NAOMI – An Experimental Platform for Multi–modeling

Trip Denton, Edward Jones, Srini Srinivasan, Ken Owens, and Richard W. Buskens

Lockheed Martin Advanced Technology Laboratories, Cherry Hill NJ 08002
{ldenton,ejones,ssriniva,kowens,rbuskens}@atl.lmco.com

Abstract. Domain-specific modeling languages (DSMLs) are designed to provide precise abstractions of domain-specific constructs. However, models for complex systems typically do not fit neatly within a single domain and capturing all important aspects of such a system requires developing multiple models using different DSMLs. Combining these models into multi-models presents difficult challenges, most importantly those of integrating the various models and keeping both the models and their associated data synchronized. To this end, we present NAOMI, an experimental platform for enabling multiple models, developed in different DSMLs, to work together. NAOMI analyzes model dependencies to determine the impact of changes to one model on other dependent models and coordinates the propagation of necessary model changes. NAOMI also serves as a useful testbed for exploring how diverse modeling paradigms can be combined.

1 Introduction

Incomplete system specifications can lead to serious problems that do not manifest themselves until very late in the development lifecycle - when the costs to repair such problems may be astronomical. Model-driven design techniques can expose design flaws early in the project lifecycle, when the cost of a design change is minimal.

Invariably, multiple models are needed to comprehensively model all aspects of a system, encompassing system structure and behavior, and possibly other aspects as well (e.g., cost). As shown by Hessellund, Czarnecki, and Wasowski [1] large and complex systems can be accurately abstracted by the use of multiple domain-specific modeling languages (DSMLs). In fact, as noted by Greenfield, Short, Cook, and Kent [2], general-purpose modeling languages (e.g. UML, SysML) often lack the semantic precision required to model all aspects of a system.

The SysML and UML communities attempt to address this issue via domain specific extensions (e.g., the UML profile for Modeling and Analysis of Real-time and Embedded systems(MARTE) [3,4] and by providing explicit capabilities to describe dependencies on models implemented in other modeling languages (e.g., through the use of parametrics in SysML [5]).

Our vision of future model-centric systems engineering and development does not require or depend on the existence of a general-purpose modeling language. Instead, modelers construct multiple models using a suite of diverse domain-specific modeling languages (DSMLs), where each DSML allows specific aspects of a system to be precisely modeled. We refer to the interrelated compositions of the various models as

K. Czarnecki et al. (Eds.): MoDELS 2008, LNCS 5301, pp. 143–157, 2008.

multi-models. An overarching challenge, described in more detail in subsequent sections of this paper, is the complexity of integrating the various models in a cohesive manner. The root causes of this challenge are the diversity of the various modeling paradigms in both specification and implementation, the need for these models to share information about the system under design, and the need for model changes to be appropriately reflected in dependent models.

In this paper, we describe NAOMI, an experimental platform for enabling multiple models, developed using different DSMLs, to work together. NAOMI allows modelers to use any modeling languages they desire, and as new modeling languages and tools are developed, they can be integrated into the NAOMI infrastructure. NAOMI provides a standard way for linking the multiple facets of a system through a structured mechanism for specifying shared attributes of the system under design. Our approach is general and is not specific to any modeling paradigm, thus it may be used wherever multiple models need to be combined into a cohesive whole. Currently, the framework is targeted toward tools for the analysis of system behavior.

2 Key Multi-modeling Challenges

Successful integration of multiple interdependent models, where each model captures different aspects of a complete system specification, poses several difficult research challenges. In this section, we describe some of these challenges and derive a set of requirements that a generalized multi-model platform must meet to address these challenges.

2.1 Challenge 1: Capturing Multi-model Interdependencies

Previous work by Nuseibeh, Kramer, and Finkelstein [6], demonstrates the difficulty of capturing and expressing model interdependencies. In a multi-model, interdependencies can be identified by a sharing or exchange of data between models. As n integrated models may have an arbitrary amount of dependencies, for a multi-model composed of n constituent models, the number of dependencies can not necessarily be asymptotically bound with a polynomial function in terms of n. Therefore, as the number of models, the complexity of their interaction, and/or the size of the system being modeled increases, it becomes extremely difficult to manually keep track of these interdependencies. Even more difficult is tracking and analyzing the implications of these dependencies, both of which are necessary and challenging aspects of multi-model integration. As we will later demonstrate, in order to maintain consistency, it is important that these dependencies be explicit.

2.2 Challenge 2: Maintaining Multi-model Consistency

Hessellund et al. [1] demonstrated the need for maintaining consistency when modeling with multiple DSMLs. In a *consistent* multi-model, we believe there can be no unpropagated model changes. When a model in a consistent multi-model is modified, changes to the model must be correctly propagated to dependent models. Such changes

may propagate through the entire set of models in the multi-model and even back to the original model itself. Not until the propagated changes converge so that no more change propagation is necessary, can we say that the multi-model is *consistent*. To reach multi-model consistency, model changes must be propagated in order with respect to their interdependencies. However, it may be difficult to determine such an ordering when the dependencies are cyclic. Even if a suitable ordering of change propagation can be determined, there is no way today to assess if the models will converge.

Additionally, constraints placed on models (e.g., performance or reliability bounds) further complicate the problem. Some changes may violate constraints and our notion of consistency must be extended to ensure that no constraints have been violated.

There is the practical consideration that multiple modelers must be allowed to make changes to their models simultaneously. Thus, propagation of model changes must be performed across compatible versions of the various models. Should particular changes not yield a consistent multi-model, rollback of the changes, again across compatible versions of the models, must be possible. An additional practical consideration is that the modeling tools themselves may evolve over time, in which the correct versions of the tools associated with the models must also be maintained to ensure that a consistent multi-model can be reconstructed.

2.3 Challenge 3: Semantic Precision of Inter-model Data Exchange

The fact that models are interdependent necessitates that they share or exchange data. The semantics of the data must be precisely known and understood by all models participating in the exchange. Achieving this precision is challenging given multiple models, expressed in different DSMLs. Model and meta-model transformation based approaches can allow for the preservation of semantics when data is exchanged between models [7,8], but this requires that the models either be described in languages derived from the same meta-model, or that experts specify the relationships between elements of the disparate meta-models. While syntactic transformation between languages may be possible and even potentially automatable, ensuring and/or preserving semantic correctness remains challenging.

2.4 Requirements

To address the challenges described in Sections 2.1, 2.2 and 2.3, a multi-modeling system must satisfy the following requirements:

1. The system must allow data exchange between models.
2. The system must support a means of specifying interdependencies between models.
3. The system must support a means of propagating changes from one model to all dependent models.
4. The system must be able to determine an ordering for propagation of model changes consistent with interdependencies among models.
5. The system must be able to identify when multi-models are consistent.
6. The system must support simultaneous updates of multiple models.
7. The system must be capable of supporting all of the above independently of modeling language or modeling tool.

8. The system must allow for the set of models and tools to vary over time.
9. The system must allow for multiple versions of models (and tools).
10. The system must support the ability to rollback to earlier versions of the models and tools.
11. The system must allow for transformations of data during inter-model exchanges.
12. The system must assure and enforce semantic correctness in inter-model data exchanges. (not addressed in this paper)

Requirement 1 follows from Challenges 1 and 3, requirement 2 follows from Challenge 1, requirements 3-11 follow from Challenge 2, and requirement 12 follows from Challenge 3. We present the design of NAOMI and show how it meets requirements 1 through 11 above. Requirement 12 will be addressed in future work.

3 NAOMI

NAOMI is an experimental platform for multi-model integration. NAOMI places no restrictions on the modeling languages or tools used to develop a multi-model of a system. It allows modelers to specify the models used to describe a system, along with their inter-dependencies, and uses this information to propagate changes to models.

Figure 1 shows a high level overview of NAOMI. Modelers edit models in DSMLs and use the Multi-Model Manager to construct the multi-model from individual models. Connectors isolate DSML specific interface code and interact with the Multi-Model Manager and the Sandbox, which provides modelers with a local copy of the Multi-Model Repository that holds the multi-model's artifacts under version control. The Execution Automation Engine determines an ordering of execution based on dependencies in the multi-model and orchestrates execution of the multi-model. The functionality provided by the system includes, model data exchange, constraint management, change propagation, model execution, and version management.

3.1 NAOMI Multi-model Repository

Various data items are stored in the Multi-Model Repository. We use the term *artifacts* to describe entities that define the multi-model. NAOMI artifacts include models, the

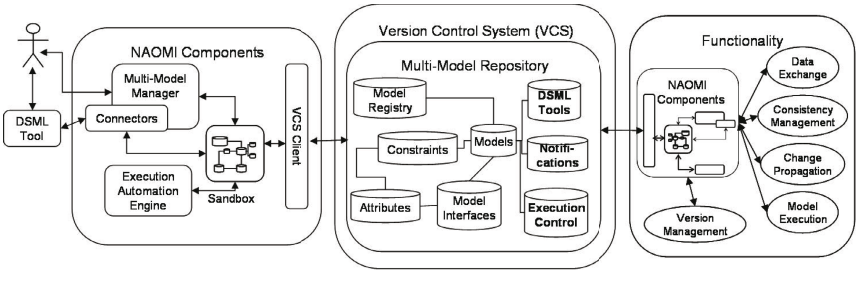

Fig. 1. High level overview of NAOMI, modelers edit models in domain specific modeling languages (DSMLs) and interact with NAOMI components which expose the multi-modeling functionality

model registry, model editing tools, interfaces, attributes, attribute constraints, and notifications. All of the NAOMI artifacts are stored in a *repository* that is under version control. This allows the entire multi-model to be reconstructed at any point during its evolution and for distributed collaboration via modifications to artifacts in the repository.

Model Attributes and Interfaces. In our notion of models, they can be grouped into two not necessarily exclusive groups, *Specification* and *Analytical* Models. *Specification* models help to define the system being modeled, for example UML may define the class hierarchy, valid system states or operational sequence diagrams. *Analytical* models, analyze aspects of the defined system, for example RT-Maude [9] may evaluate the safety parameters of the system, given a specific, state transition and component failure rate specification. Our concept of a model is a mapping written in some language, L_i, over a set of inputs, that produces a set of outputs. A model, M_j, is associated with a set of *attributes* that are inputs $I_j = \{a_1, a_2, \ldots, a_n\}$, which may be empty, and a set of attributes that are outputs $O_j = \{a'_1, a'_2, \ldots, a'_{n'}\}$, which may also be empty. We can represent the model, M_j by the tuple $M_j(L_i, I_j, O_j)$. This tuple is the interface that the model exposes to the system by adding it to the repository. A multi-model, M, can be defined as a set of models $M = \{M_1, M_2, \ldots, M_m\}$. To synchronize execution of the multi-model, we require that an attribute be owned by a single model and can only be written by the model that owns it. That is, if $a_i \in O_j$, then $a_i \notin O_n \forall O_{n \neq j} \in M$.

This input/output view of models works well where specification type models provide input for analytical models (sometimes the entire model itself serves as input for analysis), and the output of analytical models is used either as input to other analytical models or to further refine another specification model. There are cases however, where specification models only share parts of their specification. These shared parts, rather than being used "wholesale", sometimes only constrain or imply the existence of other specifications in other models. For example, in a UML model, state chart transition events may be limited to the methods defined in a specific class diagram. In the absence of specific model-to-model transformation tools, sharing this type of information in this input-output manner becomes difficult. In section 7 we discuss our plans for addressing this issue in our future work.

Model Registry. The model registry is used to determine membership in the multi-model; it is a simple list that enumerates the models in the multi-model. If a model is removed from the registry, the artifacts that correspond to that model remain in the repository, but that model will not participate in the multi-model.

Constraints. Models can have constraints associated with their inputs and outputs. These *attribute constraints* can be thought of as triggers on the value of the attribute. If a model has a constraint on input attribute a_i, the model is notified when the model that owns a_i outputs a value that violates this constraint. Output constraints are used by a model to ensure that the attributes written are valid from the point of view of the model that writes them.

Notifications. Notifications are messages sent to models to inform them of interesting events. For example, when a modeler edits a model which generates a change to an output attribute that violates some constraint, a notification is sent to the affected models.

Sandboxes. Modelers interact with NAOMI through local copies of the repository called sandboxes. Sandboxes allow modelers to work on local copies of their model without being affected by concurrent changes to the multi-model from other modelers. To interact with a project, modelers checkout local copies of the repository and make modifications. The modifications are communicated to other models by synchronization with the global version of the repository.

3.2 Functionality

Data Exchange. Data exchange between models is accomplished via attributes. The owner of an attribute writes the attribute and commits it to the multi-model repository. Readers synchronize their local sandbox with the repository and access the attribute to obtain the data output by the owner model.

Model Execution. In the context of NAOMI, model execution is the process whereby model inputs are read, the model is run, and the outputs are written. It is the combined action of reading the attributes from the repository, running the model (using the DSML tool), and writing the output attributes to the repository. For example, execution of an Excel spreadsheet model involves loading the spreadsheet, reading the input attributes into the spreadsheet, allowing the calculated cell values to be updated, and writing the output attributes. An underlying assumption is that the models are parametrized, and we distinguish between model *edits*, which are actual changes to the model, and attribute changes, which are changes to the values of attributes.

Change Propagation. When a model is modified, changes are propagated by controlled exchange of attributes between models. A multi-model may be thought of as a pseudo graph, $G(V, E)$ where each model is represented by a vertex such that $\forall M_i \in M | \exists v_i \in V$. Each edge, (u, v), represents data propagated from an output attribute of one model to the input attribute of another, that is

$$\forall (u, v) \in E | \exists M_i(L, I_i, O_i), M_j(L', I_j, O_j) : a_i \in O_i \wedge a_i \in I_j.$$

This definition admits parallel edges so E must be a multi-set. To propagate changes, we can perform a depth first search (DFS) on G and select the leaves to be executed first. We propagate changes from the leaves upward with respect to dependencies such that no model M_i is executed before another model M_j if $\exists a_i : a_i \in I_i \wedge a_i \in O_j$.

There are cases for which we can not know the order of execution based on our dependency graph. Some models, particularly nondeterministic models, may need to be executed for a specified number of iterations, or until their attributes meet some predetermined constraints. Also, when the dependency graph has strongly connected components composed of two or more models, we require some way to deal with cycles.

In both of these cases, we require expert input to successfully perform execution. We have provided a feature whereby modelers may enter rules that specify the systems behavior under these circumstances. Rules are of the form $R(M) \rightarrow S$, where R is a rule, mapping a match expression (a sequence of models participating in the execution), M, to a set of actions, S, which will be performed when M is encountered in the execution plan.

Consistency Management. In the context of a multi-model, we are interested in whether the composition of the constituent models is consistent. We assume that the individual models are consistent and define the following types of multi-model consistency.

Attribute Consistency: All attributes have been updated by their owner model since the last time their owner model's input attributes changed.
Constraint Consistency: The current values of all attributes do not violate any model specific constraints.
Attribute-Model Consistency: The modification times (time of last model edit) of all models are earlier than the modification times of their output attributes.
NAOMI Consistency: Multi-models that have the properties of attribute consistency, constraint consistency, and attribute-model consistency are defined as NAOMI consistent.

Attribute consistency indicates that all models have been executed with up-to-date input data. Therefore, there are no unpropagated attribute changes. For a specific version of the multi-model, this consistency property can be ensured by executing the multi-model. However, this assumes that the models are *deterministic*. We define a deterministic model as a model that always produces the same output for a given set of inputs, in absence of a model edit.

Constraint consistency indicates that the outputs of all of the models in the multi-model fall within acceptable ranges, and by attribute-model consistency we mean that models have not been edited since their outputs have been written. These consistency types are independent of each other, and achieving one does *not* imply achieving any other.

We say that multi-models that are attribute consistent, constraint consistent, and attribute-model consistent are *NAOMI consistent*. We believe that a NAOMI-consistent multi-model is coherent and represents a steady state of the multi-model. By steady state, we mean that executing *any* subset of the models in the multi-model in *any* order will not result in any changed attributes. Thus modelers can be confident that outputs from any model in the multi-model are reliable provided the models themselves are consistent and have been integrated such that the correct attributes are exchanged. Our notion of consistency in the context of a single model is purposely vague and intended only to convey the meaning that the individual model is correct according to its own definition.

Limitations are inherent in this approach. As previously indicated, the models must be deterministic, and there can be no cyclic dependencies in the multi-models. Also model execution must halt at some point, output must be written, and attributes must have a format that can be understood by models that read it.

Version Management. The version control system keeps all the artifacts under version control and is used to synchronize local sandboxes with the global repository. It provides modelers the capability to modify any version of an individual model or the multi-model as a whole. Changes can be rolled back, and the entire multi-model can be reconstructed at any time.

3.3 NAOMI Components

Connectors. Connectors are software components that connect individual models to the multi-model. They allow controlled information exchange between models. A primary design goal of NAOMI is to have simple, lightweight connectors, enabling the easy integration of new modeling languages and tools. Connectors are language and tool specific, but may be used for multiple tools if the tools provide equivalent interfaces to the connector. Connectors *must* provide the following:

1. A mechanism for reading attributes from local copies of attribute files and inserting the attribute contents in the appropriate model files.
2. A mechanism for extracting attributes from models and writing those attributes to local files.
3. A mechanism for writing an interface description. This capability ensures that the interface always reflects the actual inputs and outputs of the model.
4. A connector-independent, common mechanism to ensure that NAOMI can always determine when the value of an attribute has changed.

Multi-Model Manager. The Multi-Model Manager is the component responsible for assisting modelers in integrating and invoking connectors, performing consistency checks, constraining attributes, and generating automatic notifications. Additionally, the Multi-Model Manager allows modelers to visualize the repository and obtain an overall view of the multi-model.

Execution Automation Engine. The Execution Automation Engine is the component responsible for determining an order for the execution of individual models in the multi-model. This is obtained through analysis of the artifacts in the repository. The Execution Automation Engine uses this ordering to orchestrate the execution process which propagates dependencies.

3.4 Requirements Fulfillment

Table 1 maps the the NAOMI concepts to the design requirements from Section 2.4. Requirement 11 can easily be met by recasting a data transformation as a model in the normal NAOMI sense.

Table 1. NAOMI components and the requirements they fulfill

NAOMI Component	Requirement Implemented
Connectors	1, 2, 3, 7, 8, 11
Multi-Model Manager	1, 2, 3 , 5, 6, 8, 9, 11
Execution Automation Engine	3, 4, 5
Version Control System	6, 8, 9, 10

3.5 Scalability of Approach

We next address the issue of the scalability of our approach. The connectors parse the model files and insert/extract attributes. Parsing the model files can done in one pass for most types of files, so the runtime is $O(n)$ where n is the size of the model input files. We can assume that an attribute can be written in constant time, so the entire operation can be done in $O(n+m)$ where m is the number of attributes. Since $m \leq n$ the runtime is still $O(n)$.

The repository is managed by a version control system such as Subversion [10] which is designed to scale well and has been shown to work well with tens of thousands of files, i.e. Linux kernel source code trees. So the number of artifacts in the repository should not be a problem. Multiple repositories could be used to alleviate this, if it became a problem.

The Multi-Model Manager needs to keep certain artifacts in memory in order to manage the multi-model. While we have not tested it with large numbers of models, we believe it should scale to at least hundreds of models. If this became a problem, the models could be partitioned into groups, those groups could be represented by new models and a hierarchical system could be developed, mitigating this problem.

In order to propagate the model attributes throughout the multi-model, the models need to be executed (where applicable). While the execution can be distributed, the communication protocol used and the time to propagate changes could present issues if the number of models and their complexity increases. The runtime of this operation is difficult to predict given that the topology of the multi-model is a determining factor. More experiments need to be done in this area, and the existence of a platform for experimentation will make these measurements possible.

4 NAOMI Implementation

This section describes our implementation of NAOMI. An overview of the workflow is shown in Figure 2. Modelers use their modeling tool to edit DSML models. The Multi-Model Manager is used to (i) invoke the DSML connectors, (ii) browse the local sandbox, and (iii) commit model and artifact changes to the central repository. The Automated Multi-Model Execution Engine (AMMEE) is then used to analyze the central repository and prepare an execution plan, which is used to maintain multi-model consistency.

4.1 NAOMI Connectors

Connectors for NAOMI may be implemented in almost any language, as long as the connector requirements described in Section 3.3 are fulfilled. For example, an Excel connector can be written in Perl, and a UML connector can be written in Java. The integration mechanism is specific to the connector, and connector writers may implement the mechanism any way they choose. For example, an Excel connector might expect that Excel models are integrated by naming cells in a special way (e.g., a cell named `naomi_get.system.reliability` can instruct the Excel connector to retrieve the `system.reliability` attribute and place it in the cell).

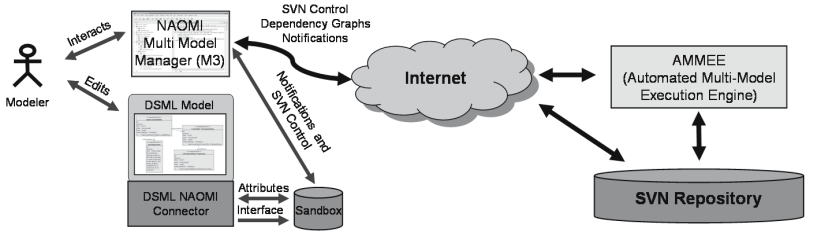

Fig. 2. NAOMI work-flow: modeler uses model tool to edit DSML model. Multi-Model Manager and connector are used to maintain sandbox files. M3 commits changes to NAOMI repository, which are analyzed by the Automated Multi-Model Execution Engine.

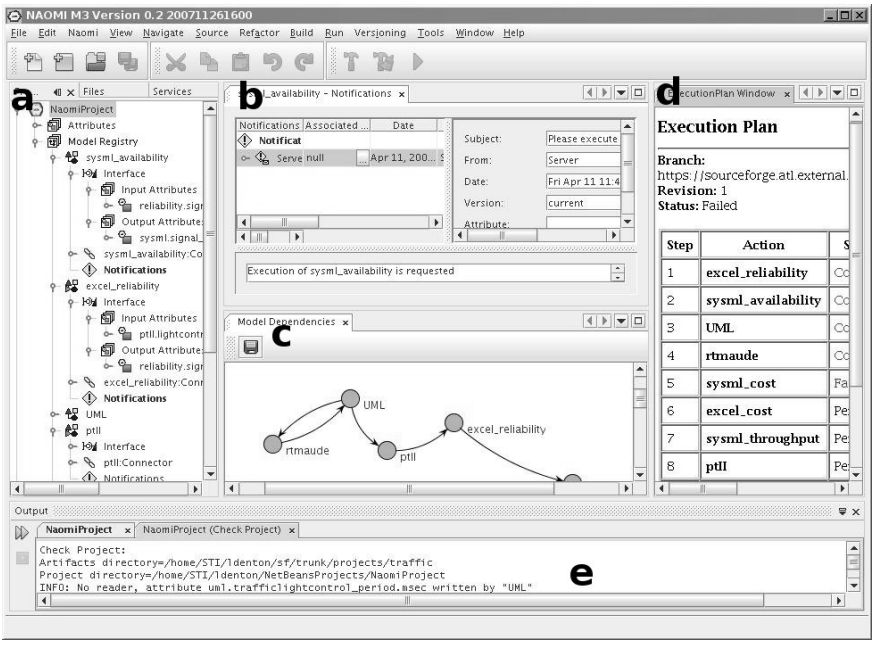

Fig. 3. M3 application showing a NAOMI project: (a) project node with DSML models, their interfaces, and connectors, (b) notification pane showing notification for the currently selected model, (c) model-dependency graph, (d) execution plan and status from AMMEE, (e) message area showing output from operation

4.2 Repository

The models and their corresponding editing tools reside inside the repository in their native forms. The repository is managed by the Subversion [10] version control system. Model interfaces, attributes, notifications, and the model registry are all implemented as XML [11] files, with defined XML schemas [12], stored inside the repository. Attribute constraints are defined as XML schemas that impose a restriction on an attribute file.

Every registered model is allowed one constraint schema per attribute, allowing each modeler to use a customized set of constraints. An XML schema validation engine is used to validate each attribute against the associated constraints.

4.3 NAOMI Multi-model Manager

The NAOMI Multi-Model Manager (M3) is implemented as a Netbeans [13] rich client to be used locally on modelers' machines to assist them in managing their sandboxes.

M3's user interface is shown in Figure 3. Section (a) of the figure shows the first level of repository visualization. Each registered model is shown as a node in an explorer-type view on M3. Expanding the model node reveals the interface, connector and notifications nodes. M3 also provides graph visualizations of the relationships between each model in the multi-model (shown in (c) of Figure 3). These visualizations allow modelers to see the full extent of the relationships between models and expose owner-less and reader-less attributes.

M3 automatically handles the sending and receiving of notifications and provides facilities for control of the SVN repository. Additionally, it works in conjunction with AMMEE to provide work flow management during model execution.

4.4 Automated Multi-model Execution Engine

The Automated Multi-Model Execution Engine (AMMEE) facilitates the generation of a workflow for integrated DSMLs in the context of a NAOMI project. When execution of a project is requested to resolve dependencies, AMMEE branches the project, queries the model registry for all registered models, and builds a dependency graph based on each model's inputs and outputs. A plan of execution, based on these dependencies, is stored in the repository. This plan is updated as execution progresses, and it allows modelers to examine the status of the multi-model's execution. AMMEE interacts with M3 to request the execution of each model until a consistent state is reached.

5 Experiments

We study model interactions in the context of a system for controlling traffic lights within a city. Our initial multi-model simply consists of requirements, partitioning, and interacting state machine algorithms for a single pedestrian light and single car light controller at the intersection of two streets. Our experimental multi-model consists of four models in four different DSMLs, which were chosen for simplicity of explanation:

- A structural model, developed in UML, that writes two attributes, "duration of red light" and "duration of green light" that specify how long the lights are on.
- An algorithmic model [14], developed in Ptolemy II, that reads the light durations and outputs a "light control failure time percent" attribute, which is the expected amount of time the two lights will be in a failure mode.
- A reliability model, developed in Excel, that reads the "light control failure time percent" attribute and writes a "signal availability percent" attribute, which is the percent of time the signal lights will be available.
- A requirements model, developed in SysML, that reads the "signal availability percent" attribute.

5.1 Experiment 1: Identifying Unanticipated Impacts

We demonstrate how NAOMI propagates changes in one model to dependent models and flags any unpropagated changes. When the "duration of red light" attribute is changed in the UML model, an attribute consistency check run in M3 fails on all models except the UML model. The reason for this can be seen by examining the model dependency graph in Figure 4; all of the other models are dependent either directly or indirectly on the output of the UML model. To resolve this problem, AMMEE defines the order of execution as visualized by the graph shown in the Figure 4. The Ptolemy II model is executed, propagating the aforementioned change in an attribute owned by the Ptolemy II model. The model execution continues in the prescribed order until the SysML model is executed, propagating the change to the SysML owned attribute, "signal availability percent." Upon completion, all attribute consistency checks pass, indicating successful attribute change propagation.

The change propagation we demonstrated shows how a seemingly simple change in one domain can cascade into others and how NAOMI can detect unpropagated changes.

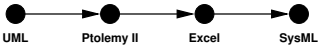

Fig. 4. Model dependency graph for Experiment 1

5.2 Experiment 2: Tracking Violations of Design Intent

Next, we tested the capability to automatically alert modelers to constraint violations. We added a constraint to the input of the SysML model to ensure that "signal availability percent" is at least 90%. We then repeated Experiment 1: the change propagated to "signal availability percent," violated the constraint, and NAOMI automatically generated a notification to the SysML modeler that the constraint was violated. This shows that NAOMI is capable of attaching constraints to attributes which can be used to ensure models exchange valid data.

5.3 Experiment 3: Managing Model Coevolution

Lastly, we demonstrate managing model coevolution in a NAOMI project. The UML model is modified so that it no longer writes values to the "duration of green light" attribute. In Figure 5, a project consistency check shows an error because the Ptolemy model is reading an attribute that is no longer written by any model. This generates an alert to the modelers of the integration problem resulting from the evolution of the UML model. This type of error would be difficult to detect without the automated consistency checking that NAOMI provides.

6 Related Work

Previous approaches to multi-modeling often used a common representation that the various models are mapped to. For example, in the area of formal specification, Zave

```
Check Project:
Artifacts directory=/projects/traffic
Project directory=/NetBeansProjects/NaomiProject
ERROR: No writer for attribute duration_of_green_light read by Ptolemy II
```

Fig. 5. Example output from a project check that shows the Ptolemy II model is reading an attribute that is not written by any model

and Jackson [15] provided a compelling method for multi-paradigm formal specification. Using their method, partial specifications are produced with languages suited to the aspects under consideration. These partial specifications are converted to assertions in first order logic, and the multi paradigm specification is equivalent to the conjunction of these first order assertions.

Brooks, Feng, and Lee [14] described the construction of a hierarchical multi-model in Ptolemy II and showed how model based specification can be enriched by the incorporation of multiple distinct modeling formalisms. They also showed how actor-oriented classes enforce consistency across multi view models. Girault, B. Lee, and E. Lee [16] combined finite state machines with various concurrency models, each having different strengths and weaknesses, inside the Ptolemy framework.

Other approaches to multi-modeling have used meta-model-based integration techniques. Hardebolle, Boulanger, Marcadet, and Vidal-Naquet [7] describe ModHel'X, which was inspired by Ptolemy and implements a multi-formalism approach to computational modeling. The Open Tool Integration Framework (OTIF), described by Karsai, Lang,and Neema [8], addresses the data integration problem through meta-model-based transformations, and they present two architectures for the framework.

The Electronic Tool Integration (ETI) [17,18] framework is an example of using light-weight data exchange as a mechanism for tool integration using web-services. However, ETI does not address issues such as managed repositories, with built-in versioning and consistency management, which are critical for scalable application to projects.

Wimmer, Schauerhuber, Strommer, Schwinger, and Kappel [19] demonstrated a semi-automatic approach to bridging DSMLs based on the manual mapping of domain-specific meta-models and UML combining a dedicated bridging language, automatic generation of UML profiles and model transformations. In contrast, we follow a light-weight data exchange mechanism, without requiring an intermediate language.

Kappel, Kramler, Kapsammer, Reiter, Retschitzegger, and Schwinger [20] describe ModelCVS, which focuses on ontology-based meta-model integration via meta-model lifting. ModelCVS is built on the concurrent versions system, CVS [21] and recognizes the key role that versioning plays in multi-view modeling.

Our approach is different; we do not seek a common representation for the multi-model's constituent models, nor do we assume the meta-models of DSMLs will have sufficient semantic overlap to allow for a mapping. We remain tool and language agnostic by decoupling the implementation of a model from the interface it exposes to the multi-model and focus on the interactions between different DSMLs.

7 Conclusion

We have outlined key multi-modeling challenges and derived a set of requirements that a generalized multi-modeling system should fulfill. By adopting a language-neutral stance and representing models through interface abstractions, we showed how multi-models can be composed from diverse DSML models and kept consistent, avoiding limitations of previous methods that seek to transform models to a common representation. We presented NAOMI, an experimental platform that implements these concepts, and showed how this platform can assist modelers by propagating changes and data across domains, alerting them to violations of constraints and notifying them of failures in consistency. This work enables the construction of robust multi-models not possible without NAOMI.

For future work we wish to address the issue of ensuring the preservation of semantic meaning during the exchange of data between models as described by Challenge 3 in Section 2.3. Additionally we want to address the limitations in our input/output based view of models, particularly specification models as discussed in section 3.1. Our research in this area is ongoing, and we are looking at using approaches like that of Bräuer and Lochmann [22] to allow relevant specification information to be shared between different modeling languages. We are continuing to investigate ways of addressing the scalability issues associated with multi-model execution raised in section 3.5.

Acknowledgments

The authors would like to thank Edward A. Lee, Christopher Brooks, and Thomas Huining Feng of the University of California at Berkeley; Jose Meseguer, Peter Csaba Ölveczky, and Kyungmin Bae of the University of Illinois at Urbana-Champaign; Artur Boronat of the University of Leicester, Douglas Schmidt, Jules White, James H. Hill, and Sean Eade of Vanderbilt University for their helpful suggestions and their contributions to the NAOMI project.

References

1. Hessellund, A., Czarnecki, K., Wasowski, A.: Guided development with multiple domain-specific languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 46–60. Springer, Heidelberg (2007)
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories. Wiley, Chichester (2004)
3. OMG: Unified modeling language specification version 2.0 (2005), `http://www.uml.org/`
4. OMG: Modeling and analysis of real-time and embedded systems (MARTE) (2006), `http://www.omgmarte.org/`
5. OMG: Omg systems modeling language (sysml) specification version 1.0 (2007)
6. Nuseibeh, B., Kramer, J., Finkelstein, A.: Expressing the relationships between multiple views in requirements specification. In: ICSE 1993, pp. 187–196. IEEE Computer Society Press, Los Alamitos (1993)
7. Hardebolle, C., Boulanger, F., Marcadet, D., Vidal-Naquet, G.: A generic execution framework for models of computation. In: Fernandes, J.M., Machado, R.J., Khedri, R., Clarke, S. (eds.) MOMPES 2007, pp. 45–54. IEEE Computer Society, Los Alamitos (2007)

8. Karsai, G., Lang, A., Neema, S.: Design patterns for open tool integration. Software and Systems Modeling (SoSym) 4(2), 157–170 (2005)
9. Ölveczky, P., Meseguer, J.: Semantics and pragmatics of real-time maude. Higher-Order and Symbolic Computation 20(1), 161–196 (2007)
10. CollabNet: Subversion version control system (2006), http://subversion.tigris.org/
11. World Wide Web Consortium: W3C Extensible Markup Language (XML) (2006), http://www.w3.org/XML
12. World Wide Web Consortium: W3C XML schema (2006), http://www.w3.org/XML/Schema
13. Sun Microsystems: Netbeans (2008), http://netbeans.org/
14. Brooks, C., Feng, T.H., Lee, E.A.: Multimodeling: A preliminary case study. Technical Report UCB/EECS-2008-7, EECS Department, University of California, Berkeley (2008)
15. Zave, P., Jackson, M.: Where Do Operations Come From?: A Multiparadigm Specification Technique. IEEE Transactions on Software Engineering 22(7), 508–528 (1996)
16. Girault, A., Lee, B., Lee, E.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(6), 742–760 (1999)
17. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: concepts and design. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 9–30 (1997)
18. Margaria, T., Nagel, R., Steffen, B.: jETI: A Tool for Remote Tool Integration. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 557–562. Springer, Heidelberg (2005)
19. Wimmer, M., Schauerhuber, A., Strommer, M., Schwinger, W., Kappel, G.: A Semi-automatic Approach for Bridging DSLs with UML? In: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (2007)
20. Kappel, G., Kramler, G., Kapsammer, E., Reiter, T., Retschitzegger, W., Schwinger, W.: Modelcvs - a semantic infrastructure for model-based tool integration. Technical report, Johannes Kepler University of Linz and Vienna University of Technology (2005)
21. Grune, D., Berliner, B., Polk, J., Jones, L., Price, D.R., Baushke, M.: Concurrent versions system (2005), http://ftp.gnu.org/non-gnu/cvs/
22. Bräeuer, M., Lochmann, H.: Towards semantic integration of multiple domain-specific languages using ontological foundations. In: Proceedings of the 4th International Workshop on (Software) Language Engineering (ATEM) (2007)