Synthesizing interpreted domain-specific models to manage smart microgrids

Mark Allison a, *, Karl A. Morris b, Fábio M. Costa c, Peter J. Clarke b

Abstract

The increase in prominence of model-driven software development (MDSD) has placed emphasis on the use of domain-specific modeling languages (DSMLs) during the development process. DSMLs allow for domain concepts to be conceptualized and represented at a high level of abstraction. Currently, most DSML models are converted into high-level languages (HLLs) through a series of model-to-model and/or model-to-text transformations before they are executed. An alternative approach for model execution is the interpretation of models directly without converting them into an HLL. These models are created using interpreted DSMLs (i-DSMLs) and realized using a semantic-rich execution engine or domain-specific virtual machine (DSVM).

In this article we present an approach for model synthesis, the first stage of model interpretation, that separates the domain-specific knowledge (DSK) from the model of execution (MoE). Previous work on model synthesis tightly couples the DSK and MoE, reducing the ability for implementations of the DSVM to be easily reused in other domains. To illustrate how our approach to model synthesis works for i-DSMLs, we have created MGridML, an i-DSML for energy management in smart microgrids, and an MGridVM prototype, the DSVM for MGridML. We evaluated our approach by performing experiments on the model synthesis aspect of MGridVM and comparing the results to a DSVM from the user-centric communication domain.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

Model-driven software development (MDSD) is a paradigm that is gaining much attention in the software development community. MDSD places models at the center of the development process and has the potential to improve productivity (Kelly and Tolvanen, 2008; Stahl et al., 2006). One aspect of MDSD that holds much potential is the use of domain-specific modeling languages (DSMLs) to specify applications for a given domain. The current approach to using DSMLs when developing applications in a given domain is to create a model of the application using the DSML, then transform the model into code in a high-level language, such as Java or C++, to be executed (Kelly and Tolvanen, 2008). This approach is very similar to that used for the text-based domain specific languages (Fowler, 2010; Mernik et al., 2005). That is, converting DSML models to code may involve a series of model-to-model, model-to-text and text-to-text transformations.

One alternative approach to transforming models into a high-level language prior to execution is interpreting the models directly using a specialized execution engine for the specific domain. DSMLs that support the direct execution of models are referred to as interpreted DSMLs or i-DSMLs (Mernik, 2012, chap. 9). The interpretation of i-DSML models requires a semantic-rich execution engine. We refer to the i-DSML execution engine as a domain-specific virtual machine (DSVM). Deng et al. (2008) describe the first manifestation of a DSVM, the communication virtual machine (CVM), used to interpret models created using the communication modeling language (CML) (Wu et al., 2012). CML is used to specify applications in the user-centric communication domain. CVM uses a four-layered architecture where each layer is responsible for reducing the semantic gap between the user-defined model and the low-level services provided to CVM by underlying communication frameworks.

The main layer in the DSVM responsible for determining the behavior of the application based on the semantics of i-DSML models is the synthesis engine (SE). The SE performs model synthesis,
that is, it takes models as input from the user, generates control scripts to be processed by the next lower layer in the DSVM, and maintains the current system state in the form of a runtime model. Our approach to synthesizing i-DSML models is based on changes made to models at runtime (between the SE runtime model and the new user model) and to the current state of the running system. The SE runtime model is at a level of abstraction similar to an i-DSML model, and is causally connected to the views of the system in the other layers of the DSVM as well as the system itself. 

Wu et al. (2012) define the semantics for the synthesis of CML models; however, there is tight coupling between the domain-specific knowledge (DSK) and the model of execution (MoE). DSK represents aspects of the language that are unique to a specific domain and MoE the generic semantics that are reusable across domains.

In this article we define model synthesis where the DSK and the MoE are separated thereby decoupling the domain semantics from the semantics inherent to all DSVMs. To elucidate our approach we have developed an i-DSML for energy management within microgrids, microgrid modeling language (MGridML) and a DSVM prototype for MGridML interpretation, microgrid virtual machine (MGridVM). The domain selected, energy management within smart microgrids, is of particular importance since it addresses requirements brought on by changes in the worldwide energy consumption patterns. MGridML provides end-users a way to define models to manage their energy consumption and MGridVM interprets these models thereby realizing the user’s requirements in the microgrid plant.

This article extends the work presented in Allison et al. (2011, 2012) by providing: (1) the complete definition of MGridML, (2) a comprehensive description of the synthesis process, (3) the complete listing of the control scripts generated by the synthesis process for the microgrid domain, and (4) a description of a study used to evaluate the synthesis process. The synthesis process consists of comparing MGridML models at runtime, analyzing model changes, defining the new behavior of the system based on the model changes and the current system’s state, and generating control scripts to be processed by the next layer in MGridVM. To show the feasibility and practicality of our approach we developed a prototype for MGridVM. We also evaluated the efficiency of the synthesis process with the DSK and MoE decoupled, and compared it to the the synthesis process in the CVM. The major contributions of this article are:

1. A definition of the metamodel of an i-DSML, MGridML, for energy management in microgrid systems.
2. A formulation of the semantics to support the interpretation of MGridML models based on changes to user-defined models at runtime.
3. A design of the synthesis process that separates domain-specific knowledge (DSK) from the model of execution (MoE).
4. A study that (a) evaluates the efficiency to synthesize i-DSML models in a DSVM for the microgrid energy management domain, and (b) compares the synthesis processes in DSVMs for two domains.

The remainder of this paper is organized as follows: Section 2 introduces fundamental concepts of i-DSMLs and DSVMs, and the microgrid energy management domain. Section 3 describes the domain analysis and metamodel for MGridML. Section 4 describes the synthesis of MGridML models and a design that supports the separation of DSK from MoE. Section 5 describes the prototype used to evaluate the synthesis process and Section 6 the results of the experimental study. Section 7 presents the related work and Section 8 concludes the paper.

2. Background

In this section we provide background on i-DSMLs and energy management systems as applied to the microgrid. We offer this synopsis as a basis for the subsequent sections contained in this paper.

2.1. Interpreted domain-specific modeling languages

Domain-specific languages or little languages are so-called as they are usually comparatively smaller than their general purpose counterparts and characteristically possess focused expressive power (Van Deursen and Klint, 1998). There are two major categories of domain-specific languages, namely (1) text-based languages (Mernik et al., 2005), and (2) graphical-based languages, which we refer to as DSMLs. Our focus is on the latter category, DSMLs, which are usually translated to an equivalent textual representation, e.g., some subcategory of XML.

Chen et al. (2005) formally define a DSML as a five-tuple, \(<C, A, M_i, S, M_r>\). The concrete syntax (C) – expresses elements in the abstract syntax and 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 behavior or meaning in some domain, and is usually represented in some formal framework. The syntactic mapping \((M_i : C \rightarrow A)\) – assigns syntactic constructs to elements in the abstract syntax; and a semantics mapping \((M_r : A \rightarrow S)\) – relates the abstract syntactic concepts to the semantic domain. The abstract syntax and the static semantics together represent the metamodel of a DSML and may be specified as \(<A, C, M_r>\).

Our research concerns a class of DSMLs referred to as Interpreted DSMLs or i-DSMLs (Mernik, 2012, chap. 9). Models created using i-DSMLs may either be a control schema (CS) or a data schema (DS). The CS represents configuration and logic structure, akin to a high-level language (HLL) program, and the DS represents the data, akin to data input to an HLL program at runtime. Fig. 1 shows the generic structure of the i-DSML models, where CS and DS are derived from the domain schema, analogous to classes in OO programming, and instances can be created from the CS and DS, analogous to objects in OO programming. This model was adopted based on the concepts of program and data as described by Plotkin (2004), when describing the specification of semantics based on syntactic transformations of programs and operations on discrete data.

Models created using i-DSMLs are directly interpreted by a semantic rich execution engine known as a domain-specific virtual machine (DSVM). The generic DSVM uses a four layered architecture (Mernik, 2012, chap. 9), including: (1) user interface – provides users with a way to declaratively specify their requirements using models; (2) synthesis engine – interprets models at runtime and generates control scripts; (3) middleware – executes the controls scripts and coordinates the delivery of domain services; and (4) broker – provides an independent API to the middleware and interfaces with the underlying frameworks and controllers to realize the services in the domain.

2.2. Microgrid energy management

Microgrids are self contained energy grids which conceptually should monitor their consumption and co-generate their own power (Hatzigiorgiou et al., 2007). As such, microgrids may operate in a grid connected mode or in an off-grid mode. Microgrids are key components in realizing the goals of the smartgrid (Hatzigiorgiou et al., 2007), the conceptual panacea to the world’s ailing energy distribution grids. There has been significant effort in the research community to address energy management within the microgrid.
domain (Dimeas and Hatzigiariou, 2005; Katiraei et al., 2005; Zaidi and Kupzog, 2008). A prominent definition and the one our work is based on is the CERTS microgrid concept (CM) (Lasseter et al., 2002).

Lasseter et al. (2002) state that the structure of CM is based on an aggregation of loads and microsources operating as a single system providing both power and heat. Fig. 2 is a diagram of the CM structure adapted from Marnay and Bailey (2004). On the left of the figure there is a point of common coupling that interfaces with the smartgrid.

The pertinent aspects of the CM are:

- **PCC** – point or point of common coupling represents that point whereby the microgrid may connect or disconnect from the distribution grid.

- **Loads** – devices that consume electrical energy. Depending 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) shedding loads – loads that are abandoned first if there is a power shortage.

- **Sources** – are devices that produce energy and may include microturbines, fuel cells, photovoltaic systems (PV), and wind turbines.

- **Storage** – are elements that store power such as supercapacitors, battery banks, and flywheels (Kroposki et al., 2008).

- **Smartmeter** – the smartmeter constitutes the portal to the advanced metering infrastructure where data is exchanged with the smartgrid via radio frequency 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.

2.3. Illustrative scenario

In this section we present a motivating scenario from the domain of microgrid energy management that will thread the article. 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 following questions: (1) Which order should the devices be turned on/off not to cause an electrical system overload? (2) How long can power be provided by the battery during islanding before the battery goes below the critical level? (3) How to reconfigure the battery after islanding so that it can be replenished from 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 address the aforementioned questions.

3. MGridML – microgrid modeling language

In this section we define the metamodel for MGridML which consists of the abstract syntax and the static semantics. The metamodel is represented using a UML class diagram and OCL statements. 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 2.3.

3.1. Domain analysis

Domain analysis is performed using the feature oriented domain analysis (FODA) approach (Kang et al., 1990) 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 (Mernik et al., 2005). The analysis centered around the CERTS microgrid concept (CM) (Lasseter et al., 2002) whereby we determined the scope, terminology, concept descriptions and feature model for the domain. Fig. 3 shows the feature decomposition of the domain in the form of a feature diagram. 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.

3.2. Metamodel

MGridML is defined in terms of its metamodel comprising its abstract syntax and static semantics. Fig. 4 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, is appropriate since the data portion is expected to change more frequently than the control portion.

The abstract syntax describes an MGridML model (or schema, terminology adopted from CML) as either a control schema (MGridControlSchema), shown on the right side of Fig. 4, or a data schema (MGridDataSchema), left side of the figure. A 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 controllers can either be atomic or composite entities. A composite controller is used to create a hierarchy of logical controllers. The atomic controller represents the different controllers that are connected to the various types of devices. Our
domain devices include sources, loads, storage units and a single 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. ADS contains one microgrid data entity (MGrid) 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 in Section 2.3. 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 article we may use schema when we are referring to an instance, particularly during the synthesis process.

MGridML Models for the Illustrative Scenario: Figs. 5 and 6 show the MGridML models used to realize the change season scenario presented in Section 2.3. Note that these models are abstract models to illustrate how the attributes in the scenario are captured. The model in Fig. 5(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 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 MCGGrid controller representing when islanding mode is activated.

The model in Fig. 5(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 °F; (2) a smart meter SM001, mapping to meter type SMT001, with 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.
Fig. 6 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 data instances. For example, the LoadDeviceType was added to the control instance shown in Fig. 6(b). Two devices were added to the data instance, shown in Fig. 6(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 remain 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.

4. Synthesizing MGridML models

Synthesizing i-DSML models is one of the main contributions of this article and is described in this section. Unlike previous presentations of the synthesis process where the domain-specific knowledge (DSK) is tightly coupled to the model of execution (MoE) (Wu et al., 2012), we present a synthesis process for MGridML models where the DSK and MoE are decoupled. The expectation is that the MoE can be reused in other domains with minimal changes. We first 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 Figs. 5 and 6).

4.1. Overview of synthesis process

The synthesis process 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) (Fernando, 1993) to define the domain-specific behavior for the synthesis process. An LTS is formally defined as a triple \( \langle L, Q, \rightarrow \rangle \), 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:

\[
I_k : I \times S_k \rightarrow O \times S_{k+1}
\]

where \( I \) is the input event alphabet that are events which may signify the receipt of a new user preference model \( U_j \) to be synthesized, or an event (Evt) representing a change in the microgrid plant.

\[
U_j = (C_{j1}, D_{j1}) \quad \text{for } j = 1 \ldots m,
\]

represents the \( j \)th instance pair of MGridML models generated by the user. \( C_{ij} \) is the control instance and \( D_{ij} \) is the data instance. This instance pair may be new or one that could not be satisfied in a previous iteration. In the event that 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 \( R_k \).

\[
R_k = (C_{ik}, D_{ik}) \quad \text{for } k = 0 \ldots n,
\]

represents the \( k \)th runtime MGridML instance pair, the control instance \( C_{ik} \) and data instance \( D_{ik} \).
currently being executed by the synthesis process. These models are causally executed 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.

Fig. 7 shows a high level view of the synthesis process for MGridVM. The inputs to the process, shown on the left of the figure, are either a user-defined model ($U_j$) or an event ($Evt$) 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 the user interface. Below is a description of the main processes shown in Fig. 7:

1. **Model received** – accepts either a user-defined model ($U_j$) or an event ($Evt$) and the current runtime model. The models are validated and then passed on to the model comparator.
2. **Model comparator** – the user-defined model ($U_j$) and the current runtime model ($R_k$) are compared to generate a change list. This is where model differentiations occur.
3. **Change interpreter** – is where either the change list or the middleware event ($Evt$) is processed. Based on a model change, the current state of the LTs, and the entry in the change event map, a controller event is generated. Alternatively, a pending or new event ($Evt$) 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. Additional details of the change event map will be presented in the next section.

4. **Update domain state** – the states of the appropriate LTs are queried and updated, and appropriate control scripts are generated and dispatched.
5. **Dispatcher** – handles the dispatching of the new runtime model ($R_{k+1}$) to the user interface or the internal state of the synthesis environment ($S_{k+1}$), and the control scripts to be executed by the middleware.

6. **Update RT** – updates the current runtime model ($R_k$) with the new runtime model ($R_{k+1}$).

### 4.2 Model of execution

In this section we describe one of the main contributions of this work, that is the separation of the domain-specific knowledge (DSK) from the model of execution (MoE). Unlike the previous work by Wu et al. (2012) 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. Fig. 8 shows the
top-level view of the MoE for the synthesis process using a UML state machine. Although there are some 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 (Uj) or an event signifying there is a state change in the plant. Submachine 2 (ModelReceived) determines the nature of the event and transitions to submachine 3 (ModelComparator) by calling either the compareUM (user model) or compareP (plant) function.

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, neighbors, propsL), where action = (add, delete, change); node is the node in the model that changes; neighbors = neighbors of node in the model, if any; propsL = list of changed controller or plant element properties.

Based on the change (Chg) being processed, Rk, and the states of the controllers, the appropriate event is identified and applied to an 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 Rk 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 (Rk+1) the function updateRT is invoked. This function call results in a transition to the submachine 6 (Dispatch). The control scripts generated by the controllers’ LTSs in submachine 5 are also sent to submachine 6. The processing in submachine 6 includes invoking the functions: (1) updateMUI that updates the user interface for MGridVM with Rk+1, the new runtime model; (2) updateRT updates Rk with Rk+1; and (3) dispatchScriptToMCM that sends scripts to middleware in the MGridVM. After updating the user interface and sending the scripts to middleware the system is returned to the ready state.

Fig. 7 shows the state machine for submachine 5 (UpdateControllers) in Fig. 8 to update the microgrid controllers, which comprises 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 contr.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 Fig. 8. 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 controller shown.

Table 1 shows the LTS for a load controller, see submachine 5.4 in Fig. 9. The columns in the table represent the transition number, source state, target state, event, guard and action. The addition of a load controller consists of two transitions (1 and 2). Transition 1 moves from the Ready state to the AddC state, the event to 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
Table 1
State machine for a LoadController (submachine 5.4 in Fig. 9).

<table>
<thead>
<tr>
<th>T.</th>
<th>Source</th>
<th>Target</th>
<th>Event</th>
<th>Guard</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controllers</td>
<td>Initial</td>
<td>Ready</td>
<td>addC(Chg)</td>
<td>1exist(Chg.node)</td>
<td>create()</td>
</tr>
<tr>
<td></td>
<td>Ready</td>
<td>addC(Chg)</td>
<td>1exist(Chg.node) &amp; &amp; exist(Chg.neighbors)</td>
<td>gen_addLC_cmd(Chg.node)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AddC</td>
<td>Ready</td>
<td>addLC(LD)</td>
<td>exist(Chg.node) &amp; &amp; LDLtype(LD)</td>
<td>gen_addLDT_cmd(Chg.node)</td>
</tr>
<tr>
<td></td>
<td>Ready</td>
<td>removeT(Chg)</td>
<td>exist(Chg.node) &amp; &amp; LDLtype(LD)</td>
<td>gen_removeLDT_cmd(Chg.node)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ready</td>
<td>setLCProp(Chg)</td>
<td>exist(Chg.node)</td>
<td>gen_setLCProp_cmd(Chg.node, Chg.neighbors)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RemC</td>
<td>removeC(Chg)</td>
<td>exist(Chg.node) &amp; &amp; LDLtype(LD)</td>
<td>gen_removeLC_cmd(Chg.node)</td>
<td></td>
</tr>
<tr>
<td>Plant elements</td>
<td>RemC</td>
<td>RemC</td>
<td>removedLC(LC)</td>
<td>!exist(LC) &amp; &amp; LCs &gt; 0</td>
<td>applyToRT(Chg)</td>
</tr>
<tr>
<td></td>
<td>RemC</td>
<td>Final</td>
<td>removedLC(LC)</td>
<td>!exist(LC)</td>
<td>applyToRT(Chg)</td>
</tr>
</tbody>
</table>

LC – load controller; LDT – load device type; LD – load device; PE – plant element; notOP – not operational; Chg = (action, node, neighbors, propsL) is a model change, where action = {add, delete, change}; neighbors – neighbors nodes in model; propsL – list of (attr, value) pairs.

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 “{ }” zero or more occurrences. Rule 1 states that a control script consists of one or more commands and Rule 2 lists the commands. Rule 4 shows the addControllerGroupCmd consisting of addControllerGroup keyword, and attributes contGroupID – id of the controller group and a list of one or more controller ids (controllerID).

4.3. Model comparison

The user model (U) and the runtime model (R) 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 (Ehrig et al., 2006). 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. In this paper we do not go into detail on the model differencing technique used in our model comparison. Langer et al. (2012) provide a detailed explanation of the different types of model differing used in model versioning. The approach we currently use in our work is based on model changes for generic modeling languages (Alanen and Porres, 2002; Lin et al., 2007; Stanek et al., 2008; Xing and Stroulia, 2005).

![Fig. 9. State machine to update controllers.](image-url)
Recall that a change to an MGridML model is captured as a tuple: (action, node, neighbors, props). The action captures the basic graph transformation, e.g., (add, delete, change) where node – the unique id of the node; neighbors – neighbors of node in the model, if any; props – 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 Fig. 5(b) and 6(b) are: (1) delete, LD001, [MDG001], null) – represents the removal of the node for the heater device, shown on the left of Fig. 5(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 Fig. 6(b) bottom right.

Algorithm 1. ModelComparator submachine 3 in Fig. 8
1: modelComparator
2: /* Chgl – list of model changes for CTs and DI’s. Each change 
3: where action = (add, delete, change), 
4: neighbors – neighbors of “node” in model, props 
5: - list of changed properties */
6: function compareUM (Uj, Rj)
7: /* Input: Uj = (Clj, Dj) user model; Rj = (Clj, Dj) 
8: runtime model */
9: 10: ChglClj ← modelDiff (Uj, Rj, Clj)
11: ChglDlj ← modelDiff (Uj, Rj, Dj)
12: /* If there is at least one Cl or Di change */
13: if Chgl.size() > 0 then 
14: chglInterpreter.interpretC(Chgl, Rj)
15: end if
16: end function
17: function comparePLUj, Rj, Evt
18: /* Input: Uj, Rj same as for analyzeUM; Evt is a plant event */
19: if type(Evt) = OPER & Rj.Evt = Evt then 
20: /* Plant element (PE) becomes operational and is 
21: in Uj but not in Rj by default */
22: compareUM(Uj, Rj)
23: else
24: chglInterpreter.interpretE(Evt, Rj)
25: end if
26: end function

Algorithm 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 (Uj, Clj) and the runtime model (Rj, Clj), shown on line 10, by invoking the function modelDiff. A simple approach is used to compare the differences between the data instances Uj, DIj and Rj, DJj, 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 Rj 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 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.

4.4. Change interpretation

Algorithm 2. ChangeInterpreter submachine in Fig. 8
1: changeInterpreter
2: /* State variables: ChglEvtMap – a static table that maps a 
3: node type, model change action and controller state to a 
4: controller event, e.g., (nodeType, action, state, event). 
5: CurContrStates – dynamic table with current controllers 
6: states, e.g., (id, state); Rj, Rj – Updated runtime model; 
7: Chgl – current changes not handled */
8: function interpretC(Chgl, Rj)
9: "Rj ← Rj
10: ChglChgl ← Chgl
11: /* All CI changes are applied before DI changes */
12: for all Chg ∈ CurChgl.CI do
13: id ← Chg.node.getContrID()
14: state ← CurContrStates.getModelId(id)
15: atomicEvt ← ChglEvtMap.getEvt(type(Chg.node), Chg.action, state)
16: Contr ← (id, contrType, atomicEvt)
17: /* Contr – contains the id of the target controller, 
18: the type of controller, and the event for the controller 
19: submachine, see Fig. 8 */
20: UpdateControllers.applyE(Contr, Chgl)
21: end for
22: for all Chg ∈ CurChgl.DI do
23: id ← Chg.node.getPE_ID()
24: state ← CurContrStates.getModelId(id)
25: atomicEvt ← ChglEvtMap.getEvt(type(Chg.node), Chg.action, state)
26: Contr ← (id, contrType, atomicEvt)
27: UpdateControllers.applyE(Contr, Chgl)
28: end for
29: end function
30: function interpretEEvt, Rj /* Evt – is generated from the plant */
31: Contr ← (Evt.id, Evt.controlType, Evt.atomicEvt)
32: UpdateControllers.applyE(Contr, null)
33: end function
34: function applyToRCChg
35: Rj+1 ← applyChange(Chg, Rj+1)
36: ChglChglRemove(Chgl)
37: if CurChgl.size() = 0 then
38: Dispatch.updateRT(Rj+1)
39: end if
40: end function

Algorithm 2 defines the functions used in ChangeInterpreter submachine in Fig. 8, including interpretC, interpretE and applyToRC. The four state variables used in the algorithm include: (a) ChglEvtMap, 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) CurContrStates, a table maintained at runtime with current controllers states, e.g., (id, state); (c) Rj+1 updated runtime model; and (d) CurChgl, the list of model changes that is being updated. An example of an entry in the ChglEvtMap 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.

The function interpretC, line 8, is non-blocking and takes as input the change list Chgl and Rj. By non-blocking we mean that after the function call applyE on line 20 interpretC continues. The first part of the function, lines 12–21, iterates through each CI change (Chgl) and generates the controller atomic event by searching ChglEvtMap 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 Rj. After all the CI changes are processed the DI changes are processed using a similar approach.

The interpretE function, line 30, takes as input an event generated by the plant (Evt) and current runtime model (Rj), 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 applyToRC function, line 34, takes as input a change that has been successfully processed by the UpdateControllers submachine, updates Rj+1 with the change, removes the change from the current list of changes, and calls the updateRT function in the Dispatch
Runtime model = $R_i$
User model = $U_i$

<table>
<thead>
<tr>
<th>Changes</th>
<th>Events</th>
<th>Control Script Generated</th>
</tr>
</thead>
</table>
| **added nodes**(mgrid controller (MCG001), load controller (LC001), pcc controller (PCC001), ... source data type (SDT001)) | *initialMCG createLC createPCC createSC addLC addPCC addSC addDLC addSMT addSDT | initializeMGrid("MCG001")
| | LC_added | addDLC addPCC addSC | |
| | PCC_added | addSDT | |
| | SC_added | LDT_added | |

---

Table 10 illustrates the execution trace of the synthesis process for the change season scenario. The columns in the table from left to right are $R_i$ – the runtime model, $U_i$ – the user model, the changes generated by comparing the $R_i$ and $U_i$ 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 Fig. 10 shows the comparison between models $R_0, CI$ (CI0 shown on the left of the first row) and $U_1, CI$ (CI1) 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 `initializeMGrid("MCG001")` which results in the control script `initializeMGrid("MCG001")` 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 1, it moves into the Ready state. The action (add) in the change results in the event `addLC` being generated (addLC)(Chg) in Table 1 which results in the control script `addLoadController("LC001",...)` being created (a result of gen_addLC_cmd(Chg.node) in Table 1 being invoked) and sent to the middleware layer for processing, as can be seen in the rightmost column of the first row in Fig. 10.

The table shows the other events and the corresponding control scripts that were generated, including the scripts to add the other controllers. The second row of the table shows 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

Fig. 10. Execution trace of the change season scenario for the control instances.

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.

4.5. Synthesis of the illustrative scenario
associated with the controllers that were previously added. For example, the event \texttt{addLDT} corresponds to adding the load device type \textit{LDT001}, which corresponds to the third transition of Table 1. The fourth row shows the events after adding the respective types for the various controllers, for example, the event \texttt{LDT \_added} signifies that the middleware has added the device type \textit{LDT001} to the controller \textit{LC001}. The fifth row shows that the runtime model and the user model, with labels \texttt{C11}, has reached stasis. The second part of the table, below the double lines, shows the steps associated with the changes from the sinter control instance to the Spring control instance, which involves adding the load data type \textit{LDT002}.

5. MGridVM prototype

To demonstrate the efficacy and applicability of this approach, we describe our prototype which creates and realizes MGridML models using the DVM architecture consisting of four layers representing different levels of abstraction. The high-level design is similar to the design described by Deng et al. (2008). 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.

5.1. High-level design

Fig. 11 shows the four layered architecture for the MGridVM, based on the architecture created by Deng et al. (2008) for the communication virtual machine (CVM). The layers of the MGridVM are described as follows:

- **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 4. The MSE takes as input an MGridML model and transforms it into 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 scripts 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 the appropriate macros are loaded and executed. These macros are execution components selected based on the scripts to be interpreted and on the state of the middleware. Note that the macros are executed immediately after they are selected. The services managed by the middleware include: 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. (Morris et al., 2012).

- **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 had successes at the 120V level using Zigbee control (Kinney, 2003). Sousa et al. (2012) have provided a metamodel based implementation of this layer. 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 of 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. Fig. 12 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 – 1000 mA 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.

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.

Fig. 13 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 be dequeued and processed. The text window in the center echos actions taken by the MSE, the buttons below this window simulate various changes to the model independent of the change list. The three text windows on the right show the MCM events, the current runtime model for the control instance and the current runtime model for the data instance.

5.2. 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 Fig. 14. The packages reflect the structure of the synthesis process outlined in Fig. 7. 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 (EMF Team, 2013a), specifically EMF Core and EMF Compare (Brun and Pierantonio, 2008). 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 packages 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 Fig. 14 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 mse::runtimeModel package contains the classes to represent the
runtime model in the MSE, including the attributes and operations required to update the runtime model.

Fig. 15 shows the packages contained in the mse::domainEntities package, that is, the packages containing the DSK entities. This is where the separation of concerns take place. There are two classes in the figure: DomainManager, which coordinates the activities of the packages, and 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, mse::controlScripts. The mgridEvents package contains the classes that represent the table for the change event map (see Algorithm 2) and the events generated for the LTSs. The mgridMLMetamodel package contains the classes that define the metamodel for MGridML. The metamodel is used by the EMF compare package (EMF Team, 2013a) when comparing MgridML models.

6. 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 MGridML models to be processed during synthesis.

**Objective 2:** Compare the performance of processing MGridML and CML models in MGridVM and CVM, respectively.

**Objective 3:** Determine the level of reusability and impact on effort in developing a DSVM SE when the domain-specific knowledge (DSK) is separated from the model of execution (MoE).

6.1. 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. All experiments were performed in the Eclipse environment (EMF Team, 2013b).

**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. During the experimentation we varied M, the number of model nodes, between 3 and 57, and N, the number of controllers between 1 and 28. The controllers in each scenario are load controllers, each connected to one load device type. The initial number of nodes is 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 compared the total running times for the SEs in the MGridVM and the CVM. We inspected both prototypes and ascertained that the SEs are candidates for comparison as: (1) the inputs to both SEs are models and events of comparable magnitude; (2) the outputs of both SEs are control scripts; and (3) the implementation platforms are very similar. The model generation automation use in Objective (1) was specifically designed to create test models with content and relationship criteria which allows for negligible performance differences by the model comparison component. By isolating the SE of both prototypes with drivers and stubs, we were able to test the SEs to identify any differences in the functionality. However, we found no difference in functionality in the respective domains, only differences in performance times.

A desired quality of a modular architecture is loose interconnectivity between modules. Addressing the DSK as a primary aspect of the SE architecture has led us to reduce its coupling from the MoE. The value of loose coupling in the maintainability and robustness of a software product is well established; tight coupling between classes leads to reduced understanding, change dependency, and the proliferation of errors (Abreu and Goul ao, 2001; Harrison et al., 1998). By reducing the coupling in the architecture we are able to support swapping domain modules with minimum effort.

To evaluate Objective 3 we analyze the code for two versions of DSVMs (CVM and MGridVM) using a set of coupling metrics and static code metrics, to provide some idea of the reusability of the MoE; and the development time in hours for the effort. The earlier version of the SEs (v1) for the CVM and MGridVM were developed prior to decoupling the DSK and MoE, and the current version (v2) after the DSK and MoE were decoupled. The current version of the SE for each DSVM was developed using Kermeta (Triskell Team, 2013), a meta-programming environment based on an object-oriented DSL optimized for metamodel engineering. Kermeta supports model and metamodel prototyping and simulation, and aspect weaving, among other features. The DSK was represented as a set of models in Kermeta, only the MoE was coded using...
Java. We are currently working on formalizing this approach and will present it in future work.

Data collection: To collect our data regarding execution time analysis for MSE we employed the eclipse testing and performance tooling platform (TPTP) (Yang and Jiang, 2007) by instrumenting the MSE code. We collected runtime data for the main classes in the MSE: Parser, ModelCompare, ChangeInterpreter, which are part of the mae::modelProcessors package; and RuntimeUpdate which is part of the mae::runtimeModelPackage, see Fig. 14. The microgrid scenarios, represented as MGridML models, range in size from 3 to 57 nodes (1–28 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. (2012). For those experiments Eclipse TPTP was also used as the tool of choice to obtain the runtime results. The data for the SE in the CVM reported by (Wu et al., 2012) 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, isAttached and participant, as shown in Fig. 5 in (Wu et al., 2012). 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.

The data collected for the two versions of the DSVMs (CVM and MGridVM) included: coupling metrics (Abreu and Goul ao, 2001), described below: static metrics – number of classes, number of methods and SLOC; and the development effort – number of hours. The coupling metrics are:

- **Outbound extra-package dependencies (OEP)** – Classes of other packages that this class depends on.
- **inbound intra-package method dependencies (IPM)** – Methods and fields in other classes of the same package that depend on this class.
- **Inbound extra-package method dependencies (IEMP)** – Methods in other packages that depend on this class.

The coupling and static metrics were obtained by analyzing the class files associated with the MoE in the different versions of the DSVMs using dependency finder (Tessier, 2010).

6.2. Results

The results of our performance evaluation for Objective 1 is shown in Table 2. The main classes analyzed, shown across the top of the table, are Parser, ModelCompare, 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). Columns 3–6 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 first row in the table shows the scenario with 1 controller and 3 model nodes, which 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 2 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. Fig. 16 shows the graphical representation of the data in Table 2. 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, which 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.

In evaluating Objective 2 of the study we show the execution times comparing the synthesis of several scenarios in both the microgrid energy management (MGridVM SE) and user-centric communication domains (CVM SE). Table 3 shows the comparison of the execution times for the CVM and MGridVM SEs 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. (Wu et al., 2012)) and the right section data for the MGridVM. The columns in the left section represents the number of participants in a communication (N partes), 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 are extrapolated from the values shown in Table 2.

Fig. 17 shows the graphical representation of the execution times in Table 3. 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 results further in the next section.

The data collected for Objective 3 include the static metrics of the code, development time and coupling measures. Table 4 shows the source line of code (SLOC), number of classes, and number of
methods for the synthesis engines in MGridVM and CVM. The table also shows the development time to create both versions of the SEs. 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. Note that for the SEs v2 the values are shown as the sum of the MoE and DSK. For example, the SE in the MGridVM consists of 314 SLOC for the MoE and 718 SLOC for the DSK, see first row of the table.

The synthesis engine in the MGridVM (v1) is larger in size when compared to the synthesis engine in the CVM (v1). 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 (EMF Team, 2013a), providing a more generic model comparator. Note that the v2 SEs 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 to perform full code generation in future work.

The last row of Table 4 shows the development time to create both versions of the SEs. The data was collected based on the logs we kept for the development time for each DSVM. For the v2 SEs we separated the time to develop the MoE and DSK in Kermeta. Note that the time to develop the v2 SEs using Kermeta is approximately 50% of the time to develop the v1 SEs.

The coupling measures for Objective 3 are shown in Table 5. Top part of the table shows the static code metrics for the classes in SE that represent the MoE. In the SE v1 the MoE is captured in the parts of the code that perform parsing of the models, model comparator, change interpreter, and update of the runtime model. Three coupling metrics are shown including: OEP - classes of other packages that this class depends on; IEPM - methods and fields in other classes of the same package that depend on this class; and IEPM - methods in other packages that depend on this class. Based on the results the MoE for v2 of the SE has the lowest total coupling measure. This would suggest that the MoE 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.

### 6.3 Discussion

The results for Objective 1, Table 2 and Fig. 16, show that as the number of nodes in an MGridML model increases the execution times for the synthesis process increases linearly. The most expensive aspect of model synthesis is 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 2, 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.

Objective 2 focused on evaluating model synthesis in two domains, one in a distributed environment (CVM) and the other in a centralized environment (MGridVM). Table 3 and Fig. 17 show that there is clearly a correlation between the execution times from MGridVM and CVM. The regression analysis of the CVM time (Columns 5 and 6 in Table 3) 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 the increased number of LTs used to capture the DSK for the synthesis process. The CVM has two main LTs (negotiation and media transfer) while there are four LTs for the controllers in the microgrid (load, storage, source, point-of-common coupling) which accounts for an increase in the number of classes. 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 regarding the models that can be processed in the synthesis engines for DSVMs.

The results obtained for Objective 3 shown in Tables 4 and 5 suggest that when the DSK is separated from the MoE there is a decrease in the time to develop the SE in the user-centric communication and microgrid energy management domains. In addition the MoE was reusable in developing the SEs for the MGridVM and CVM in the Kermeta meta-programming environment (Triskell Team, 2013). Although we did not generate the Java code for the SEs at this stage, Kermeta currently has the facility to generate Java code and we will be working in this on the future. There is also evidence to support reusability when the DSK is separated from the MoE. This is reflected in the low coupling numbers shown in Table 5 for v2 of the SE. The numbers show that coupling in the SE v2 is reduced on average by 77% when compared to the coupling in the SE v1 for the MGridVM and CVM. We acknowledge the fact that the analysis on the v1 SEs was done on complete Java implementations, whereas for the v2 SEs it was done on fully functioning SEs but the DSK was represented as models, e.g., using the Ecore format.

6.4. Threats to validity

In the following paragraphs we consider the threats to validity (Wohlin et al., 2000) for the experiments performed for Objectives 1, 2 and 3. The threats related to the experiments for Objective 1 include: the type and size of the models used during the synthesis process. 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 LTs, which may not reflect the true times for general scenarios. We did not use very large models in our experiments, 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 of model synthesis for some domains.

The threats related to the experiments for Objective 2 include: using similar hardware and software platforms, and automatic generation of domain-specific models. The CVM results were obtained from past experiments performed by Wu et al. (Wu et al., 2012) and no specifics were given for the hardware and software platforms. Not having the same hardware and software platforms for the MGridVM could impact the comparison of the performance results. The data in Tables 2 and 3 show that the number of nodes in the MGridML and CML models form 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.

The main threat related to Objective 3 is comparing the SEs of MGridVM v1 and CVM v1 to the SEs generated using Kermeta (Triskell Team, 2013). The development approach for the v1 SEs was to build the artifacts from ground up, while for the v2 SEs a meta-programming environment was used that allowed the SE artifacts to be partially generated. Since the Java code for the v2 SEs were not fully generated it was difficult to identify all the dependencies that impacted the coupling measures used. On the other hand completely separating the DSK from the MoE provided us with the opportunity to used a meta-programming environment to quickly generate the SE for the MGridVM and CVM. Finally, based on our experiments it is difficult to determine the impact on reusability and effort that our approach would have in a totally new domain. This work is planned for the future and we are currently developing an i-DSML for cloud provisioning of resources.

7. Related work

In this section we describe three specific areas of related work including model synthesis and reconciliation, synthesis in the communication virtual machine (CVM) and the use of domain-specific languages to manage microgrids.

7.1. Model synthesis

The term model synthesis has been used in several contexts in the domain-specific modeling community. The most common use of term model synthesis is the translation of a higher-level, more abstract, model into a low-level, more concrete, model (Mens and Gorp, 2006). 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 (Agrawal et al., 2006). Our work uses some of the basic concepts used in graph transformations such as comparing graphs to identify model differences (Alanen and Porres, 2002; Lin et al., 2007; Stanek et al., 2008; Xing and Stroulia, 2005), 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.

Langer et al. (2012) describe an extension to model versioning that use signifiers thereby generating merged models of superior quality to other current approaches. Currently we do not use this approach in our work. However, since their approach more
accurately handles model versioning we plan to incorporate their approach in our future work. We plan to log the different models used during the execution of an application and would therefore need a way to accurately record the changes made to the models at runtime. We also plan to re-use the models logged during synthesis in the event we need to rollback the system to a safe state.

Bencomo et al. (2013) described 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 and Vangheluwe (2012) described 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 Fig. 11.

Edwards et al. (2012) described 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.

7.2. CML and CVM

The work most closely related to the work presented in this paper is that by Wu et al. (2012). Wu et al. provide a comprehensive description of the Communication Modeling Language (CML), the first i-DSML, and the aspects of the Communication Virtual Machine (CVM), the first DSVM, related to model synthesis. CML is used to describe the requirements of user-centric communication applications which are inherently distributed applications. As a result all parties in the communication must have CVMs running in order to participate in a communication activity. Fig. 18, proposed by Deng et al. (2008), shows the high-level architecture of the CVM and the virtual communication between the various layers. Unlike the CVM, MGridVM is a centralized control system at this point, however we
envision multiple instances communicating to barter energy in a market environment.

Many of the underpinning concepts used in the synthesis process presented in this article are presented by Wu et al., who described the first iteration of the synthesis process based on changes to models at runtime. The main differences between the synthesis process by Wu et al. (2012) and our work are: (1) the domain-specific knowledge (DSK) and model of execution (MoE) are tightly coupled; and (2) there is a lack of detail with respect to model comparison and change interpretation during synthesis. We also show the results of a comparative study of the synthesis process for the CVM and MGridVM.

It is important to reduce the coupling between the DSK and MoE for this work to have broad reaching applications. The next stage of the work will be to define a generic model of execution (GMoE) and a representation of the DSK that can be used to instantiate a DSVM synthesis engine for any given domain. We present additional details on model comparison, in particular we specify the attributes of a change and how they are used during model interpretation, see Sections 4.3 and 4.4. In Wu et al. (Wu et al., 2012) the algorithms to perform model comparison and change interpretation are merged together and focuses on analyzing the control instance and the data instance, again showing tight coupling with the DSK.

7.3. Microgrid energy management

There are few domain specific languages (DSLs) directly related to energy management in microgrids. One such DSL is the Building Energy Management Language (BEML), developed by Lavelle (2010), that allows users to manage their microgrids by writing low-level functions. BEML code is compiled into Erlang and executed by an XMPP client. The XMPP client is connected to a XMPP sever via the web allowing the user to configure the Microgrid via the web. Unlike the BEML approach, MGridML models are directly executed by a virtual machine, MGridVM, and are causally connected to the state of the plant. These runtime models are presented to the user in a graphical user-friendly manner that allows the model representing energy management to be updated at runtime.

Most of the work on DSLs that include some energy management is related to home automation. Habitation, by Jiménez et al. (2009), is a DSL 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 only, while our work combines and manages all entities in the microgrid plant. Additionally, Habitation uses a code generation methodology, while MGridVM uses a runtime model interpretation technique to support dynamic reconfiguration of the microgrid.

8. Concluding remarks

In this article, we presented our approach for synthesizing interpreted DSMLs (i-DSMLs) models by facilitating the separation of domain specific knowledge (DSK) and the Model of execution (MoE). Models created with i-DSMLs are directly executed by a domain-specific virtual machine (DSVM) without first being transformed into another language prior to execution. The decoupling of the DSK from the MoE paves the way for the efficient instantiation of the model synthesis component (synthesis engine) for DSVMs in other domains.

We demonstrated the efficacy of our approach using an i-DSML for the smart microgrid management domain (MGridML), and a prototype of the DSVM, MGridVM, where the DSK and the MoE are loosely coupled in the synthesis engine. We also 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 a DSVM in another domain.
Appendix A. Static semantics

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.idTypeID <> b2.idTypeID

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 | addStorageControllerCmd | addSourceControllerCmd | addPCCControllerCmd | removeTypeCmd | addLoadDeviceCmd | addStorageDeviceCmd | addSourceCmd | addSmartMeterCmd | setLCPropertyCmd | setLegacyMeterCmd | removeEntityCmd | setPropertyCmd | requestPropertyCmd

3. initializeMGridCmd := initializeMGrid mgridID
4. addGroupControllerCmd := addGroupController contGroupIdA controllerIDa [controllerIDb]
5. removeGroupControllerCmd := removeGroupController contGroupIdA
6. addLoadControllerCmd := addLoadController controllerIDa name|cardinality|priority|controllerIDa
7. addStorageControllerCmd := addStorageController controllerIDa name|cardinality|chargeStatusa|typeIDa
8. addSourceControllerCmd := addSourceController controllerIDa name|cardinality|groupAction|typeIDa
9. addPCCControllerCmd := addPCCController controllerIDa name|cardinality|groupAction|typeIDa
10. removeControllerCmd := removeController controllerIDa
11. addLoadDeviceCmd := addLoadDevice deviceTypeIDa typenamea|criticala|usagea|controllerIDa
12. addStorageDeviceCmd := addStorageDevice deviceTypeIDa typenamea|lowerWattagea|upperWattagea|controllerIDa
13. addSourceCmd := addSource sourceIDa typenamea|wattagea|onDemanda|charginga|chargeTa (attribute1, value1, attribute2, value2)
14. addSmartMeterCmd := addSmartMeter meterIDa deviceTypeIDa tariff,usagea
15. addLegacyMeterCmd := addLegacyMeter meterIDa deviceTypeIDa
16. removeEntityCmd := removeEntity entityIDa
17. setLCPropertyCmd := setLCProperty deviceIDa attributea, valuea
18. setDevicePropertyCmd := setDeviceProperty deviceIDa attributea, valuea
19. requestPropertyCmd := requestProperty deviceIDa attributea, valuea

References


Mark Allison received his BSc and MSc degrees from the City College of New York in computer science and information systems respectively. He received his PhD from Florida International University with research interests in software engineering, more specifically in model-driven software development, energy management systems and merging artificial intelligence methods. Mark has worked in plant engineering and facilities management within the telecommunications industry for close to a decade and is currently an assistant professor of computer science with the University of Michigan at Flint. He is an active member in MENSA International and the Association for Computing Machinery (ACM).

Karl Morris is currently a PhD candidate in computer science at the Florida International University working in the area of software engineering with an emphasis on model driven software development and middleware. Karl received his BSc and MSc in computer science from Florida International University. Karl has worked as a software developer, database administrator and project manager for various private and public sector entities.

Fabio M. Costa is an associate professor at the Institute of Informatics of the Federal University of Goiás (UFG), Brazil. He got his PhD in computer science from Lancaster University, UK, in 2001, and his MSc and BSc, both also in computer science, from UNICAMP (1995) and UFG (1992), respectively. His current research interests are in the area of model-driven adaptive systems and middleware, based on his past contributions to adaptive and reflective middleware research. Fabio is a member of the ACM and the Brazilian Computer Society.

Peter J. Clarke received his BSc degree in computer science and mathematics from the University of the West Indies (Cave Hill) in 1987. MSc degree from SUNY Binghamton University in 1996 and PhD in computer science from Clemson University in 2003. His research interests are in the areas of domain-specific languages, model-driven software development, software testing, and computer science education. He is currently an associate professor of computing science at Florida International University. He is a member of the ACM (SIGSOFT, SIGCSE, and SIGAPP); IEEE Computer Society; and the Association for Software Testing (AST).