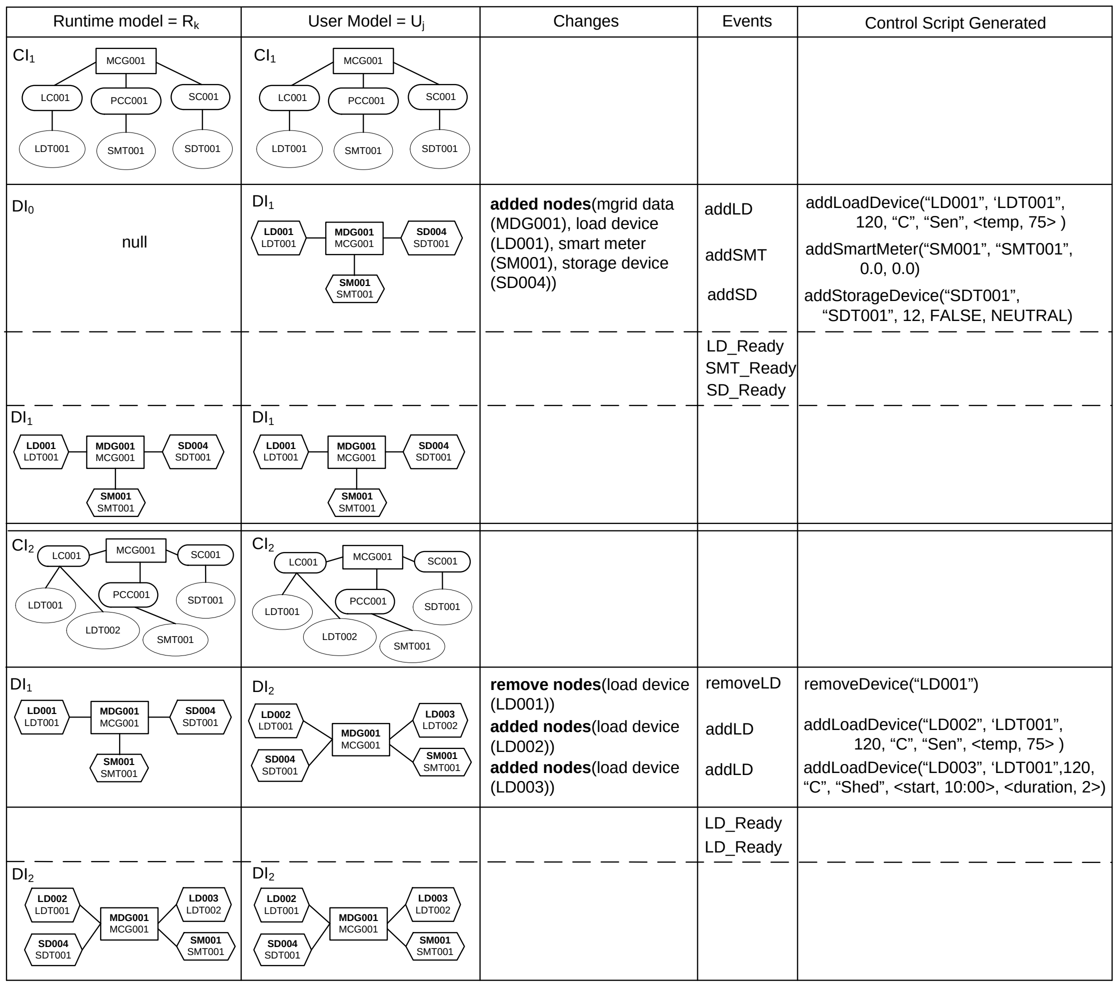
			LDT_added PCC_added SC_added	
CI_1  MCG001 LC001 PCC001 SC001 LDT001 SMT001 SDT001	CI_1  MCG001 LC001 PCC001 SC001 LDT001 SMT001 SDT001			
CI_1  MCG001 LC001 PCC001 SC001 LDT001 SMT001 SDT001	CI_2  LC001 MCG001 SC001 PCC001 LDT001 SDT001 LDT002 SMT001	**added nodes**(load data type (LDT002))	addLDT	addLoadDeviceType("LDT002", …,"LC001")
			LDT_added	
CI_2  LC001 MCG001 SC001 PCC001 LDT001 SDT001 LDT002 SMT001	CI_2  LC001 MCG001 SC001 PCC001 LDT001 SDT001 LDT002 SMT001			

Figure 3.8: Execution trace of the change season scenario for the control instances.

being generated. The details of the change generated for the addition of the load controller node (*LC001*) is as follows: action = *add*; node = *controller* with the attributes *(<LoadController, LC001, ControllerSeasonal, 1-n, ANY, 110, 120, FALSE, LDT001>)*; neighbors = *<MCG001, LDT001>*; and properties = *null*, since there are no properties to change. After the LTS for load controller *LC001* is created, see Table 3.1, it moves into the *Ready* state. The action (*add*) in the change results in the event `addLC` being generated (*addC(Chg)* in Table 3.1) which results in the control script `addLoadController(‘‘LC001’’, ...)` being created (a result of *gen_addLC_cmd(Chg.node)* in Table 3.1 being invoked) and sent to the middleware for processing, rightmost column of the first row in Figure 3.8.

The table shows the other events and the corresponding controls scripts that were generated, including the scripts to add the other controllers. The second row of the table show the events received from the middleware stating that the respective logical controllers were added. The event `LC_added` generated from the middleware is in response to the load controller *LC001* being added. The third row shows the events generated to add the types associated with the controllers that were previously added. For example, the event `addLDT` corresponds to adding the load device type *LDT001*, third transition of Table 3.1. The fourth row shows the events after adding the respective types for the various controllers, for example, the event `LDT_added` signifies that the middleware has added the device type *LDT001* to the controller *LC001.* The fifth row shows that the runtime model and the user model, with labels $\mathtt{CI}_1$, has reached stasis. The second part of the table, below the double lines, shows the steps associated with the changes from the Winter control instance to the Spring control instance, which involves adding the load data type *LDT002.*

The table in Figure 3.9 shows the data instance models associated with the change season scenario. The first and fifth rows in the table shows the control instances used to process the data instances for the Winter ($\mathtt{CI}_1$) and Spring ($\mathtt{CI}_2$) seasons, respectively. The second row of the table shows the comparison between the initial or null runtime model, $R_0.DI$ ($\mathtt{DI}_0$), and the initial user model $U_1.DI$ ($\mathtt{DI}_1$), resulting in four nodes being added. These nodes include mgrid data (*MDG001*), load device (*LD001*), smart meter (*SM001*), and storage device (*SD004*). The data in the mgrid data entity is used to map the devices to a specific mgrid controller, in this case *MCG001.* A similar approach, previously described for the control instances, is used to process the changes for the data instance. For example, to add the load device *LD001* the event *addPE(Chg)* (shown as `addLD` in the table of Figure 3.9) is generated, see transition



Runtime model = R_k	User Model = U_j	Changes	Events	Control Script Generated
CI_1	CI_1			
DI_0 null	DI_1	**added nodes**(mgrid data (MDG001), load device (LD001), smart meter (SM001), storage device (SD004))	addLD addSMT addSD	addLoadDevice("LD001", 'LDT001", 120, "C", "Sen", <temp, 75>) addSmartMeter("SM001", "SMT001", 0.0, 0.0) addStorageDevice("SDT001", "SDT001", 12, FALSE, NEUTRAL)
			LD_Ready SMT_Ready SD_Ready	
DI_1	DI_1			
CI_2	CI_2			
DI_1	DI_2	**remove nodes**(load device (LD001)) **added nodes**(load device (LD002)) **added nodes**(load device (LD003))	removeLD addLD addLD	removeDevice("LD001") addLoadDevice("LD002", 'LDT001", 120, "C", "Sen", <temp, 75>) addLoadDevice("LD003", 'LDT001",120, "C", "Shed", <start, 10:00>, <duration, 2>)
			LD_Ready LD_Ready	
DI_2	DI_2			

Figure 3.9: Execution trace of the change season scenario data instance.

10 in Table 3.1, and the control script `addLoadDevice(...)` is generated. The parameters for the `addLoadDevice` control script are: device id = `LD001`, device type = `LDT001`, wattage = `120`, control = `CONTROLLBALE`, criticality = `SENSITIVE`, and the property consist of attribute = `temperature`, value = `75`. After the devices are added the middleware generates the events `LD_Ready`, `SMT_Ready` and `SD_Ready` indicating that the runtime model can be updated to reflect the new state of the plant.

The second section of the table in Figure 3.9, demarcated by the double lines, shows the changes required for the Spring season associated with the data instances. One node is removed (load device *LD001*) and two nodes were added (load device

LD002 - A/C and load device *LD003* the pool). The events and control scripts follow a similar pattern to the description provided in the previous paragraph.

The second row has only one entry which shows a successful reply from the middleware signifying that the controllers were added. Subsequently in the third row show the scripts generated to request the addition of the device types. Once the middleware responds via events that the types were added, *updateRT* modifies the RT model to that represented in row 5.

With the submission of the Spring data instance we see the removal of *LD001* and the addition of *LD002* and *LD003*. The changes evoke the submission of one renoveDevice and two addLoadDevice Control scripts accordingly by the change interpreter. After the middleware responds with the acknowledgment of the device additions then updateRT does the appropriate modification to the runtime model and stasis is achieved.

3.3 MGridVM Prototype

To demonstrate the efficacy and applicability of this approach, we describe our prototype which creates and realizes MGridML models using the DSVM architecture consisting of four layers representing different levels of abstraction. The high-level design is similar to the design described by Deng et al. [17]. This section describes the implemented prototype consisting of the virtual machine (software) and the hardware testbed. We also provide a more detailed description of the synthesis engine, which is the focus of this work.

3.3.1 High-Level Design

Figure 3.10 shows the four layered architecture for the MGridVM, based on the architecture created by Deng et al. [17] for the Communication Virtual Machine (CVM). The layer of the MGridVM are described as follows:

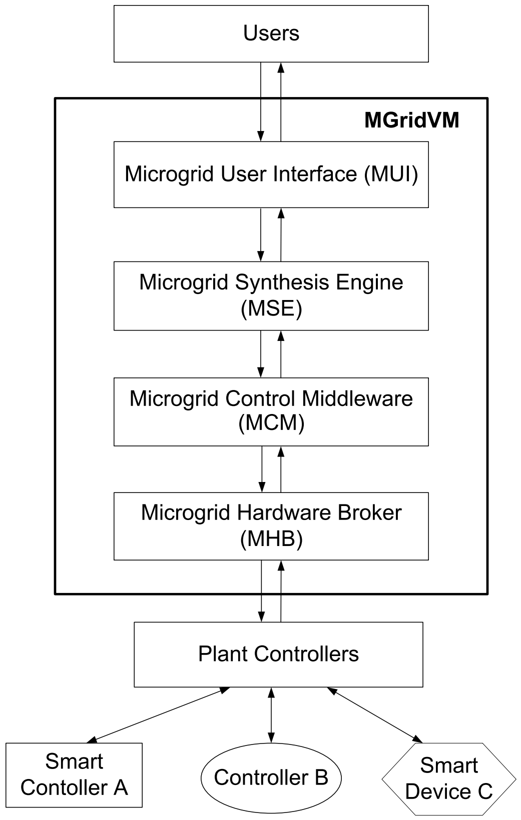

Users
MGridVM
Microgrid User Interface (MUI)
Microgrid Synthesis Engine (MSE)
Microgrid Control Middleware (MCM)
Microgrid Hardware Broker (MHB)
Plant Controllers
Smart Contoller A
Controller B
Smart Device C

Figure 3.10: MGridVM high-level architecture.

Microgrid User Interface (MUI): Provides the user with the ability to specify MGridML models to represent the configuration and behavior of the underlying physical plant. Facilitation is at a level of abstraction as to be intuitive yet expressive enough to describe most of the configurations and functionality required of the microgrid. The MUI provides two distinct types of modeling environments, one for the novice user (possibly a building occupant) and one for a more technical user (domain expert).

Microgrid Synthesis Engine (MSE): Implements the synthesis process described in Section 3.2. The MSE takes as input a MGridML model and transforms it to one or more control scripts. We provide more details of the design in the following section.

Microgrid Control Middleware (MCM): Transforms the platform independent control script into API calls to be handled by the broker layer. It is at this layer that the virtual machine may amalgamate diverse resources with distinct commands. The scripts are interpreted and an appropriate intent model is generated and executed. The services managed by the middleware includes mapping physical devices to logical controllers based on types, performing type checking, executing energy management algorithms, and enforcing policies for various device configurations. The current prototype implementation directly invokes API calls of the underlying broker layer without performing any of the logical mapping or type checking. The MCM implementation is being conducted concurrently within our research group. [57].

Microgrid Hardware Broker (MHB): This layer is responsible for issuing atomic commands to the plant and monitoring device states. Commands are issued to the plant via physical controllers and listens for an appropriate task completion response. The MHB also observes the plant for activities such as variations in power at different points in the microgrid. The current prototype implementation uses low voltage testbed, however we have have had successes at the 120V level using Zigbee control [47]. To simulate users manually operating devices, as would be expected in a real world scenario, the MHB has an independent user interface capable of provisional overriding upper level settings.

Testbed: The MGridVM prototype was tested on a low voltage direct current (DC) hardware testbed which simulated critical aspects of the microgrid. Figure 3.11 shows the hardware testbed, which consists of the following plant elements: *loads* - fan (top center of the figure) and lights (below the fan); *sources* - a photo-voltaic cell (top right of the figure); *storage* - small battery pack (top left of the figure); and *utility* - 1000mA adapted power source (bottom center of the figure). The small battery pack and photo-voltaic cell are used to simulate distributed storage and distributed generation functionality respectively. The storage is connected to a charge controller which allows for controlled charging and discharging via the relay bank.

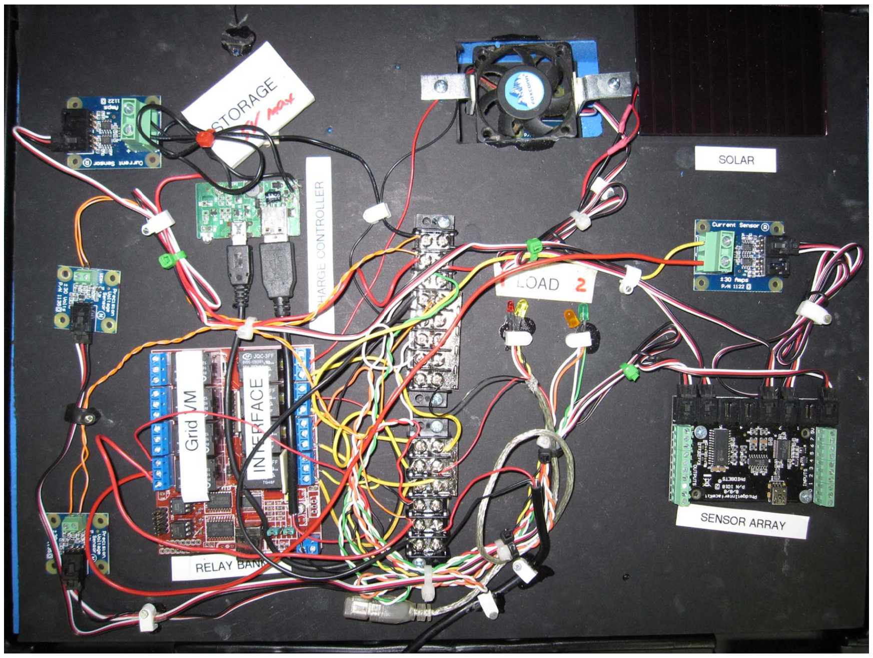
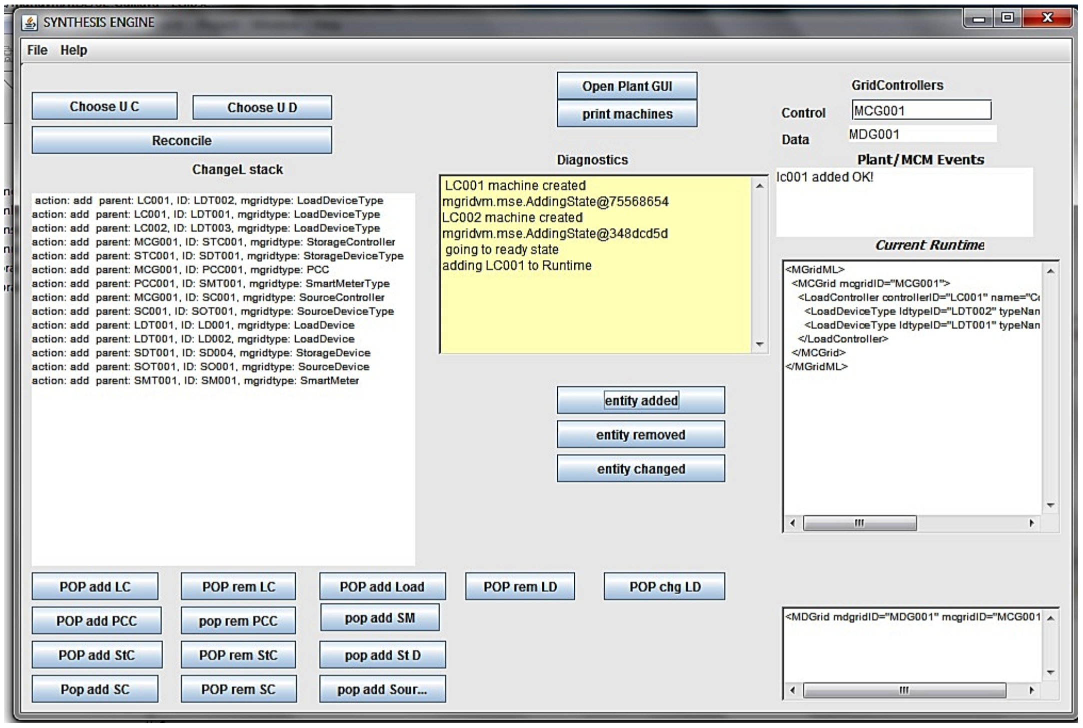


Figure 3.11: Hardware testbed for the MGridVM protoype.

The MHB communicates serially with the testbed via two USB interfaces (bottom left of the figure). The first USB interface, used for monitoring, connects to an interface which continuously monitors analog voltage and current sensors. These interfaces, placed at specific points on the testbed, converts the values to digital metering values. The sensory interface (bottom right of the figure) features configurable sampling rates and noise cancellation to reduce false readings. The second USB interface is connected to a relay bank and actuates component switching.

Figure 3.12 shows the GUI screen used to test the MSE and create models to execute in the evaluation of the MSE. The buttons on the top left are used to select the user control instance, select the user data instance, and compare the user and runtime models. The text window on the left shows the change list stack, i.e., the list of changes to be processed, the buttons below the text window allow specific changes to popped form the stack and be processed. The text window in the center echos actions taken by the MSE, the buttons below this window simulates various changes to the model independent of the change list. The three text windows on the right



```
action: add  parent: LC001, ID: LDT002, mgridtype: LoadDeviceType
action: add  parent: LC001, ID: LDT001, mgridtype: LoadDeviceType
action: add  parent: LC002, ID: LDT003, mgridtype: LoadDeviceType
action: add  parent: MCG001, ID: STC001, mgridtype: StorageController
action: add  parent: STC001, ID: SDT001, mgridtype: StorageDeviceType
action: add  parent: MCG001, ID: PCC001, mgridtype: PCC
action: add  parent: PCC001, ID: SMT001, mgridtype: SmartMeterType
action: add  parent: MCG001, ID: SC001, mgridtype: SourceController
action: add  parent: SC001, ID: SOT001, mgridtype: SourceDeviceType
action: add  parent: LDT001, ID: LD001, mgridtype: LoadDevice
action: add  parent: LDT001, ID: LD002, mgridtype: LoadDevice
action: add  parent: SDT001, ID: SD004, mgridtype: StorageDevice
action: add  parent: SOT001, ID: SO001, mgridtype: SourceDevice
action: add  parent: SMT001, ID: SM001, mgridtype: SmartMeter
```

```
LC001 machine created
mgridvm.mse.AddingState@75568654
LC002 machine created
mgridvm.mse.AddingState@348dcd5d
 going to ready state
adding LC001 to Runtime
```

Control: MCG001
Data: MDG001

```
lc001 added OK!
```

```
<MGridML>
  <MCGrid mcgridID="MCG001">
    <LoadController controllerID="LC001" name="C
      <LoadDeviceType ldtypeID="LDT002" typeNan
      <LoadDeviceType ldtypeID="LDT001" typeNan
    </LoadController>
  </MCGrid>
</MGridML>
```

```
<MDGrid mdgridID="MDG001" mcgridID="MCG001
```

Figure 3.12: GUI for the synthesis engine testbed.

show the MCM events, the current runtime model for the control instance and the current runtime model for the data instance.

3.3.2 Synthesis Engine Design

The goal of the MSE design is to separate the entities with the domain-specific knowledge (DSK) from the model of execution (MoE). The main packages for the MSE are shown in Figure 3.13. The packages reflect the structure of the synthesis process outlined in Figure 3.5. The classes associated with controlling and managing the synthesis process are in the package `mse::controller`. The classes for the model comparator, that implements the algorithm *ModelComparator*, and the change interpreter, that implements the algorithm *ChangeInterpreter* are contained in the package `mse::modelProcessor`. The model processor package uses the functionality provided by the Eclipse Modeling Framework[71], specifically EMF Core and EMF

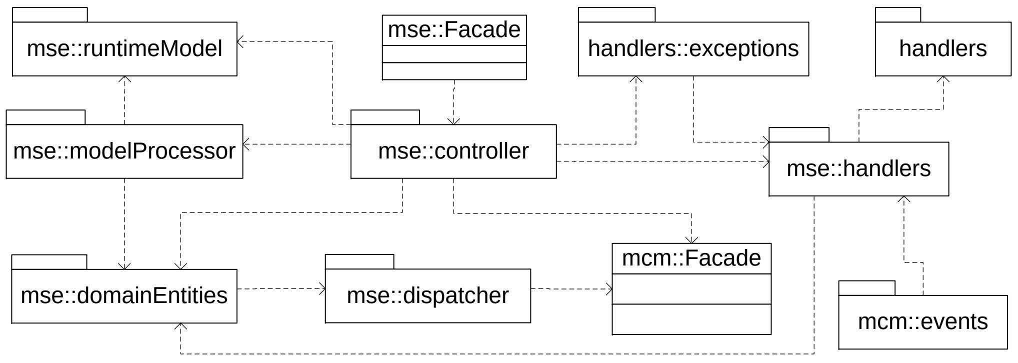


Figure 3.13: Design of the microgrid synthesis engine (MSE) showing the main packages.

Compare. The package `mse::domainEntities` contains the packages and classes with DSK and will be described in more detail later. Classes in the `mse::controller` and `mse::modelProcessor` need to know about the classes in the `mse::domainEntities` package, hence the dependency between them.

The class `mse::Facade` exposes an API to the MUI which accepts the user-defined model, and the class `mcm::Facade` exposes an API to the MSE that accepts the control scripts. The packages on the right side of Figure 3.13 contain the classes to handle events and exceptions from the MCM, and generate events for the MUI, one such event is to update the user MGridML model in the MUI. The package `mse::runtimeModel` package contains the classes to represent the runtime model in the MSE, including the attributes and operation required to update the runtime model.

Figure 3.14 shows the packages contained in the `mse::domainEntities` package, that is, the packages containing the DSK entities. There are two classes in the figure `DomainManager` - coordinates the activities of the packages, and the `ControllerManager` that coordinates the creation and updates to the LTSs for each type of controller, and handles the control scripts created by the controllers. The controllers depend on the control scripts package from the MCM, `mcm::controlScripts`. The `mgridEvents`

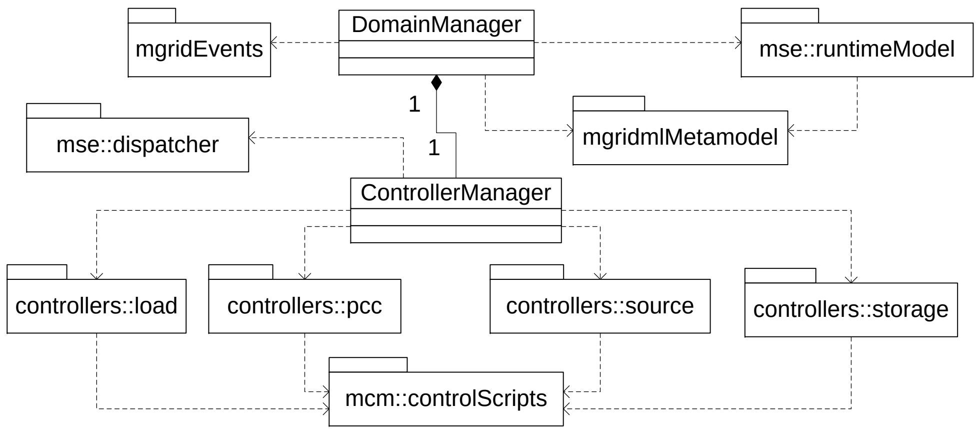


Figure 3.14: Design of the domain entities package showing the main packages.

package contains the classes that represent the table for the change event map (see Algorithm 3.2) and the events generated for the LTSs. The `mgridmlMetamodel` package contains the classes that define the metamodel for MGrdiML. The metamodel is used by the EMF compare package [71] when comparing MgridML models.

3.4 Synthesis Prototype Evaluation

The focus of this investigation is to evaluate the efficiency of the microgrid synthesis engine (MSE) with respect to the model processing required during synthesis. The aspects of the model processing we will investigate are related to model comparison and change interpretation. The specific objectives of the study are as follows:

- Objective 1: Determine how performance times change relative to the increase in the size of the MGrdiML models to be processed during synthesis.
- Objective 2: Compare the performance of the approach used when the domain-specific knowledge (DSK) is separated from the model of execution (MoE) in MGridVM versus the tightly coupled approach used in the CVM, the first DSVM prototype.

3.4.1 Experiment Method

Experimental Setup: The MSE was isolated by substituting the MUI layer with a test driver which generated MGridML models, and the MCM layer was replaced with a stub which collected the control scripts for inspection and generated the appropriate events for the MSE. The MSE and harness was placed on a laboratory computer with the following specification: CPU - Intel Core 2 Duo operating at 2.00GHz; RAM - 3 Gigabytes DDR2 Memory Bus Speed 2 X 233MHz; and the operating system was Windows 7 Ultimate.

Experimental Design: To evaluate the MSE for Objective 1 we generated several X-MGridML model instances of varying complexity to be processed by the MSE. We use this approach to set the MSE to contend with increasing model changes which would result in changes in the runtime model, list of model changes, and updates to the LTSs, accordingly. We varied M which is the number of model nodes (where M = 3 to 57) and N is the number of controllers (where N = 1 to 28). The controllers in each scenario are load controllers, each connected to one load device type. The initial number of nodes are 3 which includes the MGrid controller node, a load controller and a load device type. Our focus for Objective 1 was to evaluate the performance for the following: parsing the model, model comparison, change interpretation, and updates to the runtime model. For each of the scenarios we compared the user MGridML model with the *Null* model, that is, the initial model in the MSE prior to receiving a user model.

To evaluate Objective 2 we used the data collected for Objective 1 along with static metrics to investigate the differences within the MGridVM and CVM synthesis engines. We inspected both prototypes and ascertained that the results are candidates for comparison as: (1) the inputs to both SEs are models and events of comparable magnitude. The model generation automation was specifically designed to create test models with content and relationship criteria which allows for negligible performance differences by the model comparison component; (2) the outputs of both SEs are control scripts; and (3) the hardware platform are very similar. By isolating the SE of both prototypes with drivers and stubs, we posit that the difference in functionality is nominal.

Data Collection: To collect our data regarding execution time analysis for MSE we employed the Eclipse Testing and Performance Tooling Platform (TPTP) [85] by instrumenting the MSE code. We collected runtime data for the main classes in the MSE: *Parser*, *ModelInterpreter*, *ChangeInterpreter*, which are part of the `mse::modelProcessor` package; and *RuntimeUpdate* part of the `mse::runtimeModel` package, see Figure 3.13. The microgrid scenarios, represented as MGridML models, range in size from 3 to 57 nodes (1 to 28 controllers). We ran each synthesis scenario 12 times and to reduce the impact of anomalies we discarded outliers (the highest and lowest) and averaged the remaining 10 readings.

Several of the metrics used in the study were obtained from the experiments reported in Wu et al. [83]. For those experiments Eclipse TPTP was also used as the tool of choice to obtain the runtime results. For this experiment set, each scenario was executed 12 times and averaged after the highest and lowest values were discarded to reduce the influence of anomalies. The data for the SE in the CVM reported by Wu et al. [83] identified the number of participants in each communication scenario, however we present the data in terms of the number of nodes in the CML models. Each new participant requires at least 4 additional nodes, *device*, *mediumtype*, *isAttahced* and *participant*, as show in Figure 5 in Wu et al. [83]. Since we did not have an exact match for the number of model nodes for the MGridML and CVM models, we chose the scenarios for MGridML with the closest number of nodes for the experiments.

Prototype Development : Prior to addressing the results of the experiments we will present an overview of the developmental time of the prototypes in order to establish a basis for our comparative study and as a springboard to extrapolate effort savings

Table 3.2: Developmental Times for Prototype Synthesis Engine Components

CVM(coupled)	MGRIDVM(coupled)	MGRIDVM(decoupled)	DSK for CVM
130 hrs	155 hrs	73 hrs	25 hrs

through reuse.

The development of the prototypes was accomplished in discrete stages. Table 3.2 shows the hours expended within each stage. The first prototype , CVM (coupled) was effected in 130 hours. Duplicating this methodology and applying it to the microgrid via MGRIDVM (coupled) required 155 programming hours. This shows a noticeable increase in development time as the gains in expertise was overshadowed by the time taken to outline and capture the execution semantics of the possible interactions between model elements within this new domain. From that time it took 73 hours to decouple the DSK from the MOE; we have recorded 35 of these hours as dedicated to the MGRIDVM DSK. This decoupled version is the prototype being presented within this dissertation. In the final column we show 25 hours extended to build the standalone DSK of the CVM capable of reinstantiation.

We reiterate that there was a increase in expertise throughout these stages which should be considered when analyzing this data.

3.4.2 Results

The results of our performance evaluation for Objective 1 is shown in Table 3.3. The main classes analyzed, shown across the top of the table, are *Parser*, *ModelInterpreter*, *ChangeInterpreter* and *RuntimeUpdate*. The first column of the table shows the number of controllers defined in each model (N) and the second column the total number of nodes in each model (M). The Columns 3 to 6 show the execution times in seconds for the classes previously mentioned, and Column 7 shows the approximate total execution times for model synthesis for the various scenarios. For example, the

Table 3.3: Execution Times for Primary Classes in MSE. N - Number of controllers and M - number of model nodes.

Classes:		*Parser*	*Model Compare*	*Change Interpreter*	*Runtime Update*	Total
N	*M*	Execution Times (in secs)				
1	3	0.19	0.33	0.05	0.01	0.58
4	9	0.42	0.37	0.07	0.03	0.89
7	15	0.82	0.44	0.08	0.03	1.38
10	21	1.15	0.63	0.13	0.07	1.98
13	27	1.49	0.77	0.15	0.10	2.51
16	33	1.68	0.95	0.17	0.11	2.91
19	39	2.23	1.44	0.20	0.13	4.00
22	45	2.49	1.83	0.22	0.13	4.67
25	51	3.17	2.20	0.29	0.16	5.82
28	57	3.48	2.38	0.32	0.18	6.35

first row in the table shows the scenario with 1 controller, 3 model nodes, takes 0.19 seconds to parse, 0.33 seconds for model comparison, 0.05 seconds for change interpretation, 0.01 seconds to update the runtime model, and a total execution time of 0.58 seconds.

The results reported in Table 3.3 are for those scenarios with the controllers shown in the first column of the table. It should be noted that although the number of controllers represent an arithmetic sequence between 1 and 28 with initial value of 1 and common difference of 3, we actually recorded results for 28 scenarios each for controllers from 1 to 28. Figure 3.15 shows the graphical representation of the data in Table 3.3. The figure shows that the most expensive part of the synthesis process is the parsing of the xml file representing the X-MgridML model. The time required for parsing appears to show linear growth with respect to the number of nodes in the model. The next most expensive part of the process is model comparison, with also appears to show linear growth. For the given scenarios the change interpretation and runtime model update seems to be negligible when compared to parsing and model comparison.

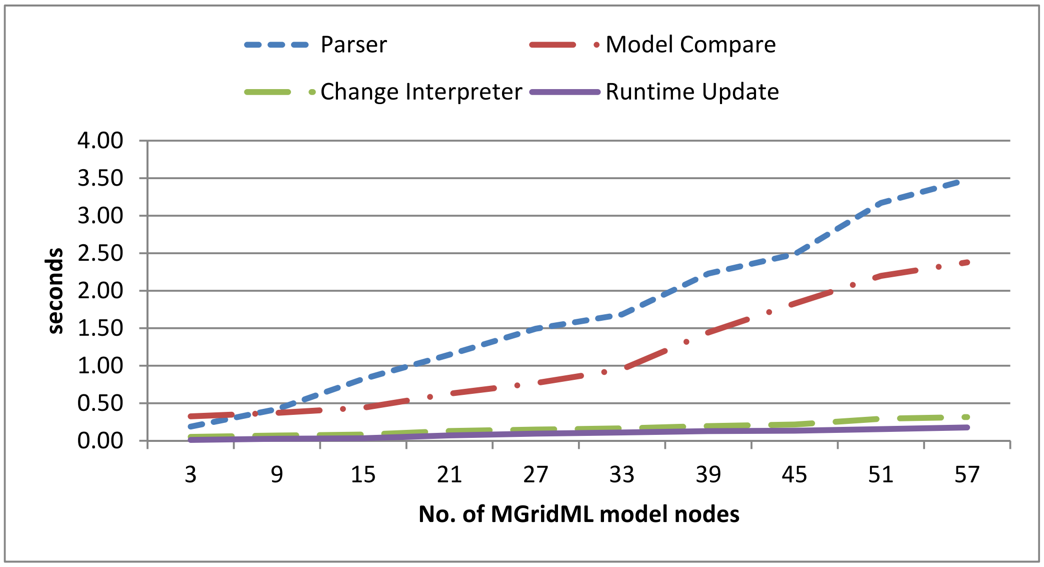


Figure 3.15: Evaluation of the synthesis for the main classes in MSE.

Table 3.4: Comparison of Static Metrics of Prototype Synthesis Engines

Metric	MGRIDVM	CVM
SLOC	2913	963
# Classes	31	22
# Methods	434	156

In evaluating Objective 2 of the study we first show several static metrics for MGridVM and CVM then show the execution times comparing the synthesis of several scenarios in both the microgrid energy management and user-centric communication. Table 5.1 shows the source line of code (SLOC), number of classes, and number of metrics for the synthesis engines in MGridVM and CVM. The synthesis engine in the MGridVM is larger in size when compared to the synthesis engine in the CVM. This is the result of more classes to represent the LTSs for the controllers in the microgrid, and the classes used to separate the DSK from the MoE. For example, the model comparison extends several classes in the Eclipse Modeling Framework [71], providing a more generic model comparator.

Table 3.5: Comparison of Execution Times for CVM and MGridVM Synthesis Engines.

CVM SE			MGridVM SE	
N_{Parts}	M	*Exec Time* (secs)	M	*Exec Time (secs)* (*Extrapolated*)
2	10	1.17	10	0.89+(1.38-0.89)/6*1 = 0.97
3	14	1.80	14	0.89+(1.38-0.89)/6*5 = 1.30
4	18	2.07	18	1.38+(1.98-1.38)/6*3 = 1.68
5	22	2.42	22	1.98+(2.51-1.98)/6*1 = 2.07
6	26	2.76	26	1.98+(2.51-1.98)/6*5 = 2.42
7	30	3.17	30	2.51+(2.91-2.51)/6*3 = 2.71

Table 3.5 shows the comparison of the execution times for the CVM and MGridVM synthesis engines for several scenarios from the respective domains. The table is divided into two sections the left section contains the data for CVM (as reported in Wu et al. [83]) and the right section data for the MGridVM. The 3 columns in the left section represents the number of participants in a communication (N_{Parts}), the number of nodes in the respective CML model (M), and the execution time in seconds to perform model synthesis for each scenario. The right section shows the number of nodes in an assumed MGridML model and the extrapolated execution times for the assumed model. The values in the rightmost column of Table 3.5 are extrapolated from the values shown in Table 3.3.

Figure 3.16 shows the graphical representation of the execution times in Table 3.5. The figure shows that execution times for both the CVM and MGridVM have a similar slope, which appears to be linear. Overall the execution times for models with a similar number of nodes show that the CVM's synthesis engine time is on average 17.1% more than that of the MGridVM's synthesis engine. We explore these result further in the next section.

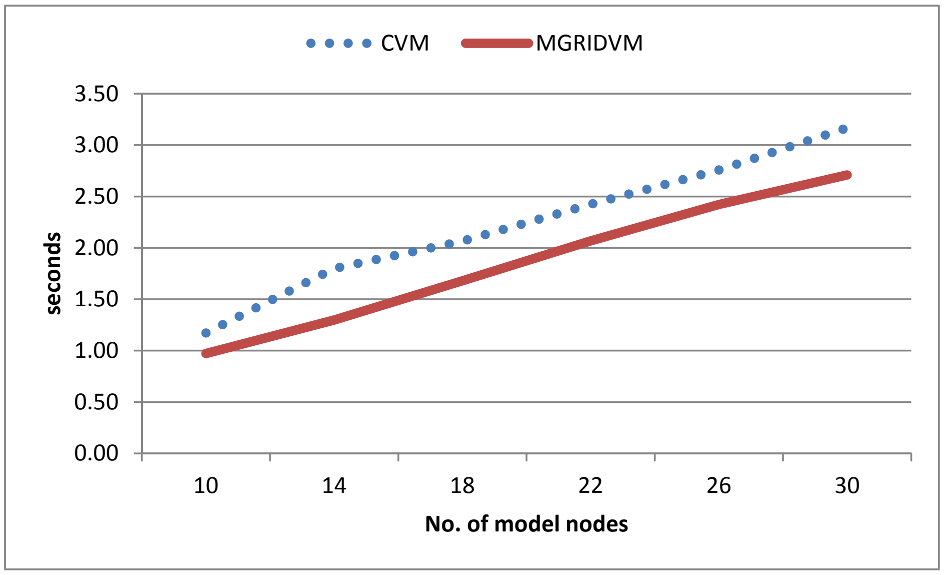


Figure 3.16: Execution metrics of MGridVM and CVM synthesis engines

3.4.3 Discussion

The results for Objective 1, Table 3.3 and Figure 3.15, shows that as the number of nodes in the MGridML models increase the execution times for the synthesis process increases linearly. The most expensive aspect of model synthesis is in the parsing of the X-MGridML representation of the MGridML model. Parsing of the model is approximately 27% greater than model comparison, 88% greater than change interpretation and 95% greater than the update of the runtime model. Performing regression analysis on the total execution times using the data in Columns 2 (M) and 7 (Time) of Table 3.3, produces a linear equation of the form $y = 0.107x - 0.138$ with an R square value of 0.96 which shows a strong relationship between the number of nodes and the execution times. It is worth noting that the slope on the regression line is 0.107 which shows that as the number of nodes increases it is expected that the execution times will increase at a fairly slow rate.

The comparison of model synthesis in the MGridVM and CVM serves two purposes: (1) to evaluate model synthesis in two domains, one in a distributed environment and the other in a centralized environment; and (2) to determine if there are any significant changes in the execution time when the DSK is separated from the MoE. Based on the static metrics shown in Table 5.1 there is an increase in the code written for model synthesis for the MGridVM versus the CVM. The CVM has two main LTSs (negotiation and media transfer) while there are four LTSs for the controllers in the microgrid (load, storage, source, point-of-common coupling) which accounts for an increase in the number of classes. In addition, the MGridVM uses a more generic method of comparing model which extends classes from the Eclipse Modeling Framework (EMF), EMF Core and EMF Compare [71], [74].

Table 3.5 and Figure 3.16 show that there is a strong correlation between the execution times from MGridVM and CVM. The regression analysis of the CVM time (Column 5 and 6 in Table 3.5) produces a linear equation of the form $y = 0.094x + 0.532$ with an R square value of 0.98 which also shows a strong relationship between the number of nodes and the execution times. The slope on the regression line for the CVM is 0.094 which shows an even slower rate of increase of the execution times as the number of nodes increase. The MGridVM faster growth rate could be the result of separating the DSK from the MoE, however the growth rate for the MGridVM is still relatively small. We can conclude that model synthesis is feasible for domains where the i-DSML models are below 60 nodes, base on our evaluation. In the future we expect to perform additional experiments on larger models to be more affirmative regrading the models that can be process in the synthesis engines for DSVMs.

Threats to validity: We consider both internal and external validity threats [81] for the experiments that were performed for Objectives 1 and 2. An external threat is related to the setup of the experiments related to Objective 1. We made the assumption that a sampling size of maximum 10 was sufficient to ascertain whether increasing model complexity would result in unpredictable execution coefficients in any of the primary algorithms of the MSE. Another external threat involved the model synthesis that was performed. We only used control instances and not data instances during model synthesis, the assumption was that manipulating control instance required more processing, i.e., the creation and deletion of the LTSs, which may not reflect the true times for general scenarios. The final external threat for Objective 1 was not using very large models, i.e., models containing hundreds of nodes, this may result in a faster growth of execution times for model synthesis, which could impact the feasibility for model synthesis for some domains.

Some of the external threats for Objective 2 are similar to that of Objective 1, specifically, that of selecting a sample size that may be too small to result in a generalization of model synthesis for the two domains being considered. Another external threat was not using the same hardware setup for the CVM experiments. The results were obtained from past experiments performed by Wu et al. [83]. No specifics were given on the hardware specification in Wu et al. [83]. An external threat that affected both experiments was the use of Eclipse TPTP [85] to instrument the code since the parameters used to tuned the instrumentation were not consistent with the instrumentation for the MGridVM and CVM synthesis engines. The main internal threat for both objectives was the automatic generation of models to be analyzed. The data in Tables 3.3 and 3.5 show that the number of nodes in the MGridML and CML models are part to a sequence, showing that the models are not randomly generated. By automatically generating the models the independent variable is not truly random resulting in some biased in the experiments.

3.5 Chapter Summary

In this chapter, we presented the methodology utilized in the development of the microgrid modeling language and described an approach to synthesizing its models. The loose coupling of the DSK and the MoE paves the way for its ultimate separation to enable the efficient instantiation of the model synthesis component (synthesis engine) for DSVMs in other domains.

The efficacy of our approach was evaluated by developing a prototype of the DSVM, MGridVM. The evaluation of this primary prototype demonstrated that our approach to realizing behavior in the synthesis engine does not contribute to deteriorated operation times when compared to previously published results for an earlier DSVM in another domain, CVM.

CHAPTER 4

SEPARATION AND REPRESENTATION OF THE DSK FOR A SYNTHESIS ENGINE

In this chapter we examine the MoE described in Chapter 3 to extract domain-specific concerns and represent them as persistent data. The residuum is consolidated to be later refined to support the ability to seamlessly replace domain-specific artifacts on demand during instantiation. In other words, the model synthesis process is reevaluated in terms of concrete artifacts.

In one of the definitive works regarding software reuse, Krueger [49] identifies application generation as an essential reuse category. This dissertation's approach is akin in spirit to approaches in application and compiler generation. The familiar notion is for common application logic to be reused. Biggerstaff et.al. [7] proposes four factors which support reuse:

1. *Finding the reusable component.* This involves identifying that portion which is similar in each application.

2. *Understanding the component.* This subtask involves the development of a mental model.

3. *Modifying components.* The modifications involve restructuring, adding to, and refactoring the existing code structure.

4. *Composing the components.* The gluing mechanism for the components identified are critical to the development of an effective solution.

To address the first two items we study the models of execution for the domains. Of primary importance is the development of an abstraction and to identify that portion of the abstraction that is fixed and that which can be modified by its user

[38, 66, 80]. To this end our abstraction is the model of execution, a reusable portion of the application logic will be the fixed portion, and the DSK is that which is subject to modification. In the next section we will illustrate a refined MoE for MGridVM which serves as the aforementioned abstraction.

Biggerstaff's third factor requires us to refactor the MoE. We accomplish this using principles of aspect-oriented design. To effectively address the approach used in separating the concerns we have employed aspect aware refactoring to modularize DSK as a distinct concerns. The act of refactoring restructures object oriented code in a methodical manner which needs to preserve behavior [59, 36]. It is usually the case where refactoring is used to improve understandability and readability, however our motive is modularity; consolidating our DSK concern as an aspect. The refactoring is *aspect aware* [36]. Our task therefore is to identify the join points of the concerns then transform the application with the intent of isolation to aspects. To effectuate the identification our technique involved walkthroughs and code inspections as supported in [32].

We accomplished the refactoring of the MoE described in chapter 3 by restructuring the MoE for MgridVM according to its major components taking care to extract any hard coded knowledge that was domain specific. We ensured that the new model of execution preserved prior behavior.

The next section will present the revised MoE which is the object for further study in the separation process.

4.1 Revisiting the MGridVM Model of Execution

Figure 3.5 shows a high level view of the synthesis process for the MGridVM which has been refactored from that in chapter 3 as to lessen the coupling between concerns. The extent of the refactoring however yields a MoE which remains domain specific

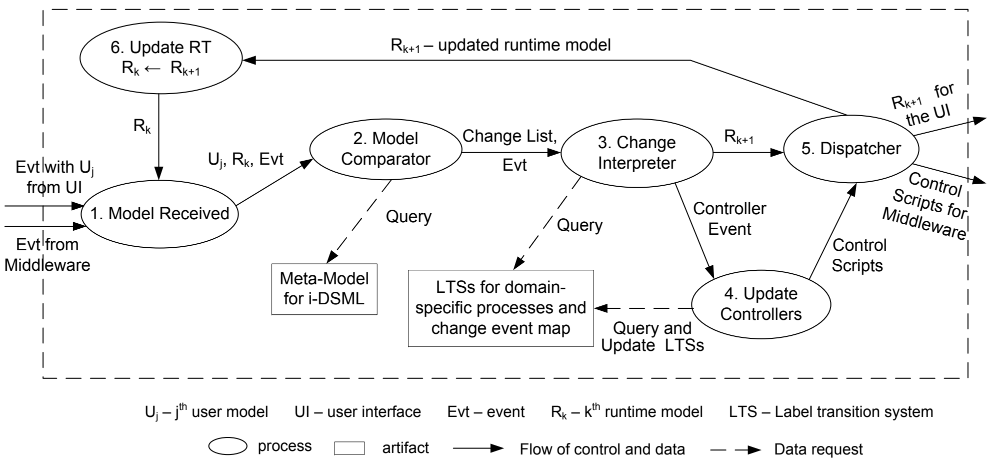


Figure 4.1: High level view of the synthesis process revised to reduce coupling

to microgrid activities. The weaker coupling however is a precursor for the DSK separation.

The inputs, shown on the left of the figure, are either an event (Evt_{UM}) representing the receipt of a user-defined model (U_j) or an event (Evt_P) from the plant generated by the middleware layer of the MGridVM. The outputs from the process are the control scripts to be executed by the middleware and the updated runtime model (R_{k+1}) to be displayed in user interface. Below is a description of the main processes shown in Figure 4.1:

1. *Model Received* - accepts either a user-defined model (U_j) or a middleware event (Evt_P), and the current runtime model (R_k). The U_j, R_k and (Evt) are then passed on to the model comparator.

2. *Model Comparator* - if the event type is Evt_{UM} then U_j and R_k are compared to generate a change list. This is where model differencing occurs and requires access to the metamodel for the i-DSML to ensure we are comparing models of the same type.

3. *Change Interpreter* - is where the change list and Evt_P are processed. Based on a model change, the current state of the LTSs, and the entry in the *change event map*, a controller event is generated. Alternatively, a pending or new event (Evt_P) reflecting an update in the microgrid plant is handled . After all changes and events are handled the updated runtime model (R_{k+1}) is sent to be dispatched. The change event map is a table that maps model changes to event in LTSs based on the current state of the LTS.

4. *Update Controllers* - the states of the appropriate LTSs are queried and updated, and appropriate control scripts are generated and dispatched.

5. *Dispatch* - handles the dispatching of the new runtime model (R_{k+1}) to the user interface and submits the control scripts to the middleware to be executed.

6. *UpdateRT* - updates the current runtime model (R_k) with the new runtime model (R_{k+1}).

The artifacts, shown as rectangles, in Figure 3.5 are where the DSK is stored. These artifacts include the metamodel for the i-DSML, the domain-specific LTSs and a domain-specific change event map.

Within this refined MoE, the major activities have been modularized to six distinct components. A closer inspection of the components will show that the metamodel, LTSs and change mappings can be separated as they were identified and included within the DSK concern. The challenge remaining is component 4 *Update Controllers*. This part of our application logic remains domain specific which we address in the next Chapter.

4.2 Persisting Domain Specific Knowledge

Our overarching goal is not to simply develop standalone persistence for the DSK but to provide constructs and a methodology with sufficient formalism to support semantic specification in concert with i-DSML metamodel development. To achieve the desired modularity we decoupled the DSK by identifying the join points where the common interpreter logic accesses the domain specific concern. The decoupling of the DSK as an aspect structurally translates to Figure 4.2. The DSK is tangled throughout the legacy architectures, comprises mainly of a *syntax* portion (the metamodel) and a *semantic* portion. The syntax is described using the i-DSML metamodel and is accessed by all three of the major components. The semantics will describe the meaning of each of the elements of the language and their interaction. This we capture persistently using Labeled Transition Systems(LTSs) and change mapping.

The *join points* are signified by the contact of dashed line contact with the major processes. Figure 4.2 is a restructuring of the earlier synthesis process presented in Figure 4.1 to allow for the separation of the DSK. The MoE for MGridML requires an *Update Controllers* which is specific to that energy management domain. In order to facilitate alternate domain concerns such as (re)negotiation and media transfer which are found in CML, this process had to me refactored to make it more generic. The new *State Manager* process now assumes the responsibility to update the domain entity's LTS's. Since the runtime models used by the synthesis process is also domain specific, in the spirit of concern consolidation, the *State Manager* has now also assumes the functionality of *UpdateRT*. As we can see in Figure 4.2, the Dispatcher is not affected by the DSK concern (no join points), as as such remains relatively unchanged in functionality.

To define the DSK associated with the language synthesis process, the author is required to define the set of state transition systems for the domain and the change mapping. We have previously used LTSs in our prototype designs, however any similar automata may suffice. Offered as an representative instances, table 4.1 shows a state transition system to accomplish media transfer within CVM. For each state machine

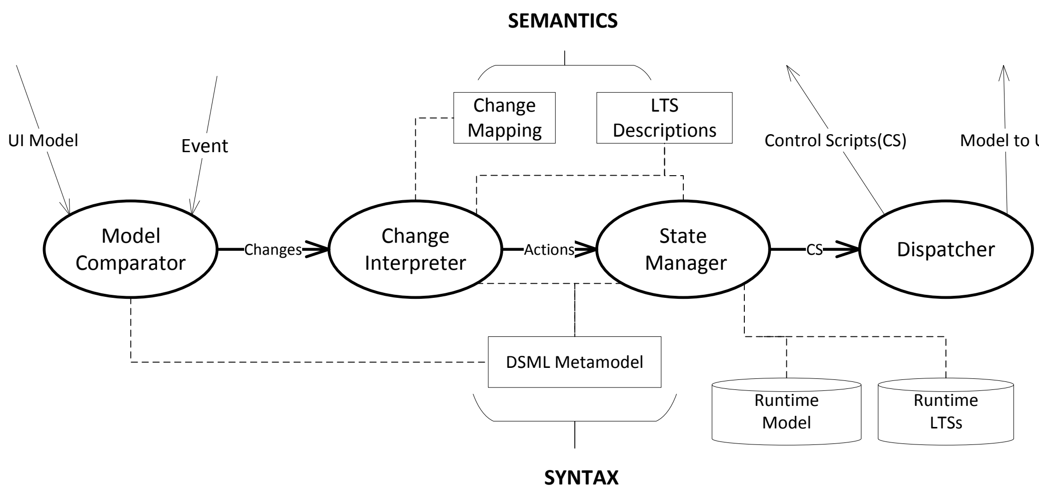


Figure 4.2: Representation of the Separation of DSK from the MoE.

the author is required to provide the source states, target states, events, guards and control scripts.

This concept is made persistent using an XML representation which allows for the following desired features [8, 5]:

- Unambiguous representation and retrieval of information.
- Robustness of format. The tree structure format is highly adaptable to more complex data modeling.
- The XML format is widely accepted and tool support is available.

Appendix D illustrates the persistent representation of the media transfer LTS as XML. Within CVM we need to store the XML state transitions for `Media Transfer` and `(Re)negotiation.`

For MgridVM we need to persistently represent all plant entities. These include: `Load Devices, Storage Devices, Source Devices, Smart meter, Load Controllers, Storage Controllers, Source Controllers` and the `PCC`.

Table 4.1: State transition table for media transfer.

Tr	*Source_State*	*Target_State*	*Event*	*Guard*	*Action*
0	**Initial**	Ready	initiateNeg ‖ intiateInviteNeg		
1	Ready	StreamEnabled	enableStream		genStreamEnable_Script
2	Ready	StreamEnabled	enableStreamRec		genStreamEnableRec_Script UCI.notify(DS_{i+1})
3	StreamEnabled	StreamEnabled	enableStream	!IsStreamEnabled	genStreamEnable_Script
4	StreamEnabled	StreamEnabled	disableStream	IsStreamEnabled && # streams > 1	genStreamDisable_Script
5	StreamEnabled	StreamEnabled	enableStreamRec	!IsStreamEnabled	genStreamEnableRec_Script UCI.notify(DS_{out})
6	StreamEnabled	StreamEnabled	disableStreamRec	IsStreamEnabled && # streams > 1	genStreamDisableRec_Script UCI.notify(DS_{out})
7	StreamEnabled	StreamEnabled	sendNonStream		genNonStreamSend_Script
8	StreamEnabled	StreamEnabled	sendForm		genSendForm_Script
9	StreamEnabled	StreamEnabled	recNonStream		UCI.notify(DS_{out})
10	StreamEnabled	StreamEnabled	recForm		UCI.notify(DS_{out})
11	StreamEnabled	Ready	disableStream	# streams == 1	genCloseStream_Script
12	StreamEnabled	Ready	disableStreamRec	# streams == 1	genCloseStreamRec_Script UCI.notify(DS_{out})
13	Ready	Ready	sendNonStream		genNonStreamSend_Script
14	Ready	Ready	sendForm		genSendForm_Script
15	Ready	Ready	recNonStream		UCI.notify(DS_{out})
16	Ready	Ready	recForm		UCI.notify(DS_{out})
17	Ready	**Final**	terminate		

content...

Another critical component of the DSK is the mapping between changes or change patterns and actions. This mapping is used to raise internal events to be processed by the LTSs. Table 4.1 shows a few sample pattern to action mappings from the microgrid domain. **x**'s are placed to signify points where the actual content does not matter; we are looking for patterns. The first three mappings state that if a change occurs to add or delete a data instance element then raise the corresponding element type event. The fourth mapping states that for all changes simply raise a update event. It is essential to declare this domain specific knowledge outside the GMoE framework to allow for the *change interpreter* to effectively process the list of changes arising from the *model comparator*. The change patterns for CVM is more extensive due to the distributed nature of the domain and can be seen in Appendix I. Now that we have illustrated the GMoE and how the DSK may be persistently represented,

Table 4.2: Partial List of Change Mapping for a MGridVM SE Instance

	Pattern	Action
1	(added,NODETYPE = LoadDevice,x,x)	*addLoadDevice(nodeID)*
2	(added,NODETYPE = SourceDevice,x,x)	*addSourceDevice(nodeID)*
3	(removed, NODETYPE = SourceDevice,x,x)	*removeSourceDevice(nodeID)*
4	(changed, NODETYPE = x,x,x)	*change(nodeID, property)*
n	**...**	**...**

we next present a our metamodel approach to instantiating the synthesis engine by recombining the concerns.

4.3 Chapter Summary

This chapter presented the separation of the DSK and subsequent representation as artifacts. Given the MoE from a earlier proof of concept prototype we applied aspect-oriented software design principles to modularize the architecture to result in a extensible framework and domain specific knowledge in the form of LTSs and tabular representations. The implementation of this framework including the gluing mechanisms for the DSK is presented and evaluated in the following chapter.

CHAPTER 5

DEVELOPMENT AND EVALUATION OF THE GENERIC MODEL OF EXECUTION

In this chapter we generalize that portion of the application logic which remains after separating the DSK and develop the gluing mechanisms. To demonstrate the feasibility of this generalized MoE (GMoE) we propose a metamodeling approach and instantiate synthesis engines for MGridVM and the CVM. The utility of GMoE hinges on (1) its viability to be implemented as a synthesis engine and produce the appropriable control scripts given a metamodel of the i-DSML, and a model of the i-DSML's DSVM, and control and data instances from the domains, and (2) its applicability to reduce developmental effort for developers instantiating i-DSML execution engines.

To instantiate a SE instance we make use of the rich toolset of the Eclipse Modeling Framework (EMF) and the *Kermeta* meta-language to respectively describe our metamodel and to weave in the execution semantics presented in Chapter 4. To accomplish this analysis we (1) detail the successful prototype implementation and discuss design considerations, (2) present a trace of scenarios in CML and in MGridML, and (3) perform a comparative study against earlier prototypes with respect to developmental effort utilizing historical data, and (3) using experimental data evaluate the prototype's architecture in a controlled setting to determine coupling between the framework and the DSK extension. This coupling is contrasted with that of earlier prototypes within the two domains.

5.1 Generalizing Model Synthesis

To delve deeper, Figure 5.1 shows the activities involved in the generalized model synthesis process. The core technology of the GMoE is an adaptable runtime model which is a causal representation of the system under control. The GMoE comprises three primary processes: *Model Comparison*, *Change Interpretation* and *State Management*. The model comparator compares the user preference model with an adaptable runtime model to produce a list of changes. The changes are sent to the change interpreter which raises the appropriate internal events or actions according to the change received. As the name suggests, the state manager updates the runtime state. The output of this component are control scripts and a new runtime model which are dispatched to other layers. We will next detail the GMoE using the activity diagram in figure 5.1 .

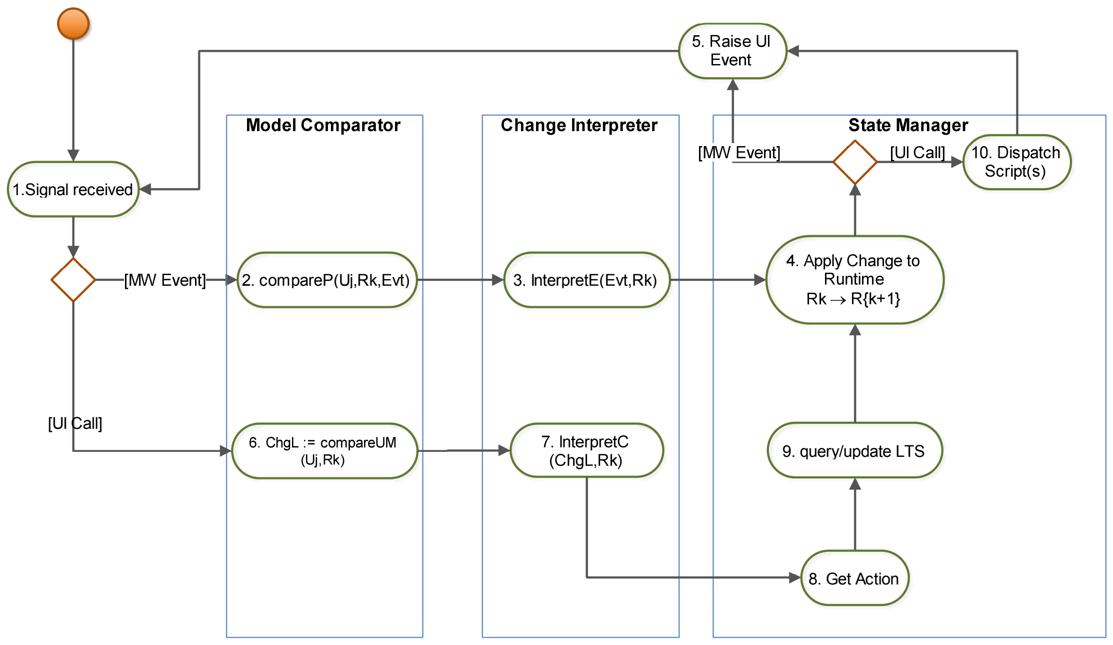


Figure 5.1: Activity Diagram of the GMoE

Our walkthrough of the GMoE process begins when the system awaits a new signal. If the signal is a middleware (MW) event the system progresses to the model comparator *activity 2* where it calls a `compareP` method. The `compareP` method analyses the event to ascertain how the event applies to the runtime model. Within the change interpreter, `interpretE`, inspects the event and recommends the appropriate sequence of atomic changes to the runtime model to the state manager. In the state manager *activity 4* updates the runtime model and *activity 5* informs the UI of any changes made to the environment. Dependent on the type of change, the UI may then trigger a new model comparison.

The second pathway addresses calls from the UI. A UI call is directed to *activity 6*, `compareUM` compares the generates a list of changes, `chgL` based on the difference between user and runtime models. Each change (*Chg*) is a 4-tuple of the form (*action*, *node*, *neighbs*, *propsL*), where *action* = {*add*, *delete*, *change*}; *node* is the node in the model that changes; *neighbs* - neighbors of *node* in the model. The `chgL` is inspected by *activity 7*, *interpretC*. Based on the change (*Chg*) being processed, and R_k, the appropriate event is identified through the change mapping at *activity 8.* The event is applied to the LTSs in *activity 9* to generate control scripts. *Activity 9* is located within the state manager which creates, destroys and updates the LTSs as required. After the changes are interpreted and the current runtime model is transformed to the new runtime model (R_{k+1}) the control scripts generated by the elements' LTSs are sent to the middleware by *activity 10. Activity 5* then updates the user interface and the system is ready for the next signal.
We next present the persistent representation of the DSK.

5.2 The Synthesis Engine MetaModel

A software's architecture captures design decisions which ultimately governs its behavior, quality and structure [72]. We formulated our design decisions based on a propensity towards adaptation; seeking a representation capable of capturing the complexity of the model synthesis approach in two loosely coupled components. The components are a reusable framework to house the GMoE, and an easily specified and interchangeable component to accommodate DSK. Our approach utilizes a model based representation with a fixed structure to instantiate our SE; a model for processing models. The metamodel for the DSVM language forms the basis for constructing a i-DSML synthesis engine from modules and define their interaction utilizing a semantic overlay.

The DSVM SE metamodel is separate and apart from the i-DSML metamodel. Since the GMoE (which includes the Model Comparator, Change Interpreter and State Manager components) is intended for reuse, it is incorporated into the interpreter framework. In this way the GMoE constitutes the framework and the DSK is the framework's extension. Each concern is separated and compartmentalized. Figure 5.2 shows the abstract syntax for the generic SE metamodel. The SE metamodel's central coordinator is aptly named `SE_Manager`. The `State_Manager` is responsible for the runtime representation of the underlying system and serves as a conduit for the GMoE to manage activities of the DSK. It is solely within the `State_Manager` where the *aspects* are joined; this achieves the loose coupling architecture that is at the heart of the refactoring efforts. The gluing of the persistent DSK *semantic* artifacts (the change mappings and LTSs) is accomplished using the `DomainManager`. The `DomainManager`, in addition, is the point of access to the i-DSML metamodel which is applied within `SE_Manager` to relate the non-generic classes to the `ModelComparator`, `ChangeInterpreter` and for runtime updates using the `State_Manager`.

The metaclasses within the GMoE abstract syntax contains methods. These methods represent *semantic actions* required by the framework. These semantic actions are an extension of the core activities which we saw in figure 5.1. A high level algorithm 5.1 shows instantiation process whereby the semantic actions of the GMoE

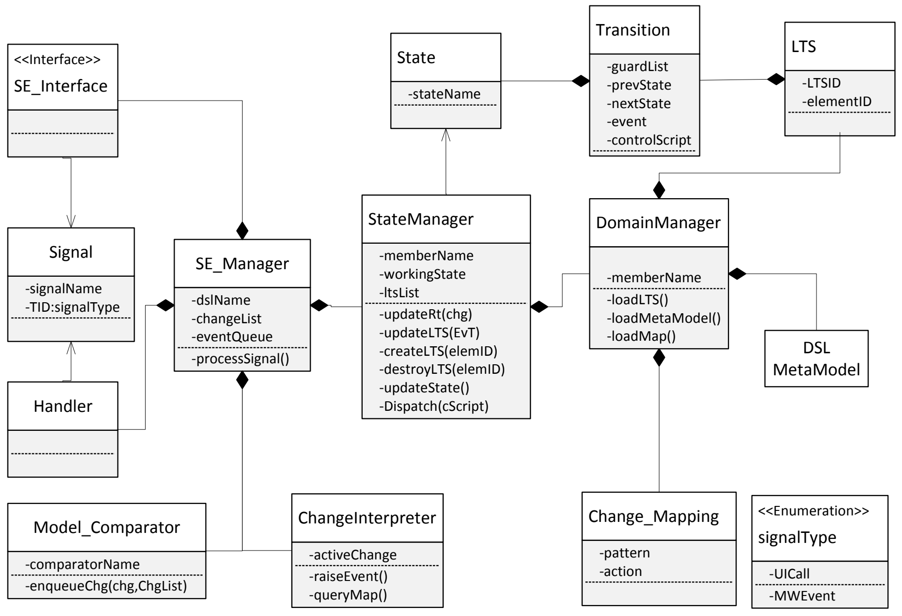


Figure 5.2: Metamodel for Synthesis Engine Definition

and the DSK's persistent artifacts are incorporated. The function `SE_Build` initially ensures that the SE instance submitted complies to the metamodel. Upon validation the instance is serialized to classes and the runtime model are created an set to `null`. The DSK artifacts(LTS and ChangeMap) are next loaded and finally the semantic actions.

Algorithm 5.1 is a representation of how the SE is built using the metamodel presented. Initially the `SE_ metamodel` validates the required `SE_Instance`. Next the instance is serialized to objects. At this point the runtime model initialized to `null`. The DSK concern in terms of its syntax and semantics is subsequently inputted and the build becomes ready for a signal.

5.3 The Instantiation Process

In this section we explain the design and implementation of the Generic Model of Execution (GMoE) prototype. To instantiate a SE model instance(MoE) representing

Algorithm 5.1 *Building the Synthesis Engine*

```
 1: SE_Build
Require: SE_Instance - (Synthesis Engine Model), SE_MetaModel
       // Verify SE Model Instance */
 2: if validModel(SE_Instance, SE_MetaModel) then
 3:    Serialize_Classes(SE_Instance)
 4:    Runtime ← StateManager.runtimeModel
 5:    STATEManager.updateRT(null.)
 6:    //Load DSK concern as persistent artifacts
 7:    ChgMap ← DomainManager.loadMap()
 8:    while $element \in elementList$ do
 9:       LTS ← DomainManager.loadLTS($element$)
10:    end while
11:    i_dsmlMetaModel ← DomainManager.loadMetaModel()
12:    //Finally dynamically bind the implementors of semantic actions
13:    BindToClasses(implementors[ ])
14:    SE_Manager.awaitSignal()
15: end if
```

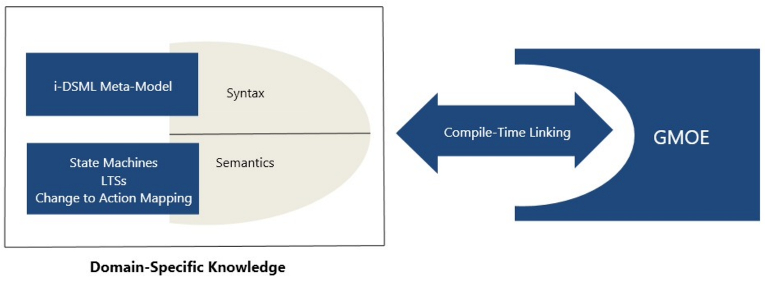


Figure 5.3: Overview of GMoE Approach

the GMoE and the DSK for the i-DSML *(DSL metamodel, LTS's etc.)* we make use of the rich tooling of the Eclipse Modeling Framework (EMF) and the kermeta meta-language to describe our metamodel and to weave in the execution semantics.

5.3.1 Kermeta

Kermeta can be thought of as an aspect oriented programming language and an integrating MDE platform capable of weaving static and dynamic semantics into models. Kermeta uses an action language which employs object-oriented mechanisms and sequential control structures. In fact a MOF metamodel can be looked at as a valid kermeta program complete with classes and attributes but without behavioral aspects. Using Kermeta we extend the SE metamodel to include static semantics, dynamic semantics, and model transformation concerns. Since our metamodel is a static model we weave the execution semantics by declaring classes of similar names equip with operations as kermeta *aspects* . In kermeta, classes with the same qualified name are merged within the interpreters memory therefore attaching dynamism to the structure. The metamodel is now overlaid with kermeta aspects corresponding to each metaclass to describe their execution semantics.

To facilitate the instantiation of the GMoE we developed a driver *SE_launcher*, written in the Kermeta meta-language, to manage the build. Figure 5.4 shows a high level state machine of the *SE_Launcher* which presents the necessary steps to arrive at the instantiated synthesis engine which is the target of our approach. To instantiate the Synthesis Engine the *SE_Launcher* has to first load and register the SE and LTS metamodels in states 1 and 2. These metamodels are generic to any SE instance. State 3 is where our machine takes on its context. The SE model instance allows us to subsequently load the i-DSML metamodel and semantics of the language as Kermeta aspects. Once the GMOE aspects, (Change Interpreter, Model Comparator, etc.) are loaded then we have a recognizable synthesis engine.

Figure 5.5 shows the architectural makeup of the essential components of the generic approach. Note that there is a distinct separation (gluing represented by dashed line) of the GMoE and DSK components demonstrating very loose coupling of the concerns.

5.3.2 Switching SE Instances

To switch between languages the SE_Launcher has to load a new SE model instance which carries the syntactic and semantic elements of the DSVM language. Once the new SE model is loaded then the i-DSML .ecore is registered and the system is ready

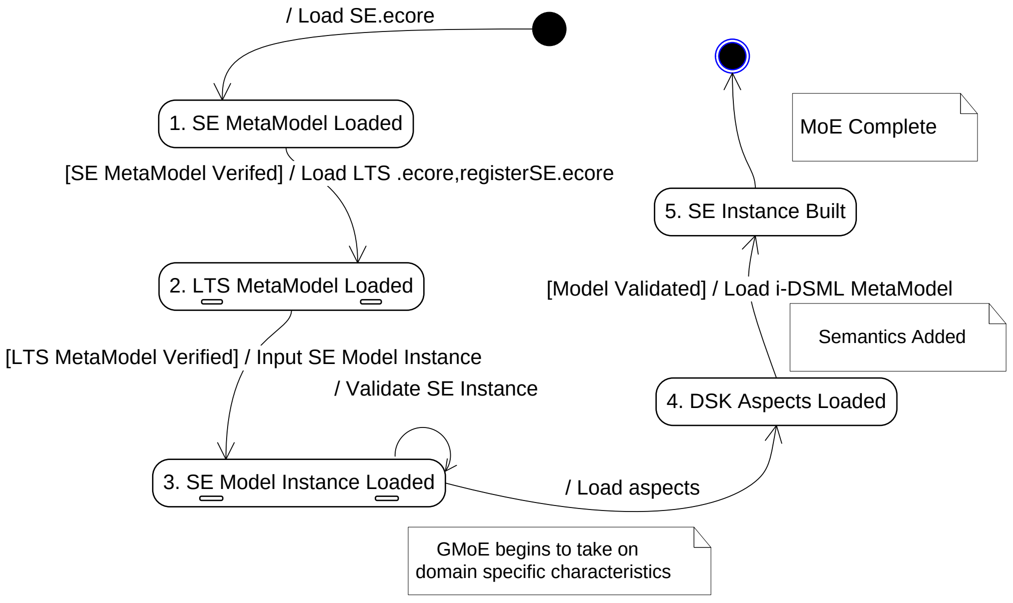

/ Load SE.ecore
1. SE MetaModel Loaded
[SE MetaModel Verifed] / Load LTS .ecore,registerSE.ecore
2. LTS MetaModel Loaded
[LTS MetaModel Verified] / Input SE Model Instance
/ Validate SE Instance
3. SE Model Instance Loaded
/ Load aspects
GMoE begins to take on domain specific characteristics
4. DSK Aspects Loaded
Semantics Added
[Model Validated] / Load i-DSML MetaModel
5. SE Instance Built
MoE Complete

Figure 5.4: State Machine of SE_Launcher implementation resulting in the targeted Synthesis Engine.

to process the i-DSML models. Note that the LTSs are defined in the prototype using its own metamodel. Individual LTSs are created, traversed and destroyed by the domain manager as warranted during runtime.

5.4 The Communication Domain Evaluation Scenarios

To ascertain the utility of our implementation we present indicative scenarios from the communications domain. The scenario choices are from those used to test the earlier CVM prototype so we are able to check known outputs against those of the GMoE prototype. By running the scenarios on earlier prototypes we are capable of determining the validity of the GMoE synthesis.

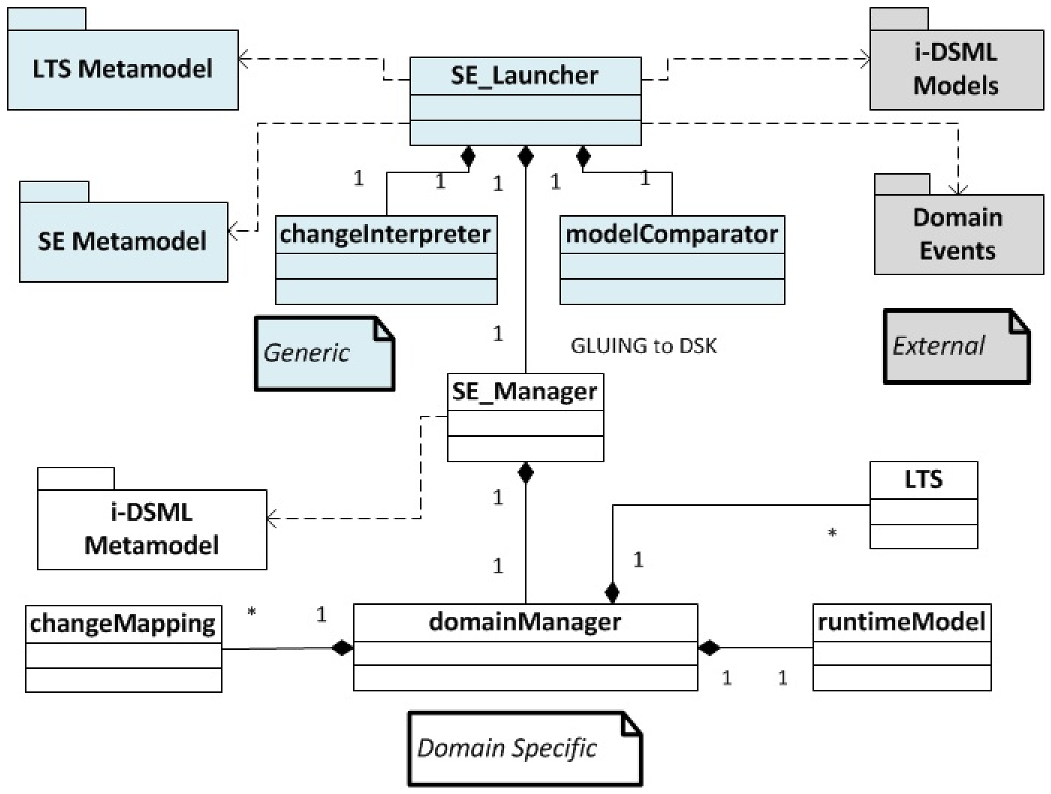


Figure 5.5: Generic SE Architecture

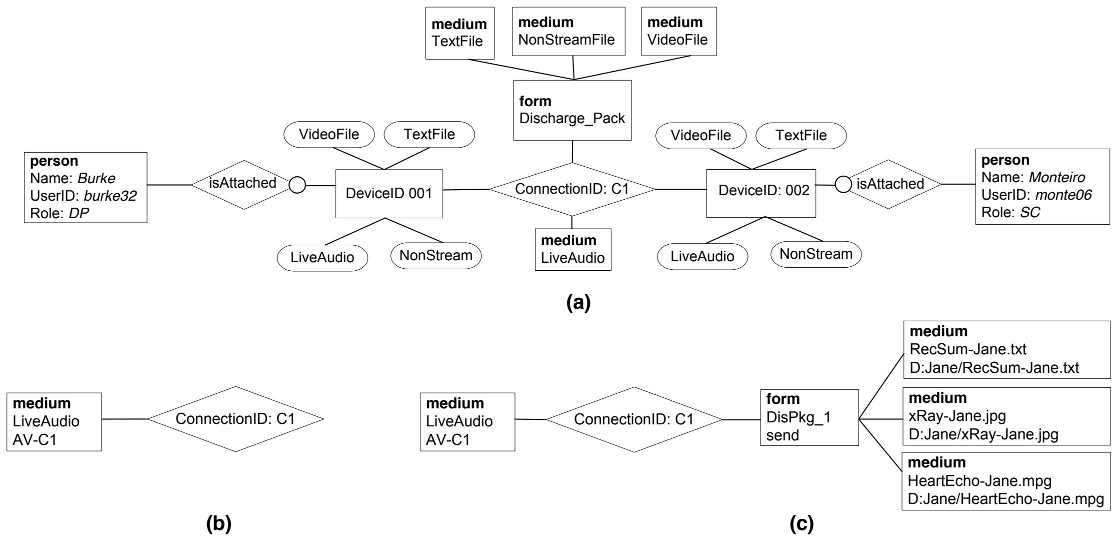


Figure 5.6: Two-Way Communication Scenario

5.4.1 Communications SE Setup

In order to instantiate the CVM Synthesis Engine to process the scenario the GMoE framework SE_Launcher needs to be extended with the DSK from the user-centric communications domain. To accomplish this, there are some preliminary steps that needs to done before the framework can process CML models to produce the targeted control scripts. First the metamodels for the LTSs, the language and synthesis engine,*(LTS.ecore,CML.ecore, SEVM.ecore)* are registered using *EPackages registration*. The LTS and the SEVM metamodels are domain independent. A CVM synthesis engine schema (see Appendix G) CVM_SE.xmi is inputted to the framework. This model describes the artifacts necessary for the framework to function.

5.4.2 SCENARIO 1 - CVM Application

The scenario starts on the day of discharge of Baby Jane (patient). Dr. Burke needs to discuss outpatient care for the patient and initiates an audio video call to Dr. Montiero. During the discussion Dr. Burke needs to clarify some basic points and thinks that sharing some patient records will be required. During the conversation she dynamically creates a record *DisPkg_1* containing the patients record's summary as a text file *JaneRecSum-Jane.txt,* a picture of the latest x-ray, *xRay-Jane.jpg* and video of her electrocardiogram, *HeartEcho-Jane.mpg.* After discussing the details, and being sufficiently satisfied Dr. Burke terminates the call.
Figure 5.6 presents our first scenario. In (a) we see the predefined GCML control schema representing a two-way call in a medical setting. The attributes within the entities that are not required for this walkthrough are not represented in the diagram. In (b) we see the initial data schema used to set up the call and in (c) is the data schema which represents the transfer of the patient files.
At the onset the initial runtime model is null. This may be represented as $(CS_0, DS_0) =$

$(null, null)$

With the submission of the control/data schema pair (CS_1, DS_1) corresponding to Dr.Burke's initial communication request we get a model comparison between null and CS_1. The change List generated is:

```
added (connection(C1),
form("Discharge_Pack"),
device(001),
isAttached(001),
person("burke32")
device(0021),
isAttached(002),
person("monte06")
```

The event raised by the Change interpreter after the query to the change mapping (Appendix I) is:

```
initiateNeg ,localSameCI
```

The events raised leads to the following control scripts upon querrying the negotiation LTS :

```
createConnection("CI")
sendSchema("C1","burke32","monte06",CI_1,DI_0)
```

and subsequently,

```
sendSchema("C1","burke32","monte06",CI_1,DI_0)
```

At this point the runtime model is reconciled to $(CS_1, null)$ and Dr. Burke's control schema becomes the runtime control instance.

To enable audio-visual communication between the parties there needs to be a data instance. When Dr. Burke's data instance (b) , DI_1, is submitted the model comparator generates as change :

```
added (medium(AV_C1))
```

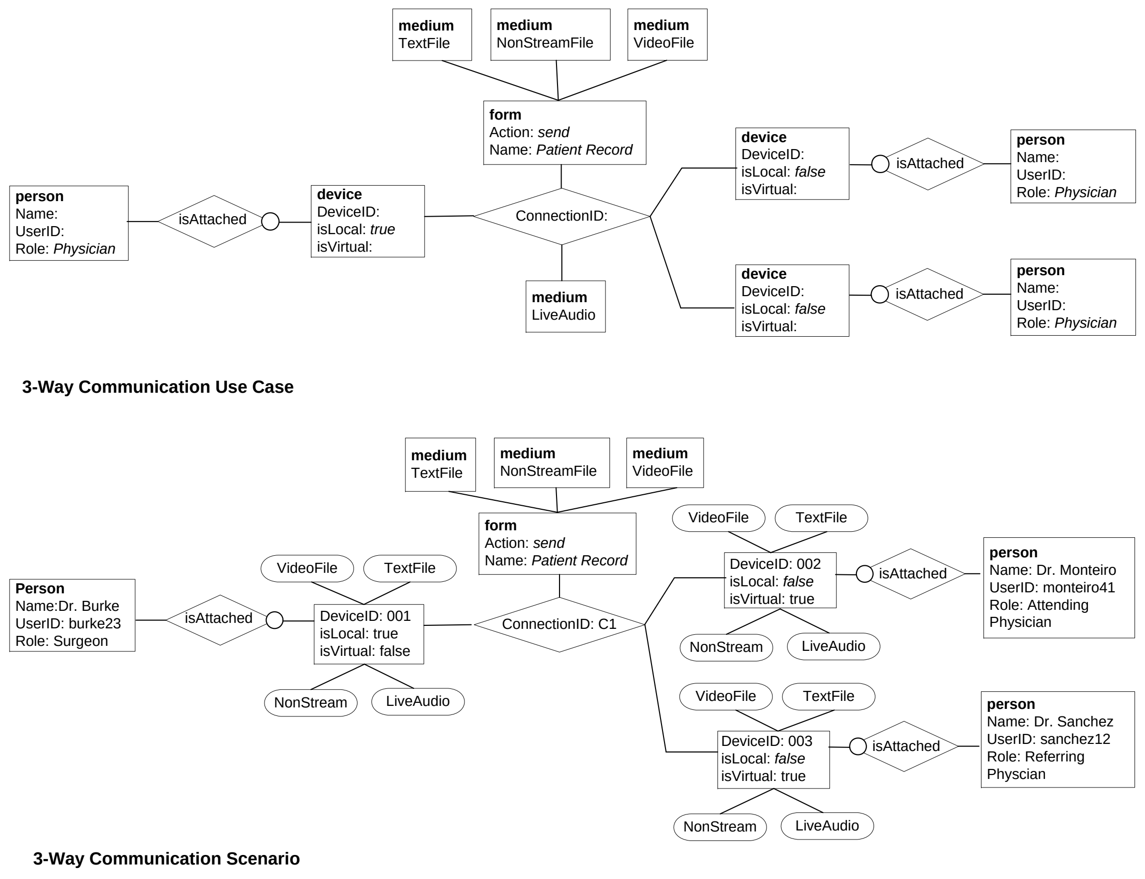

medium TextFile
medium NonStreamFile
medium VideoFile
form Action: send Name: Patient Record
person Name: UserID: Role: Physician
isAttached
device DeviceID: isLocal: true isVirtual:
ConnectionID:
medium LiveAudio
device DeviceID: isLocal: false isVirtual:
isAttached
person Name: UserID: Role: Physician
device DeviceID: isLocal: false isVirtual:
isAttached
person Name: UserID: Role: Physician
3-Way Communication Use Case
medium TextFile
medium NonStreamFile
medium VideoFile
form Action: send Name: Patient Record
Person Name:Dr. Burke UserID: burke23 Role: Surgeon
isAttached
VideoFile
TextFile
DeviceID: 001 isLocal: true isVirtual: false
NonStream
LiveAudio
ConnectionID: C1
VideoFile
TextFile
DeviceID: 002 isLocal: false isVirtual: true
NonStream
LiveAudio
isAttached
person Name: Dr. Monteiro UserID: monteiro41 Role: Attending Physician
VideoFile
TextFile
DeviceID: 003 isLocal: false isVirtual: true
NonStream
LiveAudio
isAttached
person Name: Dr. Sanchez UserID: sanchez12 Role: Referring Physcian
3-Way Communication Scenario

Figure 5.7: Three-Way Communication Scenario

From the change mapping, the Change Interpreter now generates:

`enableStream`

Querrying the state machine yeilds as control scripts:

`enableinitiator("CI","LiveAudio")`

and

`sendSchema"C1","burke32","monte06",CI_1,DI_1)`

The runtime model now becomes (CI_1, DI_1)

When Dr Burke submits the second data instance (c), DI_2, this triggers the Model Comparator to compare the runtime model (CI_1, DI_1) and (CI_1, DI_2) which yields the following change:

`added(form("DisPkg_1"))`

The Change Interpreter raises the following event:

```
sendform
```

Querying the LTS results in leads to control scripts:

```
sendForm("C1","DisPkg_1", "RecSum-Jane.txt"),

sendForm("C1","DisPkg_1", "xRay-Jane.jpg") ,

sendForm("C1","DisPkg_1", "HeartEcho-Jane.mpg") and

sendSchema"C1","burke32","monte06",CI_1,DI_2)
```

The runtime model now becomes (CI_1, DI_2)

5.4.3 SCENARIO 2 - CVM Application

If in Scenario 1 Dr. Burke did not shut down the communication connection, yet instead had decided that he needed to include Dr. Sanchez, the referring physician then this would involve adding an additional person corresponding to the control instance shown in Figure 5.7.

The submitance of the new control instance CI_2 triggers a comparison between CI_1 current runtime control instance and the new user control instance CI_2 yielding as change:

```
added(device,isAttached ,person)
```

raising:

```
initiateReNeg
```

A query to the LTS generates:

```
("C1","burke32","monte06", "sanchez12", CI_1,DI_2)
```

The comnnection achieves stasis when the runtime model is updated to (Ci_2, DI_2)

5.5 The Energy Management Domain Evaluation Scenarios

In this section we present indicative scenarios from the energy management domain to test the ability of the GMoE based synthesis engine instantiation to generate control scripts as in the earlier prototype described in Chapter 3 of this dissertation. A walk-though of the model processing components will be presented to demonstrate how the prototype builds the control scripts for the microgrid DSVM.

5.5.1 Microgrid SE Setup

The GMoE prototype had been configured to generate scripts from the communications domain in the previous scenarios therefore will not currently accept the microgrid models. To restructure the prototype to for microgrid models we will have to remove the domain specific aspects which form the CVM and replace it with that of MGridVM. Since the structure of the DSVM SE model should remain the same there is no need to change the SE metamodel. Additionally the LTSs follow the same structure so the same LTS metamodel is also valid. The change in DSK occurs by swapping the *SE_launcher's* link to the SE model from `CVM_SE.xmi` to `MGRIDVM_SE.xmi`. in addition the framework's language metamodel has to be changed from `CML.ecore` to `MGRIDML.ecore`. Once this is accomplished and the `MGRIDML.ecore` is registered to in *kermeta* then we are ready to accept microgrid based models.

5.5.2 SCENARIO 3 - MGridVM Application

This simple scenario begins with our actor Lisa, a resident of southern United States needing to configure her home's microgrid to incorporate the operation of a newly installed controllable pool pump and her air conditioning unit. Currently the system has no configuration, therefore Lisa has to submit a MGridML control schema to indicate the configuration of the plant and a data instance to signify the desired properties of the devices to be controlled. Lisa wants to first deploy a configuration which identifies her grid structure. This control Schema should include her smart-meter which enables her to exchange data with her local utility, and logical load controllers through which she regulates these devices. The devices mentioned are consumers of electricity and are therefore loads. The control instance submitted is shown in Figure 5.8 (a). The instance shows the core element, the *microgrid central controller*, MCG001. Connected to MC001 are two controllers. One is a *point of common coupling*, PCC001, and the other is a *load controller* which she intends to use in managing her devices. Attached are to each controller are their respective types which are for describing and associating the devices in the data instance.

The submission of the first control instance to GMoE prototype triggers a comparison between the current runtime model $(null, null)$ to the new user define instance $(CI_0, null)$. The resulting changelist is:

```
added (MCGrid ("MCG001") ,
LoadController("LC001"),
LoadDeviceType("LTD001"),
LoadDeviceType("LTD002"),
PCC("PCC001") ,
SmartMeterType("SMT001"))
```

The Change Interpreter now accesses the change mapping and raises the following events:

```
initialMCG,
createLC
create PCC
addLC
addPCC
```

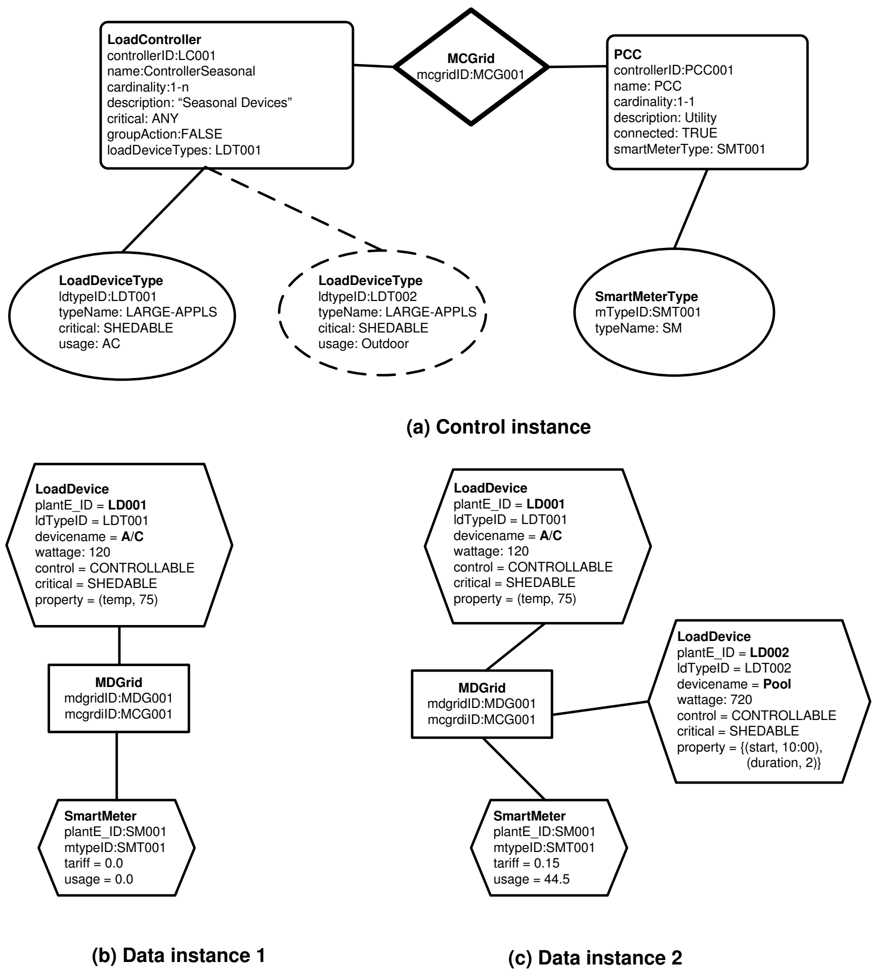


Figure 5.8: MGridML Scenario 1

`addLDT`

`addSMT addLDT`

The LTSs are now queried and yeilds the following control scripts:

`initializeMGrid("MCG001")`

```
addLoadController("LC001", " LDT001,LDT002" )

addPCCController("PCC001", "SMT001")

...................

addLoadDeviceType("LDT001", "AC")

addMeterType("SMT001","SM")

addLoadDeviceType("LDT002", "AC")
```

At this point, the *stub*, which is the stand in for the lower layers of the DSVM (the Microgrid Control Middleware (MCM) and MicroGrid hardware Broker (MHB)) returns a confirmation that the scripts were executed and the runtime model is updated to $(CS_1, null)$. At this point Lisa's microgrid is not fully configured and needs a data instance to tell the DSVM the particulars regarding the desired operational states of the devices she wants to manage.

Lisa next submits a data instance, Figure 5.8 (b) which only contains the connection to the utility via the smartmeter and only one load as a test to be cautious; the air conditioner which she intends to work at 75 Degrees.

By submitting the new data instance the model comparator *diffs* the current runtime model $(CI_1, null)$ against the new user preference model, (CS_1, DI_1). Recall that Di_1 is Figure 5.8 (b). The changelist generated is :

```
added(MDGrid("MDG001","MCG001"),
LoadDevice("LD001", "LDT001", "temp,75"),
SmartMeter("SM001","SMT001"))
```

The resulting events raised by accessing the change mapping are:

```
addLD
addSMT
addSD
```

The LTSs are queried and the following microgrid control scripts are generated:

```
addLoadDevice("LD001", "LDT001", "temp,75")
addSmartMeter("SM001","SMT001")
```

Being sufficiently satisfied, Lisa's now adds the second load which is the pool. To accomplish this she appends Figure 5.8 (b) with a new node denoting the pool and its setting to create Figure 5.8 (c). She wants this device to be attached to the outdoor LoadDevice type. Note that she did not simply submit the MDgrid node with the pool as a loadDevice attached. This would result in the system releasing the AC from control; effectively pruning it from its runtime model. This is important as the MGridVM semantics dictates that the model elements need to be explicitly defined and represented in models. Submitting the preference model(Ci_1, DI_2) results in the changelist:

```
added(LoadDevice("LD002", "LDT002", "start,10:00,duration,2"))
```

the subsequent LTSs query results in:

```
addLoadDevice("LD002", "LDT002", "start,10:00,duration,2"))
```

The scenario ends when the SE receives confirmation that the scripts are executed and statsis is achieved when the runtime model is reconciled to (CI_1, DI_2) .

5.5.3 SCENARIO 4 - MGridVM Application

In this scenario, Lisa wants to make repairs to the AC and worries about it suddenly becoming active. She also wants to automate her outdoor lights to turn on at 6pm and turn off at daybreak which is about 5pm. She joins this device to the outdoor device types. The system may choose to treat these devices distinctly due to some concern defined by the user, however in this prototype we have not implemented policy concerns. To address the new configuration she submits the model shown in Figure 5.9. We assume that the existing configuration from the previous scenario is still active and the runtime model is (CI_1, DI_2). The submission causes the model comparator to yield the following as changes:

```
removed(LoadDevice(LD001))
added(LoadDevice("LD003", "LDT002", "start,18:00,duration,11"))
```

The corresponding events raised are:

```
removeLD
addLD
```

This in turn queries the LTSs and yields:

removeDevice(”LD001”)

addLoadDevice(”LD003”, ”LDT002”, ”start,18:00,duration,11”))

5.5.4 Result

By putting the prototype through its paces using these scenarios we found out the its capabilities were the same as the previous prototypes. The models we used, while structurally complete , used elements with minimal attributes needed to be processed to control scripts.

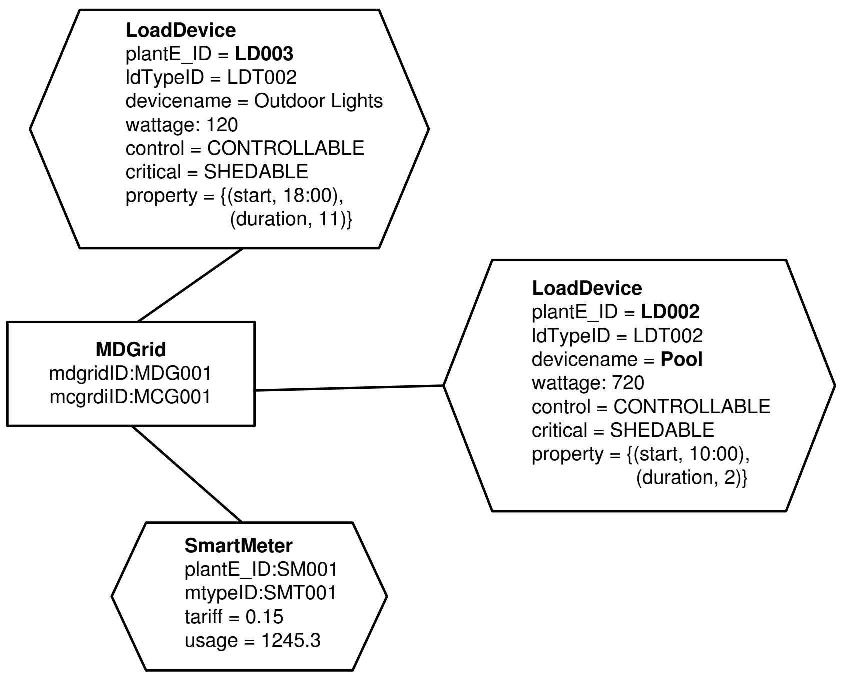
5.6 Evaluation of Instantiation

The second dimension of our study applies to the overarching goal of the GMoE to facilitate reuse of model interpretation knowledge by furnishing a generic framework. To evaluate the utility of the prototype our objectives were:

Objective 1: To ascertain an estimate to which the approach saved developmental effort through reuse.

Objective 2: To determine the extent of architecture's aptness to be restructured using an analysis and comparison of coupling metrics of the GMoE with earlier prototypes.

Objective 3: To inspect the performance of the GMoE prototype compared to V1 prototypes by recording the running times under increasingly more complex models.



(a) Data instance 1

Figure 5.9: MGridML Scenario 2

5.6.1 Experiment Setup

To evaluate the first objective we (1) used compiled historical development data related to programmer hours to build new SEs utilizing the GMoE methodology and the earlier CVM and MGridVM prototypes, and (2) assuming a correlation between complexity and work effort as in [3] we use SLOC, number of classes, and number of methods as code metrics to show a relationship to the development time.

The analysis to satisfy our second objective concerns coupling. We compared the earlier coupled prototypes with the GMoE prototype along the following dimensions:

- *Inbound Intra-Package Method Dependencies (IIPM)* - methods in other classes of the same package that depend on this method.

- *Outbound Intra-Package Feature Dependencies (OIPF)* - methods and fields within the classes of the same package that this method depends on.

- *Inbound Extra-Package Method Dependencies (IEPM)* - methods in other packages that depend on this method.

- *Outbound Extra-Package Feature Dependencies (OEPF)* - methods and fields in other packages that this method depends on.

The aforementioned metrics were captured using *Dependency Finder* [73], which is a suite of tools for analyzing compiled Java code, particularly, computing object-oriented software metrics that give you an empirical quality assessment of the code.

The third objective requires a comparison of the running times of the earlier prototypes against the GMoE based SEs. To accomplish this we reused the data collected from the chapter 3 evaluation of the V1 prototypes along with the same models (CML and MGRIDML) as input. In seeking a true reflection of the performance we controlled threats by reusing the same machine, models and collection mechanisms used in the prior experiments. All prototypes were pretested to ensure that they generated the same control scripts given the same input models.

5.6.2 Results

Table 5.1 shows the evaluation metrics for the synthesis engines in both domains. Columns 2 and 3 show the data for the earlier version (v1) of the SEs, Columns 4 and 5 the current version (v2) of the SEs built from the GMoE and DSK. The GMoE versions values are shown as sums of the GMoE and DSK. The SE in the the second versions are smaller in size than the v1 SEs since the DSK is represented as a model in Kermeta and we currently do not generate the Java code for the DSK. We plan

Table 5.1: Comparison of static code metrics and development times for both versions of the SE.

Metric	MGVM (v1)	CVM (v1)	MGVM (v2) (GMoE + DSK)	CVM (v2) (GMoE + DSK)
SLOC	2913	963	(314 + 718) = 1035	(314 + 452) = 766
# Classes	31	21	(26 + 49) = 75	(26 + 38) = 64
# Methods	434	156	(187 + 216) = 403	(187 + 78) = 265
Development Time (hrs)	155	130	(38 + 35) = 73	(38 + 25) = 63

Table 5.2: Comparison of coupling metrics for the classes in both versions of the SE that form the GMoE.

Coupling	MGVM (v1)	CVM (v1)	GMoE (v2)
IIPM	80	7	2
OIPF	135	13	2
IEPM	0	25	0
OEPF	336	107	112
Total:	551	152	116

to perform full code generation in future work. The final row of Table 5.1 shows the development time to create the versions of the SEs.

The coupling measures at the method level for Objective 2 are shown in Table 5.2. Top part of the table shows the static code metrics for the classes in SE that represent the GMoE. Note the lowest number of methods for the two versions of the SE is for the SE v2. The lower part of the table shows the coupling metrics including the method level metrics previously described. In general the coupling measures for the SE v2 is the lowest, except for OEPF which is higher for SE v2 than for CVM v1. This is expected since there are a large number of features dependent on the classes and methods containing the DSK. This would suggest that in general GMoE used in the SE v2 has a higher level of reusability in the development of SEs for other domains. Note however, this measurement was taken in the context of an application developed using Kermeta.

Table 5.3: Comparison of performance metrics and regression slopes for the SE prototypes.

M	MGVM (v1)	CVM (v1)	MGVM (v2) (GMoE + DSK)	CVM (v2) (GMoE + DSK)
10	0.97	1.17	3.82	4.52
14	1.3	1.8	5.42	6.74
18	1.68	2.07	7.06	9.29
22	2.07	2.42	9.83	12.07
26	2.42	2.76	11.85	14.81
30	2.71	3.17	13.74	18.22
Regression Slopes	0.088	0.094	0.512	0.681

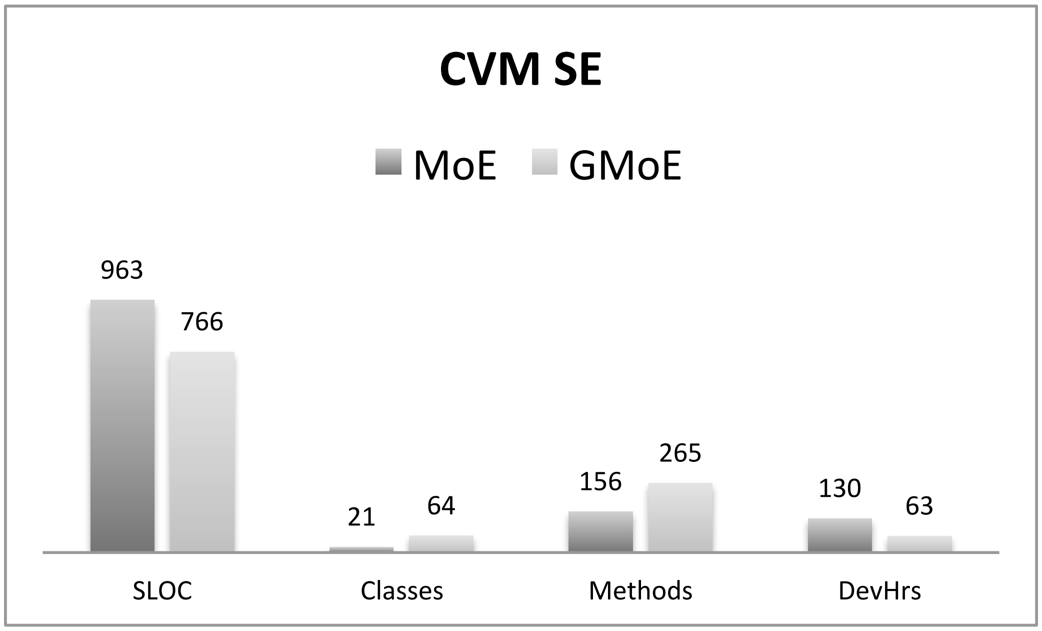


Figure 5.10: Comparison of CVM v1 and CVM v2

The performance metrics for objective 3 are shown in Table 5.3. In the top section of the chart is the performance times for each prototypes in seconds elapsed to process models of with number of nodes M. The lower section shows the slopes of the regressions applied to the respective datasets.

5.6.3 Discussion

Figures 5.10 and 5.11 show a graphical representation of our comparison of the version of the SE. The v2 approach required an increase in the number of classes and methods;

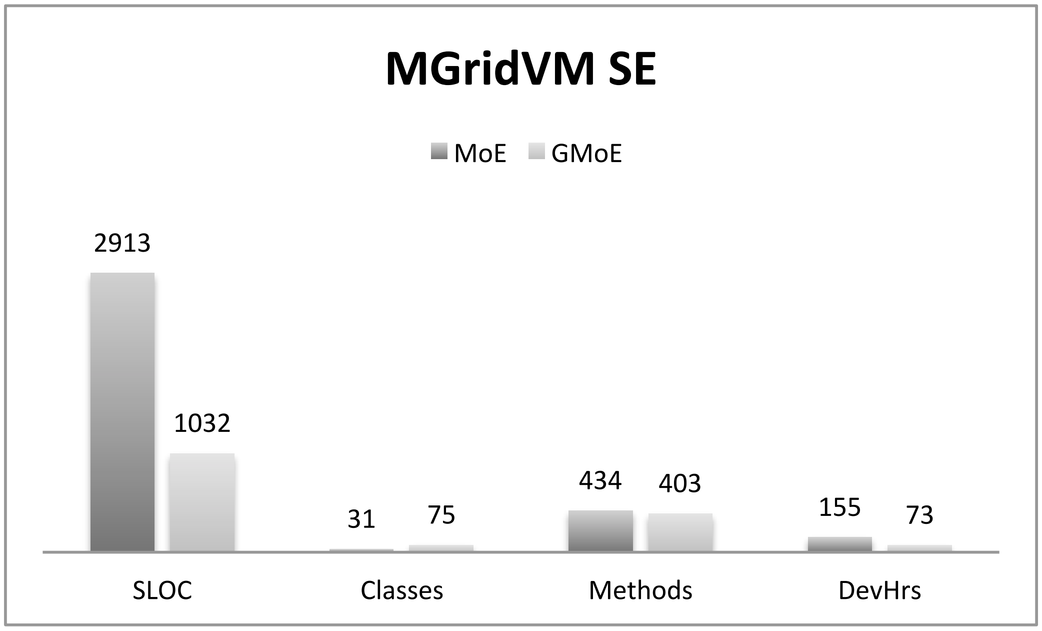


Figure 5.11: Comparison of MGridVM v1 and MGridVM v2

this can be attributed to the additional gluing mechanisms required. We were gratified to see the time to develop the v2 SEs using Kermeta is approximately half of the time to develop the v1 SEs. This can be explained as Kermeta was used in the second generation prototypes. Kermeta has allowed for less debugging and coding as we were operating at a higher level of abstraction. The coupling between our aspects was significantly reduced which was expected as our paramount focus was on the loose coupling between the two concerns.

Figure 5.12 shows the graphical representation of the performance metrics for the prototype versions corresponding to Table 5.3. We have used regressions and assume a linear trend to determine the slopes of the datasets and are sufficiently satisfied to proceed with the analysis as our correlations coefficents are over .98 for the sampling. We focus on three critical aspects which emerge from our inspection.

Firstly, the elevation or *y intercept* differences between prototypes of the same domain (v1 vs. v2) are larger which reflects a significant decrease in performance by the v2 prototypes. We expected that the use of kermeta in the v2 prototypes

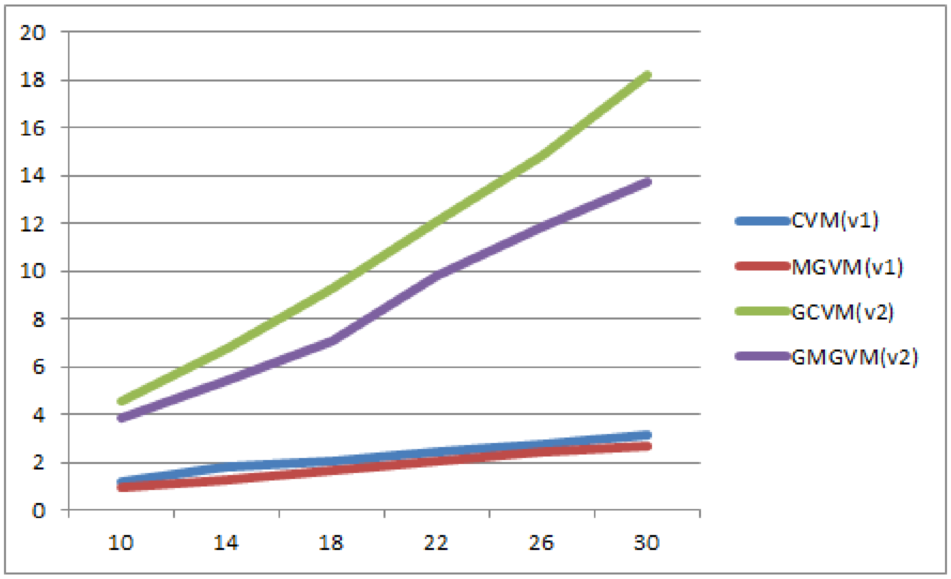


Figure 5.12: Comparison of v1 and v2 performances

would result in less optimal lower level code than the java counterparts, therefore our supposition was validated.

Secondly the v2 prototypes show a steeper slopes as model complexity increases. We tested the difference of rates and found a p-value of 1.63×10^{-9}, which concludes a significant difference. This demonstrates than not only are there increased performance times between versions but the performance degradation widens. Again this can be attributable to the kermeta implementation. Our third observation which became more apparent with the v2 prototypes is the higher slopes for CVM regressions over the MGVM regressions. This is also evident in the v1 prototypes but less visible. To delve deeper into this phenonena we apply the analysis of covariance (ANCOVA) model to analyze the relationship between v2 prototype performances. ANCOVA uses a null hypothesis stating that the regressions are statistically equal. The tests for homogeneity between regressions rejects the null hypothesis with $p = 0.000530 < .05$.

We assume the critical p-value of .05 for statistical significance. We can therefore assert that the regressions for the v2 prototype performances are statistically significantly dissimilar over the dataset. This we attribute to the structural difference between the models in each domain, exaggerated by the performance lag experienced by kermeta.

5.6.4 Threats to validity

In accord with Wohlin et.al. [81], we considered both internal and extrenal threats to validity for all three evaluation objectives. The GMoE prototype was developed using some kermeta modules which are at a higher level of abstraction than Java code. This is an external threat to the approximation of effort done in objective 1. The *Compile Kmt to EMF plugin* (Java) is still experimental at present, but we intend to address this threat when the transformation becomes available. An internal threat to objective 1 is our assumption of a causal relationship between complexity and work effort. In addition, while source lines of code (SLOC) and the quantity of classes and methods are a good indicator of complexity, these metrics are by no means exhaustive.

An internal threat which affects all objectives is It has to be taken into consideration that the developers gained expertise in model-driven development. The same developers were used in the MGridVM project and could have gained some developmental domain insight to build the GMoE more efficiently. The development of the CVM (v1) prototype and GMoE prototype (v2) were developed by different developers which may skew the comparison. An external threat to objective 2 is the dimensions along which the coupling comparison was evaluated. While these metric are not exhaustive we are sufficiently confident that they highlight the modular differences in the approaches.

We used the same models and results from the version 1 prototypes in objective 3. This means that the internal threat of not using very large models in our performance evlauation is carried over. This threat could result in greater than linear synthesis times for models of complexity greater than those evaluated.

5.7 Chapter Summary

In this chapter, we presented our methodology to refine the commonalities of the interpreter logic for i-DSML synthesis engines into a GMoE. We subsequently presented details of our GMoE prototype implementation to demonstrate the proof of concept. The prototype was sufficiently validated by ensuring that the scenarios from the version 1 prototypes produced the same control scripts as output.

Three comparative studies against the earlier prototypes were performed: (1) the effort required in synthesis engine development; (2) measuring the coupling of the DSK; and (3) a performance analysis of the GMoE prototype. The first two studies provides an argument for productivity gains and increased quality by employing the GMoE. The third study recorded large performance losses in model synthesis by the GMoE. This should however be tempered by considering that this prototype was implemented using kermeta. In our next section we conclude by consolidating the research put forward in this dissertation.

CHAPTER 6

CONCLUSION

6.1 Introduction

In this dissertation we investigated the problem of *how to decouple the domain-specific knowledge from the model of execution in the synthesis engine, thereby producing a generic model of execution.* We have contended that in developing a generic model of execution for synthesizing i-DSML models we may assist authors of i-DSMLs to develop their language syntax and the required execution semantics in a totalistic manner. By the development of this methodology I sought to reduce the effort required to develop subsequent DSVMs. There existed a gap in the research surrounding the specification of execution semantics and no previous work addressing reusing interpreter logic as it applies to i-DSMLs. In addressing the primary research problem the study explored three distinct subproblems:

1. How to formulate the semantics of model synthesis for in MGridVM, based on changes to user-defined models at runtime?

2. How can DSK semantics be represented in a persistent manner and the MoE be defined to support synthesis engine instantiation for a specified domain?

3. How to instantiate the synthesis engine, given a representation of the DSK and a GMoE framework?

At the onset of the investigation into a solution to the first subproblem there was an existing i-DSML, Communications Virtual Machine (CVM). In developing the DSVM for a new i-DSML, MgridML, we noted that not only did the language's abstract syntax necessitate the separation of Data and Control constructs, but the execution semantics required similar components to the existing CVM. This set in motion our research path into a deep investigation towards exploiting the commonalities. A second observed phenomena was the tremendous effort required for a DSVM build and the level of redundancy in the development of DSVMs. Both virtual machines required well in excess of a hundred hours of development time, much of which was dedicated to implementing very similar model processing components. This furthered our zeal to provide a methodology that can be used by other language authors, promoting the focus of developmental efforts on that portion of the synthesis engine that is wholly specific to the domain.

Within this chapter we first present the empirical analysis of each subproblem in Section 6.2 then discuss the implications of the study in Section 6.3. We will address recommendations for future research in Section 6.4 and conclude the chapter and this dissertation in Section 6.6.

6.2 Empirical Analysis

While we employ scenario traces to validate the builds of each major prototype. We have relied heavily on empirical analysis to guide our path towards addressing the research question. Of major concern was greater than linear time responses to increased complexity. This would indicate a limitation on the complexity of the models and thefore the utility of the approach.

The MGridML/VM prototype was developed in two stages. An early rudimentary prototype we developed to determine feasibility of the applying an i-DSML approach to the energy management domain. The success in the alpha prototype prompted a refinement to a MoE with a loosely coupled intrinsic DSK. This prototype was evaluated on its ability to satisfy its core requirements and also compared with the early CVM prototype in which the DSK was more tightly coupled. The results of the empirical analysis demonstrated that the prototype could process models of increasing complexity in linear time. The loosely coupled architecture showed some performance tradeoffs as expected.

To separate and develop a persistent representation of the DSK as artifacts we examined the model of execution of the previous prototype and refactored the architecture according to the principles of aspect oriented design. With a focus on separated the concerns, the DSK being the primary concern, we observed that the artifacts for the DSK were of two primary types, *syntax* and *semantic* elements. The syntax for the architecture had from the i-DSML metamodel, while the semantics could be contained within automata representations of the pertinent model concepts or elements and a change mapping. To make these entities persistent we employed a labeled transition system representation and a simple tabular representation respectively. Both may be store in file format. The second subproblem called for the DSK to be easily representable in the two domain platforms. The remainder of the MoE was reified to a framework with the DSK as the frameworks extension. To bind the DSK to the framework we took a model-based approach. The architecture is such that the framework should be able to read the DSK and supporting artifacts as a model with a fixed configuration but with multivariate internal properties and attributes.

The GMoE approach was reified to a third prototype as proof of concept and for evaluation in terms of effort expended in artifact development, coupling of concerns and performance. The first set of experiments quantified the benefits of the GMoE approach with respect to implementation effort. What was found is roughly a 50% reduction in development time. The significance of this reduction has to be tempered by the fact that the developers had gained some expertise after working in the domain and were working at a higher level of abstraction using the kermeta language.

The second set of experiments validated our hypothesis and general intent for a loosely coupled architecture. The metrics applied to coupling were *Inbound Intra-Package Method Dependency*, *Outbound Intra-Package Feature Dependency*, *Inbound*

Extra-Package Method Dependency , and *Outbound Extra-Package Feature Dependency.* The emphasis on coupling stems for a quality consideration, but foremost we postulated that by reducing coupling, the gluing mechanism required to recombine the GMoE and a DSK would be small and unobtrusive within the development process.

The third set of experiments focused on the performance of the GMoE based prototype. Since the earlier prototype had reflected linear times for models of increasing complexity we wanted to ensure that this response was carried through to the new prototype. The new prototype also reflected linear times which alleviated our concerns. The performance however was severely degraded with the overhead costs of kermeta. We were pleased with the outcome of this aspect of the study as performance times greater than linear would impact the utility of the approach.

In total the empirical analysis results demonstrated no significant deviation from our expectations. We are satisfied with he breath of the tests performed at each stage of the investigation.

6.3 Implications

The implication of this approach has far reaching scope. One of the major obstacles to the widespread acceptance and application of i-DSMLs is the magnitude of effort and expertise required in the development of their execution engines. While this dissertation addresses only the synthesis engine layer of the DSVM, we have seen that the approach fosters a reuse of expertise and redundant components. This approach will allow for less effort being expended in the development of new i-DSMLs. Secondly, since the language developer will know a priori as to the artifact requirements for her DSK, then the metamodel and DSK can be built at the same time ensuring traceability. The developer would now be able to describe a model element's syntax and go on to describe its execution; building the language downwards.

6.4 Future Work

We did not implement all the features that were identified in our feature analysis of the microgrid domain. The inclusion of policies within MgridML will allow for greater control of the plant. For example the user could attach a policy to the the PCC to state that if the voltage of the macro-grid falls below a defined threshold then the microgrid should island itself. The user could also state via policy that if the tariffs from the macro-grid exceeds .012 USD per Kilo Watt then switch to storage reserves, and then again only if the storage capacity is greater than 50 percent.

The inclusion of policies has exiting possibilities as now behavior would be derived within the model and from model changes. Now we would have to revisit the synthesis engine to allow for interpreting policies. A further consideration which makes this path even more interesting is to investigate how conflicting policies would be resolved. A second feature that we believe is very interesting to pursue is predictive augmentation of the DSVM by expanding its environmental sensory facilities such as accessing weather forecasting web services, and utilizing the user energy usage history. A third consideration meriting exploration within MGRIDVM is in the assurance of privacy, security and safety. While some portion of these concerns may be addressed through policy, there is definitely much work that is required to harden the DSVM. With respect to the GMoE we foresee exploration into the arena of autonomic computing for the synthesis engine. The synthesis engine is the pivotal technology for i-DSML interpretation and as such will see much attention in the future.

6.5 Summary

At the onset of the dissertation there existed no reusable construct to reuse interpreter logic for i-DSMLs. We have presented a GMoE and a specification for persistent DSK artifacts which can be used to develop i-DSML synthesis engines. We encourage the research community to contribute its collective wisdom in developing this paradigm which is in its infancy.

BIBLIOGRAPHY

[1] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software & Systems Modeling*, 5(3):261–288, 2006.

[2] M. Alanen and I. Porres. Difference and union of models. In *UML 2003-the unified modeling language: modeling languages and applications: 6th international conference, San Francisco, CA, USA, October 20-24, 2003: proceedings*, page 2. Springer, 2002.

[3] Allan J. Albrecht and John E Gaffney Jr. Software function, source lines of code, and development effort prediction: a software science validation. *Software Engineering, IEEE Transactions on*, (6):639–648, 1983.

[4] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[5] Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, et al. State chart xml (scxml): State machine notation for control abstraction. *W3C Working Draft*, 2007.

[6] Nelly Bencomo, Amel Bennaceur, Paul Grace, Gordon Blair, and Valérie Issarny. The role of models@ run. time in supporting on-the-fly interoperability. *Computing*, 95(3):167–190, 2013.

[7] Ted J Biggerstaff and Alan J Perlis. Software reusability: vol. 1, concepts and models. 1989.

[8] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[9] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[10] B.R. Bryant, J. Gray, M. Mernik, P.J. Clarke, R.B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems/ComSIS*, 8(2):225–253, 2011.

[11] 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.

[12] Tony Clark, Paul Sammut, and James Willans. *Superlanguages: developing languages and applications with XMF.* Ceteva, 2008.

[13] P.J. Clarke, V. Hristidis, Y. Wang, N. Prabakar, and Y. Deng. A declarative approach for specifying user-centric communication. In *Collaborative Technologies and Systems, 2006. CTS 2006. International Symposium on*, pages 89–98. IEEE, 2006.

[14] Siobhàn Clarke and Elisa Baniassad. *Aspect-oriented analysis and design.* Addison-Wesley Professional, 2005.

[15] Benoît Combemale, Xavier Crégut, Marc Pantel, et al. A design pattern to build executable dsmls and associated v & v tools. In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, 2012.

[16] J den Haan. Model driven development: Code generation or model interpretation?

[17] Y. Deng, S. Masoud 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.

[18] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[19] A. Dimeas and N. Hatziargyriou. A multi-agent system for microgrids. *Methods and Applications of Artificial Intelligence*, pages 447–455, 2004.

[20] A.L. Dimeas and N.D. Hatziargyriou. Operation of a multiagent system for microgrid control. *Power Systems, IEEE Transactions on*, 20(3):1447–1455, 2005.

[21] George Edwards, Yuriy Brun, and Nenad Medvidovic. Automated analysis and code generation for domain-specific models. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 161–170. IEEE, 2012.

[22] George Edwards and Nenad Medvidovic. A methodology and framework for creating domain-specific development infrastructures. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 168–177. IEEE, 2008.

[23] George Edwards, Chiyoung Seo, and Nenad Medvidovic. Model interpreter frameworks: A foundation for the analysis of domain-specific software architectures. *Journal of Universal Computer Science*, 14(8):1182–1206, 2008.

[24] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[25] Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[26] Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[27] T. Fernando. Comparative transition system semantics. In *Computer Science Logic*, pages 149–166. Springer, 1993.

[28] Robert Filman, Tzilla Elrad, Siobhán Clarke, et al. *Aspect-oriented software development*. Addison-Wesley Professional, 2004.

[29] DSM Forum. Domain specific modeling, 2011. http://www.dsmforum.org/ (March).

[30] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series. Pearson Education, 2010.

[31] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[32] Daniel P Freedman and Gerald M Weinberg. *Handbook of walkthroughs, inspections, and technical reviews: evaluating programs, projects, and products*. Dorset House Publishing Co., Inc., 2000.

[33] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM, 2010.

[34] Sumit Gulwani. Synthesis from examples. *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, 10(2), 2012. Invited talk paper.

[35] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012. Invited talk paper.

[36] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays)*, pages 19–35, 2003.

[37] N. Hatziargyriou, H.Asano, R. Iravani, and C. Marnay. Microgrids. *IEEE Power & Energy Magazine*, july:78–94, 2007.

[38] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] Z. Jiang and R.A. Dougal. Hierarchical microgrid paradigm for integration of distributed energy resources. In *Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE*, pages 1–8. IEEE, 2008.

[40] M. Jiménez, F. Rosique, P. Sánchez, B. Álvarez, and A. Iborra. Habitation: A domain-specific language for home automation. *Software, IEEE*, 26(4):30–38, 2009.

[41] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis. Technical Report CMU/SEI-90-TR-21, CMU, Nov 1990.

[42] Gabor Karsai. Structured specification of model interpreters. In *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS'99. IEEE Conference and Workshop on*, pages 84–90. IEEE, 1999.

[43] F. Katiraei, M.R. Iravani, and PW Lehn. Micro-grid autonomous operation during and subsequent to islanding process. *Power Delivery, IEEE Transactions on*, 20(1):248–257, 2005.

[44] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Pr, March 2008.

[45] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for uml models. *Software Engineering*, 64:105–116, 2005.

[46] R.B. Kieburtz, L. McKinney, J.M. Bell, J. Hook, A. Kotov, J. Lewis, D.P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, page 552. IEEE Computer Society, 1996.

[47] P. Kinney et al. Zigbee technology: Wireless control that simply works. In *Communications design conference*, volume 2, 2003.

[48] B. Kroposki, R. Lasseter, T. Ise, S. Morozumi, S. Papatlianassiou, and N. Hatziargyriou. Making microgrids work. *Power and Energy Magazine, IEEE*, 6(3):40–53, 2008.

[49] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[50] Robert Lasseter, Abbas Akhil, Chris Marnay, John Stephens, Jeff Dagle, Ross Guttromson, A. Sakis Meliopoulous, Robert Yinger, and Joe Eto. White paper on integration of distributed energy resources. the certs microgrid concept. Technical report, Consortium for Electric Reliability Technology Solutions, prepared for the U.S. Department of Energy, April 2002.

[51] Yuehua Lin, Jeff Gray, and Frdric Jouault. Dsmdiff: A differentiation tool for domain-specific models. *European Journal of Information Systems, Special Issue on Model-Driven Systems Development (Mark Lycett, Esperanza Marcos, and Veda Storey, eds.)*, 16(4):349–361, 2007.

[52] Raphael Mannadiar and Hans Vangheluwe. Modular artifact synthesis from domain-specific models. *Innovations in Systems and Software Engineering*, 8(1):65–77, 2012.

[53] C. Marnay and O. Bailey. The certs microgrid and the future of the macrogrid. *Berkeley Lab Report# LBNL-55281*, 2004.

[54] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[55] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[56] Marjan Mernik and IGI Global. *Formal and practical aspects of domain-specific languages: Recent developments*. Information Science Reference, 2013.

[57] K.A. Morris, J. Wei, P.J. Clarke, and F.M. 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. IEEE, 2012.

[58] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Model Driven Engineering Languages and Systems*, pages 264–278. Springer, 2005.

[59] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, 1992.

[60] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software engineering - theory and practice (4. ed.)*. Pearson Education, New Jersey, USA, fourth edition, 2009.

[61] M Pipattanasomporn, H Feroze, and S Rahman. Multi-agent systems in a distributed smart grid: Design and implementation. In *Power Systems Conference and Exposition, 2009. PSCE'09. IEEE/PES*, pages 1–8. IEEE, 2009.

[62] Elvinia Riccobene and Patrizia Scandurra. Weaving executability into uml class models at pim level. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, page 1. ACM, 2009.

[63] Horst WJ Rittel and Melvin M Webber. Planning problems are wicked. *Polity*, 4:155–69, 1973.

[64] José Eduardo Rivera, José Raul Romero, and Antonio Vallecillo. Behavior, time and viewpoint consistency: Three challenges for mde. In *Models in Software Engineering*, pages 60–65. Springer, 2009.

[65] Daniel A Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In *Model Driven Architecture–Foundations and Applications*, pages 63–78. Springer, 2008.

[66] Devon Simmonds, Raghu Reddy, Robert France, Sudipto Ghosh, and Arnor Solberg. An aspect oriented model driven framework. In *Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, EDOC '05, pages 119–130, Washington, DC, USA, 2005. IEEE Computer Society.

[67] M.A. Simos. Organization domain modeling (odm): Formalizing the core domain modeling life cycle. *ACM SIGSOFT Software Engineering Notes*, 20(SI):196–205, 1995.

[68] O Sims. Presentation: Mda: The real value. *Object Management Group website: www. omg. org/mda/presentations. htm*, 2002.

[69] Thomas Stahl, Markus Voelter, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, first edition, 2006.

[70] J. Stanek, S. Kothari, and Kang Gui. Method of comparing graph differencing algorithms for software differencing. In *Proceedings of the IEEE International Conference on Electro/Information Technology, 2008. EIT 2008.*, pages 482 – 487, May 2008.

[71] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[72] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice.* Wiley Publishing, 2009.

[73] Jean Tessier. Dependency finder, version 1.2.1-beta4, November 2010. http://depfind.sourceforge.net/(December,2013).

[74] Antoine Toulmé and I Inc. Presentation of emf compare utility. In *Eclipse Modeling Symposium*, pages 1–8, 2006.

[75] KG Van den Berg, Jose Maria Conejero, and Ruzanna Chitchyan. Aosd ontology 1.0-public ontology of aspect-orientation. 2005.

[76] A. Van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[77] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):36, 2000.

[78] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management.* John Wiley & Sons, 2013.

[79] Y. Wang, D.J. DeWitt, and J.Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 519–530. IEEE, 2003.

[80] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. A survey on uml-based aspect-oriented design modeling. *ACM Comput. Surv.*, 43(4):28:1–28:33, October 2011.

[81] Claes Wohlin, Per Runeson, Martin Hst, Magnus C Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in software engineering.* Springer Publishing Company, Incorporated, 2012.

[82] C. Wolfe, T. Graham, and W. Phillips. An incremental algorithm for high-performance runtime model consistency. *Model Driven Engineering Languages and Systems*, pages 357–371, 2009.

[83] 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, 2011.

[84] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65. ACM, 2005.

[85] Z. Yang and M. Jiang. Using eclipse as a tool-integration platform for software development. *Software, IEEE*, 24(2):87–89, 2007.

[86] A.A. Zaidi and F. Kupzog. Microgrid automation-a self-configuring approach. In *Multitopic Conference, 2008. INMIC 2008. IEEE International*, pages 565–570. IEEE, 2008.

[87] Jing Zhang. Metamodel-driven model interpreter evolution. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 214–215. ACM, 2005.

APPENDIX A

PARTIAL STATIC SEMANTICS FOR MGRIDML

```
context ControlSchema inv:
self.mgridModelID  <> null
and self.allinstances -> forAll(cs1,cs2| cs1. mgridModelID <> cs2. mgridModelID)
and self.PCC -> size() ==1

context DataSchema inv:
self.mgridModelID  <> null
and self.allinstances -> forAll(ds1,ds2| ds1. mgridModelID <> ds2. mgridModelID)

context StorageDevice inv:
self.capacity * self.upperThreshold >= self.currentCharge
and self.capacity * self.lowerThreshold <= self.currentCharge
and self.allinstances -> forAll (s1,s2|s1<>s2 implies s1.plantE_ID <> s2.plantE_ID)
and self.deviceTypeID -> forAll(b1,b2|b1<>b2 implies b1. deviceTypeID <>b2. deviceTypeID
and self.deviceTypeID -> exists(StorageDeviceType.sdTypeID)

context LoadDevice inv:
self.allinstances -> forAll (s1,s2|s1<>s2 implies s1.plantE_ID <> s2.plantE_ID)
and self.controlledby -> forAll(b1,b2|b1<>b2 implies b1.controllerID <>b2.controllerID
and self.ControlEnum <> null
and self.deviceTypeID <> null

context SourceDevice inv:
self.allinstances -> forAll (s1,s2|s1<>s2 implies s1.plantE_ID <> s2.plantE_ID)
and self.controlledby -> forAll(b1,b2|b1<>b2 implies b1.controllerID <>b2.controllerID
and self.ControlEnum <> null
and self.deviceTypeID <> null

context SmartMeter inv:
self.monitors -> size = 1

context StorageController inv:
self.allinstances -> forAll (s1,s2|s1<>s2 implies s1.controllerID <> s2.controllerID)
and self.contains -> forAll(b1,b2|b1<>b2 implies b1.sdTypeID <>b2.sdTypeID)

context LoadController inv:
self.allinstances -> forAll (s1,s2|s1<>s2 implies s1.controlID <> s2.controlID)
and self.contains -> forAll(b1,b2|b1<>b2 implies b1.ldTypeID <>b2.ldTypeID)

context SourceController inv:
self.allinstances -> forAll (s1,s2|s1<>s2 implies s1.controlID <> s2.controlID)
and self.contains -> forAll(b1,b2|b1<>b2 implies b1.soTypeID <>b2.soTypeID)

context PCC inv:
self.connects<> null
```

APPENDIX B

CONTROL SCRIPTS FOR MGRIDVM

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$_A$

APPENDIX C

CONTROL SCRIPTS FOR CVM

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$_A$ }
16. *requestMediaCmd* := **requestMedia** connectionID$_A$ mediaName$_A$
17. *sendNegTokenCmd* := **sendNegToken** personID$_A$
18. *requestNegTokenCmd* := **requestNegToken** connectionID$_A$

APPENDIX D

MEDIA TRANSFER STATE MACHINE PERSISTENTLY REPRESENTED AS XML

```xml
<sm:StateMachine
xmlns:sm="http://www.stateforge.com/StateMachineJava-v1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.stateforge.com/StateMachineJava-v1
http://www.stateforge.com/xsd/StateMachineJava-v1.xsd">
    <!-- General settings -->
    <settings asynchronous="false"
namespace="com.stateforge.statemachine.examples.BusinessObject">
        <object instance="myBusinessObject"
class="BusinessObject"/>
    </settings>
    <!-- Events -->
    <events>
        <eventSource name="Events">
            <event id="initiateNeg" name="initiateNeg"/>
            <event id="initiateInviteNeg"
name="initiateInviteNeg"/>
            <event id="enableStream" name="enableStream"/>
            <event id="enableStreamRec" name="enableStreamRec"/>
            <event id="disableStream" name="disableStream"/>
            <event id="disableStreamRec"
name="disableStreamRec"/>
            <event id="sendNonStream" name="sendNonStream"/>
            <event id="sendForm" name="sendForm"/>
            <event id="recNonStream" name="recNonStream"/>
            <event id="recForm" name="recForm"/>
            <event id="terminate" name="terminate"/>
        </eventSource>
    </events>
    <!-- States -->
    <state name="Media_Transfer">
        <state name="Initial">
            <transition event="initiateNeg" nextState="Ready"/>
            <transition event="initiateInviteNeg"
nextState="Ready"/>
        </state>
        <state name="Ready">
            <transition event="enableStream"
nextState="StreamEnabled">
                <action>genStreamEnable_Script</action>
            </transition>
            <transition event="enableStreamRec"
nextState="StreamEnabled">
                <action>genStreamEnableRec_Script; UCI.notify()
</action>
            </transition>
            <transition event="sendNonStream" nextState="Ready">
                <action>genNonStreamSend_Script</action>
            </transition>
            <transition event="sendForm" nextState="Ready">
                <action>genSendForm_Script</action>
            </transition>
```

1

```
            <transition event="recNonStream" nextState="Ready">
                <action>UCI.notify()</action>
            </transition>
            <transition event="recForm" nextState="Ready">
                <action>UCI.notify()</action>
            </transition>
            <transition event="terminate" nextState="Final"/>
        </state>
        <state name="StreamEnabled">
            <transition event="enableStream" condition="!
isStreamEnabled" nextState="StreamEnabled">
                <action>genStreamEnable_Script</action>
            </transition>
            <transition event="disableStream"
condition="IsStreamEnabled &amp;&amp; stream_cnt &gt; 1"
nextState="StreamEnabled">
                <action>genStreamDisable_Script</action>
            </transition>
            <transition event="enableStreamRec"
condition="IsStreamEnabled" nextState="StreamEnabled">
                <action>genStreamEnableRec_Script; UCI.notify()
</action>
            </transition>
            <transition event="disableStreamRec"
condition="IsStreamEnabled &amp;&amp; stream_cnt &gt; 1"
nextState="StreamEnabled">
                <action>genStreamDisableRec_Script; UCI.notify
</action>
            </transition>
            <transition event="sendNonStream"
nextState="StreamEnabled">
                <action>genNonStreamSend_Script</action>
            </transition>
            <transition event="sendForm"
nextState="StreamEnabled">
                <action>genSendForm_Script</action>
            </transition>
            <transition event="recNonStream"
nextState="StreamEnabled">
                <action>UCI.notify()</action>
            </transition>
            <transition event="recForm"
nextState="StreamEnabled">
                <action>UCI.notify()</action>
            </transition>
            <transition event="disableStream"
condition="stream_cnt == 1" nextState="Ready">
                <action>genCloseStream_Script</action>
            </transition>
            <transition event="disableStreamRec"
condition="stream_cnt == 1" nextState="Ready">
                <action>UCI.notify()</action>
```

2

```
            </transition>
        </state>
        <state name="Final"/>
    </state>
</sm:StateMachine>
```

3

APPENDIX E

SYNTHESIS ENGINE ECORE

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="SES"
    nsURI="http://www.eclipse.org/2012/SEML" nsPrefix="SES">
  <eClassifiers xsi:type="ecore:EClass" name="SEManager">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="SEID"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="lts"
lowerBound="1" eType="#//LTSArray"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dm" lowerBound="1"
eType="#//DomainManager"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="cm" lowerBound="1"
eType="#//ChangeMapping"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="cmm"
lowerBound="1" eType="#//ControlMetaModel"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dmm"
lowerBound="1" eType="#//DataMetaModel"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="interface"
lowerBound="1"
        eType="#//Interface" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="LTSArray">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="LTSID"
unique="false" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"
        defaultValueLiteral="LTS"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="location"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="format"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ControlMetaModel">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="elementID"
unique="false"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"
defaultValueLiteral="control"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="location"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DataMetaModel">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="elementID"
unique="false"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString" defaultValueLiteral="data"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="location"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DomainManager">
```

APPENDIX F

LTS ECORE

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="lts"
    nsURI="http://www.kermeta.org/lts" nsPrefix="fsm">
  <eClassifiers xsi:type="ecore:EClass" name="LTS">
    <eStructuralFeatures xsi:type="ecore:EReference" name="ownedState"
upperBound="-1"
        eType="#//State" containment="true" eOpposite="#//State/owningLTS"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="initialState"
lowerBound="1"
        eType="#//State"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="finalState"
upperBound="-1"
        eType="#//State"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="activeState"
eType="#//State"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="State">
    <eOperations name="step" eType="#//String">
      <eParameters name="c" eType="#//String"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="owningLTS"
eType="#//LTS"
        eOpposite="#//LTS/ownedState">
      <eAnnotations source="http://www.topcased.org/uuid">
        <details key="uuid" value="114915013380911"/>
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="#//String">
      <eAnnotations source="http://www.topcased.org/uuid">
        <details key="uuid" value="114915013382412"/>
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="outgoingTransition" upperBound="-1"
        eType="#//Transition" containment="true"
eOpposite="#//Transition/source">
      <eAnnotations source="http://www.topcased.org/uuid">
        <details key="uuid" value="114915013382413"/>
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="incomingTransition" upperBound="-1"
        eType="#//Transition" eOpposite="#//Transition/target">
      <eAnnotations source="http://www.topcased.org/uuid">
        <details key="uuid" value="114915013382414"/>
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Transition">
    <eStructuralFeatures xsi:type="ecore:EReference" name="source"
lowerBound="1"
        eType="#//State" eOpposite="#//State/outgoingTransition">
      <eAnnotations source="http://www.topcased.org/uuid">
```

APPENDIX G

CVM SYNTHESIS ENGINE SCHEMA

CMLSEManager.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<SES:SEManager xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SES="http://www.eclipse.org/2012/SEML"
xsi:schemaLocation="http://www.eclipse.org/2012/SEML ../metamodels/semlv3.ecore"
SEID="CVM">
  <lts location="platform:/resource/SE/CMLmodels/LTS/"/>
  <dm>
    <ci/>
    <mc location="JavaPart.JavaComparator"/>
    <rt Data="C:/dsltk/kermeta/SE/CMLmodels/CMLD0.xmi"
Control="C:/dsltk/kermeta/SE/CMLmodels/CMLC0.xmi"/>
  </dm>
  <cm location="platform:/resource/SE/files/CMLchangemapFile.txt"/>
  <cmm File= "C:/dsltk/kermeta/SE/metamodels/CMLcmm.ecore"/>
  <dmm File= "C:/dsltk/kermeta/SE/metamodels/CMLdmm.ecore/>
  <interface UI="C:/dsltk/kermeta/SE/CMLmodels/CMLD1.xmi"
Middleware="C:/dsltk/kermeta/SE/CMLmodels/CMLC1.xmi"/>
</SES:SEManager>
```

Page 1

APPENDIX H

MGRIDVM SYNTHESIS ENGINE SCHEMA

MGRIDMLSEManager.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<SES:SEManager xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SES="http://www.eclipse.org/2012/SEML"
xsi:schemaLocation="http://www.eclipse.org/2012/SEML ../metamodels/semlv3.ecore"
SEID="CVM">
  <lts location="platform:/resource/SE/MGRIDMLmodels/LTS/"/>
  <dm>
    <ci/>
    <mc location="JavaPart.JavaComparator"/>
    <rt Data="C:/dsltk/kermeta/SE/MGRIDMLmodels/CMLD0.xmi"
Control="C:/dsltk/kermeta/SE/MGRIDMLmodels/CMLC0.xmi"/>
  </dm>
  <cm location="platform:/resource/SE/files/CMLchangemapFile.txt"/>
  <cmm File= "C:/dsltk/kermeta/SE/metamodels/MGRIDcmm.ecore"/>
  <dmm File= "C:/dsltk/kermeta/SE/metamodels/MGRIDdmm.ecore/>
  <interface UI="C:/dsltk/kermeta/SE/MGRIDMLmodels/MGRIDD1.xmi"
Middleware="C:/dsltk/kermeta/SE/MGRIDMLmodels/MGRIDC1.xmi"/>
</SES:SEManager>
```

Page 1

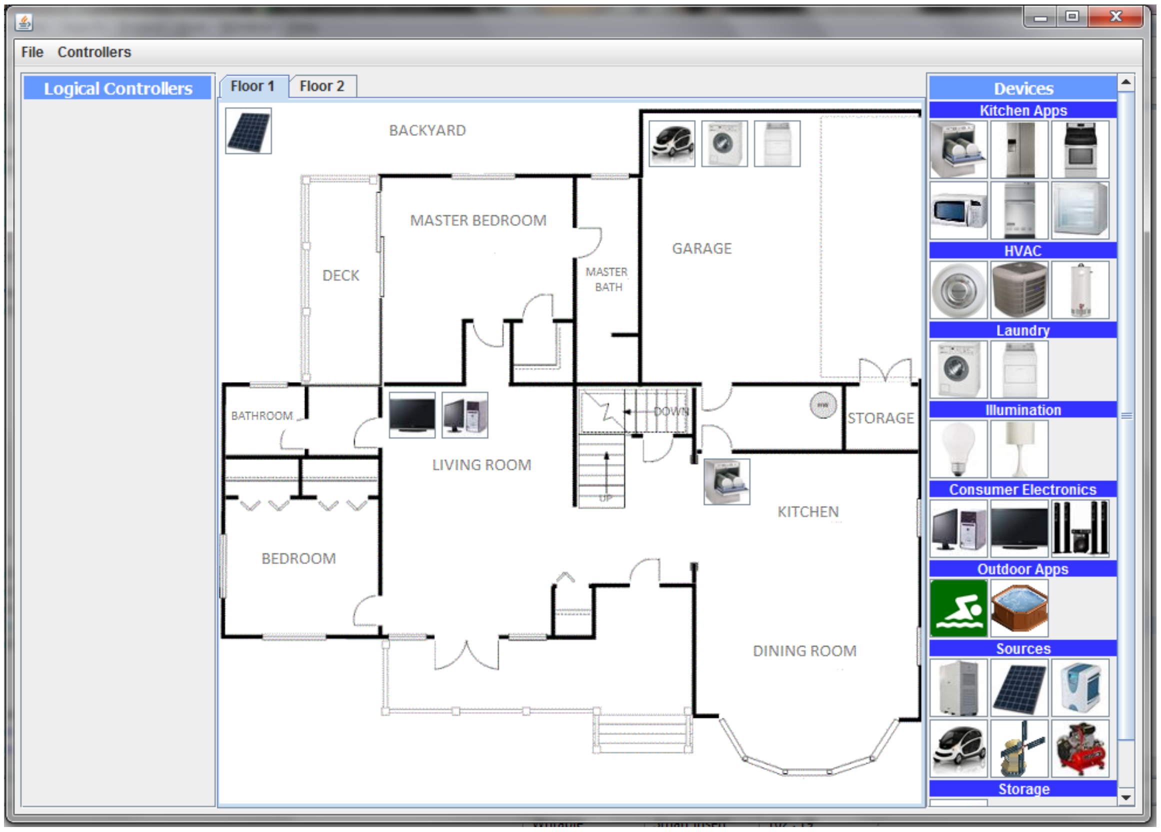
APPENDIX I

CVM CHANGE MAPPING

No.	*CI Models Changes (nodes)*	*Event*	*Explanation*
Source of CI Model: **UCI** (updated model supplied by user)			
1	**added**: connection, device, isAttached (local), person, ...	`intitiateNeg`	initiates a new connection => negotiation of CI
2	**removed**: connection, device, isAttached (local), ...	`removeSelf`	if last connection => terminates all communication
3	**added**: connection, device, isAttached (remote), person	`intitiateReNeg`	initiates a new connection => re-negotiation of CI
4	**removed**: device, isAttached (local), person	`intitiateReNeg`	removes a connection, assuming there are no more remote parties on this connection, and there is still at least one other connection => re-negotiation of CI
5	**removed**: connection, device, isAttached (remote), person	`intitiateReNeg`	removes a connection, assuming there is still at least one other connection => re-negotiation of CI
6	**added**: device, isAttached (remote), person	`intitiateReNeg`	adds a new party to a connection => re-negotiation of CI
7	**removed**: device, isAttached (remote), person	`intitiateReNeg`	removes party from a connection, assuming there are other remote parties on the connection => re-negotiation of CI
8	**added**: medium capability (to device)	`intitiateReNeg`	if the new medium type is not a subtype of an existing medium => re-negotiation of CI
9	**added**: medium type (to connection)	`intitiateReNeg`	if the new medium type is not a subtype of an existing medium => re-negotiation of CI
10	**added**: form type (to connection)	`intitiateReNeg`	if the new form type is not a subtype of an existing medium => re-negotiation of CI
11	**removed**: medium capability (from device)	`intitiateReNeg`	if a medium type is removed from a device may impact capabilities => re-negotiation of CI
12	**removed**: medium type (from connection)	`intitiateReNeg`	if medium type is removed this restricts the types on the connection => re-negotiation of CI
13	**removed**: form type (from connection)	`intitiateReNeg`	if form type is removed this restricts the types on the connection => re-negotiation of CI
Source of CI Model: **UCM** (initiator of negotiation)			
14	**No Change**	`localSameCI`	no change to the CI during negotiation
15	**Any Change**	`localChangeCI`	remote party change CI restarts negotiation
Source of **new** CI Model: **UCM** (non-initiator of the negotiation and CI model **not** seen before from the initiator)			
16	**Change (see 6, 7 – 13)**	`inviteNeg`	invitation for negotiation from the non-initiator
Source of CI Model: **UCM** (non-initiator of negotiation and CI model seen before from the initiator)			
17	**No Change**	`remoteSameCI`	No change to the CI during negotiation
18	**Changes (see 6, 8 – 13)**	`remoteChangeCI`	Change to the CI indicates negotiator needs to restart negotiation.

APPENDIX J

VERSION 1 PROTOTYPE USER INTERFACE



VITA

MARK ALLISON

Fort Lauderdale, Florida

1986-1990	B.S., Computer Science The City University of New York - City College New York, New York
1990-1993	M.S., Information Systems The City University of New York - Graduate School New York, New York
2006-2014	Doctoral Candidate Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Allison, M., Allen, A., Yang, Z., and Clarke, P. (2011)*A Software Engineering Approach to User-Driven Management and Control of Dynamic Energy Systems within the Micro-Smartgrid.* International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 59-64.

Allison, M., Yang, Z., Morris, K., Clarke, P. and Costa, F. (2012) *Managing Smart Microgrid Behavior by Synthesizing Domain-Specific Models.* IEEE High Assurance Software Engineering Conference (HASE), pp. 185-192.

Allison, M. and Joo, S. (2014) *Revisiting Polya's Approach to Foster Problem Solving Skill Development in Software Engineers* IEEE International Conference on Computer Science and Education (ICCSE), *In press.*

Allison, M., Morris, K., Costa, F. and Clarke, P. (2014) *Synthesizing Interpreted Domain-Specific Models to Manage Smart Microgrids* The Journal of Systems and Software Elsevier. *In press.*

Allison, M. and Kendrick L. *Towards an Expressive Embodied Conversational Agent Utilizing Multi-Ethnicity to Augment Solution Focused Therapy.* AAAI Florida Artificial Intelligence Research Society conference (FLAIRS), pp. 332-337.

Clarke, P., Wu, Y., Allen, A., Hernandez, F., and Allison, M. (2012) *Towards Dynamic Semantics for Synthesizing Interpreted DSMLs. Formal and Practical Aspects of Domain Specific Languages: Recent Developments,* IGI-Global Publisher, pp. 242-269.