View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DigitalCommons@Florida International University

Florida International University
FIU Digital Commons

School of Computing and Information Sciences | College of Engineering and Computing

5-2011

Challenges and Directions in Formalizing the Semantics of Modeling Languages

Barrett R. Bryant
University of Alabama - Birmingham

Jeff Gray
University of Alabama - Tuscaloosa

Marjan Mernik
University of Maribor

Peter J. Clarke
School of Computing and Information Sciences, Florida International University, clarkep@fiu.edu

Robert B. France
Colorado State University

See next page for additional authors

Follow this and additional works at: http://digitalcommons.fiu.edu/cs_fac

Part of the Computer Sciences Commons

Recommended Citation

Bryant, B. R., Gray, J., Mernik, M., Clarke, P. J., France, R. B., Karsai, G.: Challenges and Directions in Formalizing the Semantics of Modeling Languages. Computer Science and Information Systems, Vol. 8, No. 2, 225-253. (2011)

This work is brought to you for free and open access by the College of Engineering and Computing at FIU Digital Commons. It has been accepted for inclusion in School of Computing and Information Sciences by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

Authors
Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, and Gabor Karsai

This article is available at FIU Digital Commons: http://digitalcommons.fiu.edu/cs_fac/10

DOI:10.2298/CSIS110114012B

Challenges and Directions in Formalizing the Semantics of Modeling Languages

Barrett R. Bryant[1], Jeff Gray[2], Marjan Mernik[3], Peter J. Clarke[4], Robert B. France[5], and Gabor Karsai[6]

[1] Department of Computer and Information Sciences,
University of Alabama at Birmingham, Birmingham, Alabama 35294-1170, USA
bryant@cis.uab.edu
[2] Department of Computer Science, University of Alabama
Tuscaloosa, Alabama 35487-0290, USA
gray@cs.ua.edu
[3] Faculty of Electrical Engineering and Computer Science, University of Maribor
2000 Maribor, Slovenia
marjan.mernik@uni-mb.si
[4] School of Computing and Information Sciences, Florida International University
Miami, Florida 33199, USA
clarkep@cis.fiu.edu
[5] Computer Science Department, Colorado State University
Fort Collins, Colorado 80523-1873, USA
france@cs.colostate.edu
[6] Institute for Software-Integrated Systems, Vanderbilt University
Nashville, Tennessee 37235, USA
gabor.karsai@vanderbilt.edu

Abstract. Developing software from models is a growing practice and there exist many model-based tools (e.g., editors, interpreters, debuggers, and simulators) for supporting model-driven engineering. Even though these tools facilitate the automation of software engineering tasks and activities, such tools are typically engineered manually. However, many of these tools have a common semantic foundation centered around an underlying modeling language, which would make it possible to automate their development if the modeling language specification were formalized. Even though there has been much work in formalizing programming languages, with many successful tools constructed using such formalisms, there has been little work in formalizing modeling languages for the purpose of automation. This paper discusses possible semantics-based approaches for the formalization of modeling languages and describes how this formalism may be used to automate the construction of modeling tools.

Keywords: model-based tools, modeling languages, semantics.

1. Introduction

With increasing frequency, scientists and engineers in diverse areas of focus, as well as end-users with specific domain expertise, are requiring computational processes to allow them to complete some task (e.g., avionics engineers who seek input on a modeled design from verification tools, or geneticists who need to describe computational queries to process a gene expression). A challenge emerges from the lack of knowledge of such users in terms of expressing their computational desire (i.e., such users typically are not familiar with programming languages). Model-driven engineering (MDE) is an approach that provides higher levels of abstraction to allow such users to focus on the problem, rather than the specific solution or manner of realizing that solution through lower level technology platforms [46][52]. However, the potential impact of modeling is reduced due to the imprecise nature in which modeling languages are defined [26]. The large majority of modeling languages are defined in an ad hoc manner that lacks precision and a common reference definition for understanding the meaning of language concepts. In current practice, the meaning of a modeling language is often contained only in a model translator (we will use the term *model interpreter* in this paper to refer to such translators) that converts a model representation into some other form (e.g., source code). The current situation in MDE is not unlike the early period of computing when the definition of a programming language was delegated to "what the compiler says it means." Such an approach not only promotes misunderstanding of the meaning of a modeling language, but also limits opportunities for automating the generation of various language tools (much like the adoption of grammars provided a reference point for compiler and other tool generation for a programming language).

The advantages of formal specification of programming language semantics are well-known. First, the meaning of a program is precisely and unambiguously defined; second, it offers a unique possibility for automatic generation of language-based tools (e.g., [27]). Unfortunately, formal specifications, syntax and semantics, of modeling languages have not been developed to this level yet. Although the syntax of modeling languages is commonly specified by metamodels, an appropriate and standard formalism for specifying the (behavioral) semantics of modeling languages does not yet exist. Hence, there is no automatic generation of model interpreters, debuggers, simulators and verification tools.

In this paper, we describe challenges and directions in formalizing the semantics of modeling languages. The ideas developed in this paper were derived from the *Workshop on Formalization of Modeling Languages* held in conjunction with the *European Conference on Object-Oriented Programming (ECOOP)* in Maribor, Slovenia, on June 21, 2010. The paper is organized as follows. Section 2 motivates the need for semantics in modeling languages and reviews existing work in this area. In Section 3, we describe an approach based on state machine models. Section 4 describes a metamodel-based approach to semantics. In Sections 5 and 6, we discuss our experiences with

semantics-based modeling tools for verification. Finally, we conclude in Section 7.

2. The Need for Semantics in Modeling Languages

Much of the success of MDE is dependent on the descriptive power of domain-specific modeling languages (DSMLs) [24][29][50]. One of the current challenges of adopting a DSML is the lack of a precise description of the semantics of the DSML. Initial attempts are described in [9], [10] and [16]. The typical technique for specifying the syntax and static semantics of a DSML is to use a metamodel, which describes concepts in a problem domain and their relationships. A standard known as MOF (Meta-Object Facility) has been proposed for defining the syntax of modeling languages by following a similar role as BNF and its variants (e.g., EBNF) for programming languages. Metamodels are currently even used for specifying the syntax of domain-specific programming languages [42]. However, the situation concerning syntactical description of languages is completely different from semantics. It is often easier to describe the structure of a DSML using a metamodel than it is to specify the syntax of a programming language using BNF. However, specifying detailed behavior (semantics) is much harder with DSMLs. In our opinion, this is why only the syntax of current DSMLs are formally described, but the semantics are left toward other less than desirable means. For example, as will be discussed further in Section 5, the semantics of the UML (Unified Modeling Language) metamodel is defined using a mixture of OCL (Object Constraint Language) and informal text, which is clearly unacceptable for formal analysis. Hence, the meaning (semantics) of models are often not formally described. For this purpose, general-purpose programming languages (e.g., C++) are often used to define model interpreters that have an internal representation of the semantics of a DSML. The lack of a formal definition of DSML semantics contributes to several problems, as highlighted in the following paragraphs.

Tool Generation Challenges: The semantics of DSMLs are not defined formally. Hence, proving properties about concepts and relationships in the domain is not possible. Moreover, a model interpreter cannot be automatically generated in most cases. A further consequence is that various other model-based tools (e.g., debuggers, test engines, simulators, verifiers) also cannot be generated automatically.

Tool Analysis Challenges: Model interpreters are often implemented with general-purpose programming languages (GPLs). This has several consequences. Verifying a model interpreter is a very difficult, if not impossible task. As such, verification, optimization, and parallelization of models can be expressed only through GPLs.

Formal Language Design: DSMLs are also languages that need to be designed properly. This leads to several key questions: What are the design

principles for modeling languages? How are the results of domain analysis used in modeling language design?

Modeling Language Composition: In practice, multiple domains might be involved to describe different perspectives of a modeled system. In such a case, there is a need for composing DSMLs together. Presently, there is little support for formal composition and evolution of DSMLs.

2.1. Related Work in Modeling Language Definition

Some work on the generation of various modeling tools has already been investigated. Different approaches to the issue of defining the semantics of DSMLs have been proposed; these differ in their applicability and potential of leveraging automatic or at least semi-automatic language tool generation.

2.2. Mapping the DSML into Existing Formal Languages

A common way of defining the semantics of a modeling language is through *translation semantics*, where the abstract syntax of the main DSML is mapped into the abstract syntax of an existing formal language, with well-defined and understood semantics. The mapping is achieved through model transformations. An advantage of this approach is that the DSML can convey existing tools of the language into which it is translated. A common critique of this approach is that since the semantics definition is not defined in the metamodel of the DSML, it is very challenging to correctly map the constructs of the DSML into the constructs of the target language. The underlying cause for this is that the mappings are not at the same level of abstraction and the target language may not have a simple mapping from the constructs in the source language. Another issue of the translation semantics approach is the mapping of execution results (e.g., error messages, debugging traces) back into the DSML in a meaningful manner, such that the domain expert using the modeling language understands the result.

One concrete approach that uses translation semantics is called *semantic anchoring* [9], which uses the well-known Abstract State Machines (ASM) formalism [7] to define the semantics. We will discuss the technique in detail below. This solution maps the abstract syntax of the DSML, which was defined in the GME (Generic Modeling Environment) metamodeling tool [33], into well-established semantic domains, called semantic units (e.g., timed automata, and discrete event systems) that have been defined in the ASML (Abstract State Machine Language) tool. The initial work on semantic anchoring did not show any application of tool generation from the semantics specification, although the usage of ASML enables compilation, simulation, test case generation and verification of ASML specifications, as will be discussed further in Section 3. A similar concrete approach was proposed by Di Ruscio et al. [16], which also did not demonstrate any tool generation

based on the semantics definition. Gargantini, Riccobene and Scandurra [22] introduce a semantic framework based on ASM, which also includes three translational semantics techniques: semantic mapping, semantic hooking and semantic meta-hooking. The authors do not demonstrate any tool generation from their semantics specifications. The Moses tool suite [21], which defines the syntactical aspects (e.g., vertex edge/types, syntactical predicates) of the language with a Graph Type Definition Language (GTDL), uses ASM for prototyping model interpreters to achieve the definition of semantics. Based on this kind of formal specification, the Moses tool suite generates animation and debugging tools for visual models. The work presented in [43] describes a translation semantics definition with Maude, which is a rewriting logic-based language. Based on such a semantics definition simulation, reachability and model-checking analysis tools can be generated. Sadilek and Wachsmuth [44] present a semantics definition based on a transition system, where the states are defined by metamodel instances and the transitions are defined by model transformations. The work of Hahn [25] uses the Object-Z language [48] as the means of defining the translation semantics.

2.3. Weaving the Semantics into the Metamodel

Another approach is to *weave behavior* into the abstract syntax (i.e., the metamodel) by a meta-language (also called action language), which can be used to specify the bodies of operations that occur in the metamodel. This permits the model to be executable, because the semantics are defined inside the operation bodies. The significant drawback of this approach is the fact that some meta-languages are very similar to 3rd generation programming languages; therefore, they have to be used in an operative way. The advantage of this approach is the fact that this kind of semantics specification can be mastered by most domain experts.

A well-known representative of this approach is the Kermeta tool [40], which extends an abstract meta-layer with an imperative action language to weave a semantic definition within the metamodel. Kermeta constructs contain specification of operations over metamodel elements. The built-in support for specification of operational semantics enables the automatic generation of simulation and testing tools. Another example is the approach proposed by Scheidgen and Fischer [45], where an operation is specified through the use of OCL statements and an activity diagram. The graphical format of this meta-language is particularly familiar to users with a strong modeling background. The authors mentioned that in the future they will work on automatic debugger generation. Soden and Eichler [49] propose a similar approach based on the usage of activity diagrams as the meta-language. Their future work will be implemented in a framework known as the Model Execution Framework (MXF) and should take an important place in the Eclipse environment. Based on the semantics definition, various tools like trace analysis and runtime verification will be automatically generated. The Mosaic XMF framework [3], which uses an extended OCL language to provide

semantics, is another representative of the semantics definition approach. Initial work that corresponds to the behavior weaving approach was also undertaken in UML [51], where action semantics were proposed to achieve the goal of executable UML models. To define the semantics of a new language, no notation was enforced, but the authors "suggest activities with action semantics for language modelling." Ducasse et al. [17] use Smalltalk as a meta-language in their DSML semantics definition.

2.4. Defining the Semantics with Rewrite Rules

Semantics also can be specified through *rewriting systems,* where the system typically consists of rewrite rules. Each rewrite rule consists of a left- and a right-hand side. The execution of a rewrite system is based on the repeated application of the rewrite rules to an existing configuration (e.g., model). A rule is applied when the left-hand side of the rule is found in the configuration, in such a way that this occurrence will be replaced by the right-hand side of the rule. The execution is complete when there is no rule that can be applied to the configuration. Typically, the existing approaches employ graph rewriting where the semantics can be specified in an operational fashion through the graphical definition given by graph grammars. Graph rewriting provides a mathematically precise and visual specification technique by combining the advantages of graphs and rules into a single computational paradigm [53].

Graph rewriting specification was employed in the AToM3 tool [32], which uses triple graph grammars as rewriting rules. One of the interesting features of AToM3 is that the definition of rewriting rules is given through concrete syntax, which makes semantic specification especially amenable for domain-experts. AToM3 can use graph grammar definitions to generate visual model simulators and implement model optimizations and code generation. The dynamic metamodeling [19] approach describes the semantics of UML behavior diagrams with collaboration diagrams, which are used in graph transformations. The authors mention future work on the generation of model simulators. Ermel et al. [20] enable translation of UML behavior diagrams into graph transformations, which are the basis for semantics that are used to generate a visual simulator of UML models.

2.5. Other Approaches to the Definition of Semantics

There also exist other examples of generating tools from semantic definitions that are described in GPLs. Perhaps a valuable lesson can be learned even from these examples. One of the most well-known approaches is Ptolemy [18], which is a tool that enables animated interpretation of hierarchically composed domain-specific models. Models in Ptolemy consist of heterogeneous domains (models of computation) that can have different semantics. Adding a new DSML to Ptolemy is cumbersome, because the

syntax and semantics have to be defined manually (i.e., hand-coded) in Java by implementing a "director" that assigns executable semantics to the DSML constructs.

3. Defining the Semantics of Modeling Languages

We view semantics as a mapping from the abstract syntax (A) of the DSML to some semantic domain (D). The abstract syntax defines the fundamental modeling concepts, their relationships, and attributes used in the DSML, and the semantic domain is some mathematical framework whose meaning is well-defined. The abstract syntax defines the data structures that represent the modeling constructs, and, as such, it can be considered as a schema for the models. For instance, in a modeling language representing Finite State Machines (FSMs), we will need data structures for states and transitions, which need to be related to each other such that one can find the source and target states of transitions. Instances of such data structures do represent FSMs, and algorithms are available to analyze them. The concrete syntax (S) is the human-readable manifestation of the abstract syntax. In our FSM example, the concrete syntax can be textual (e.g., a simple language where an FSM is represented as a set of names for states, and a set of transitions represented in the form 'state1 → state2', where state1 and state2 are names of states), or it can be graphical (e.g., a graphical notation with bubbles representing states and arrows connecting bubbles representing the transitions). There is always a well-defined mapping between A and S. We use the concrete syntax to create and modify the models, with the assistance of a customized metamodeling tool, such as the GME. Note that changes on the models performed using the concrete syntax must eventually be reflected as changes in the abstract syntax form of the models.

An example for the visual depiction of abstract syntax is shown in Figure 1, which uses the UML class diagram graphical formalism. The abstract syntax is that of a Stateflow-style [36] hierarchical state machine, with `States` and `Transitions` being the main elements. The top-level model element `Stateflow` is a `Folder` that acts as a container for models. This container will contain `States` that contain other `States` and `Transitions`. The recursive containment of states within states allows the composition of hierarchical state machines. Transitions connect `TransConnectors` that are abstract (only their derived classes can be instantiated), and that could be `States`, `Junctions`, initial transitions (`TransStart`), history junctions (`History`), or references (`ConnectorRef`) that point to other `TransConnectors`. States may also contain Data or Event elements, as well as an optional reference to a data type (`TypeBaseRef`). Note that this composition expressed as abstract syntax follows the legal composition of model elements available in the Stateflow language. For example, a `Transition` cannot connect a `Data` element to a `State` – there is no legal association between them.

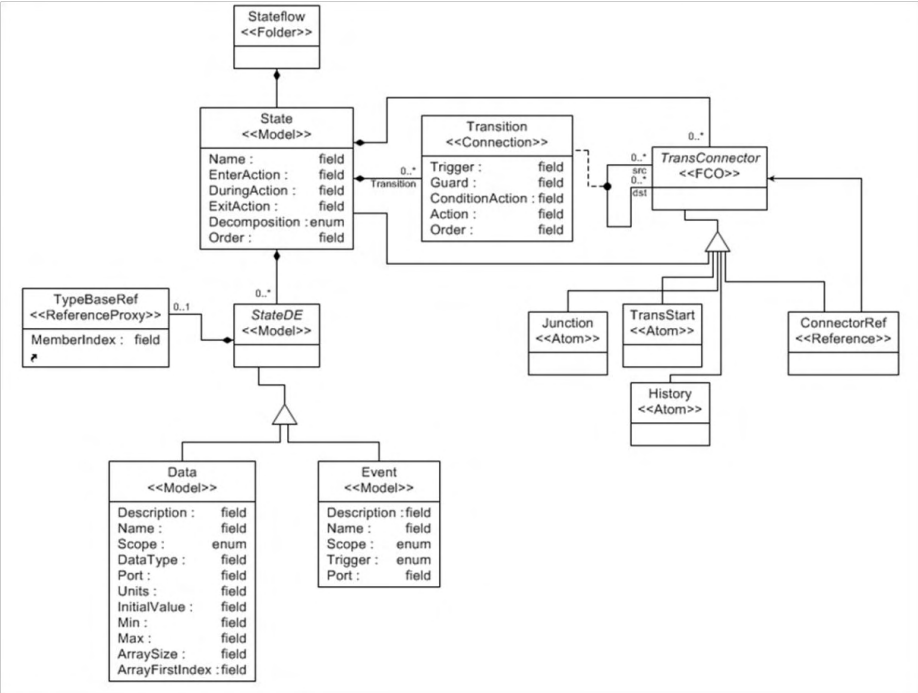


Fig. 1. Abstract syntax for a DSML representing a Hierarchical FSM

One can also define well-formedness constraints (*C*) over the abstract syntax. In our example, a well-formedness constraint could specify that there must be precisely one state marked as “initial” among the states contained in a Stateflow model, and the sub-states of a state. Such constraints delineate what models are considered ‘correct’ with respect to a static notion of semantics; the constraints can be checked on the models directly, without referring to a semantic domain.

The semantic domain for such a DSML could be a finite state *machine (M)* (implemented in hardware or software), with a finite set of *states* (with precisely one, distinguished state called the *initial state*), a finite set of triggering *events*, and state *transitions* between states. Transitions are labeled with triggering events and Boolean guard expressions over some variables of the system. A model expressed in the DSML compliant with the abstract syntax will map to a specific machine that operates as follows: The machine is always in a specific state, called the *current* state. When the execution starts, the current state is the initial state. When an event arrives, it is matched against the event labels attached to transitions emanating from the current state, and if a matching label is found the transition is selected. The guard for the selected transition is evaluated, and if it is true then the current state becomes the target state the transition points to. If the event does not match any event on an outgoing transition (or if it does match, but the guard is false), the current state does not change. It is required that if multiple transitions are

selected, at most one guard can be true, otherwise the behavior is non-deterministic and the model is incorrect. Note that this machine does not have hierarchical states.

The semantics of the models can be defined by a mapping $m : A \rightarrow D$ that instantiates a specific (non-hierarchical) finite state machine from a model. After the machine is created, it operates in some environment according to the algorithm described above. Note that the semantics is *ultimately* defined by our understanding of how the machine works: although it can be formalized, it is still dependent on our (possibly inaccurate) understanding of the operation of the machine. After this understanding is refined, we can 'build' it as a digital circuit or as a software simulator. Note that the semantic domain defines the meaning of a model with respect to a dynamic notion of semantics; one needs a "machine" to execute the computation denoted by the model.

Note that not all DSMLs have an executable (or 'operational') semantics. For instance, UML class diagrams are not 'executable,' however, they can be expressed in various forms (e.g., C++ code consisting of classes with data members and member functions). Some DSMLs have very weak opportunity for semantics definition; for instance, UML use-case diagrams can only be paraphrased in a natural language, without any formal mapping. Below, we restrict the discussion to DSMLs that do have executable semantics.

Drawing from the example, we can observe that the specification of semantics may be accomplished in two steps: (1) defining the 'semantic domain,' and (2) defining the mapping between the abstract syntax and the constructs of the semantic domain. For a pragmatic approach one can envision a translator for (2), and a simulator (or interpreter) for (1) that interprets the result of (2) with some input. Below we describe two variations on how these steps can be accomplished.

3.1. Definition via a Semantic Unit

Assume we have well-defined, accepted, and well-understood modeling languages whose semantics are simple and defined in a non-ambiguous, preferably executable way. Let's call these core modeling languages *semantic units*. An example of a semantic unit could be the domain of simple finite state machines, as described in the previous section. If a new DSML needs to be defined, one has to specify the semantics of this new language by showing how the models built in the new language could be reduced to (or transformed into) the well-defined semantic units. The principle is illustrated in Figure 2.

In this method, the semantics are mainly defined by the transformation $M_{DSMLi,SU}$ that maps the abstract syntax of the DSML (A of DSML-i) to the abstract syntax of a semantic unit (A of SU). The concrete syntax (C) of the DSML is related to the abstract syntax of the DSML (A) via a mapping (M_{Ci}). The semantic domain of the DSML is some S, and the notional semantics of the DSML is defined via the mapping M_{Si}. The key idea here is that we define the M_{Si} mapping in two steps: (1) the transformation ($M_{DSMLi,SU}$), and (2) the

semantic mapping of the semantic unit M_{SU}. Note that semantic units also have a DSML: a concrete syntax (C of SU), a semantic domain (S of SU), an abstract syntax (A of SU), and the mappings: M_{C2} for the syntax and M_{SU} for the semantics. The base semantic domain is much simpler than a higher level DSML. The transformation can be specified formally, for instance in the form of graph transformation rules [2], which represent how to rewrite a higher level DSML into the lower level DSML; hence, establishing a formal, yet executable mapping between the two languages. For the example described above, the transformation rewrites the hierarchical, Statechart-like state machine into a flat, non-hierarchical state machine.

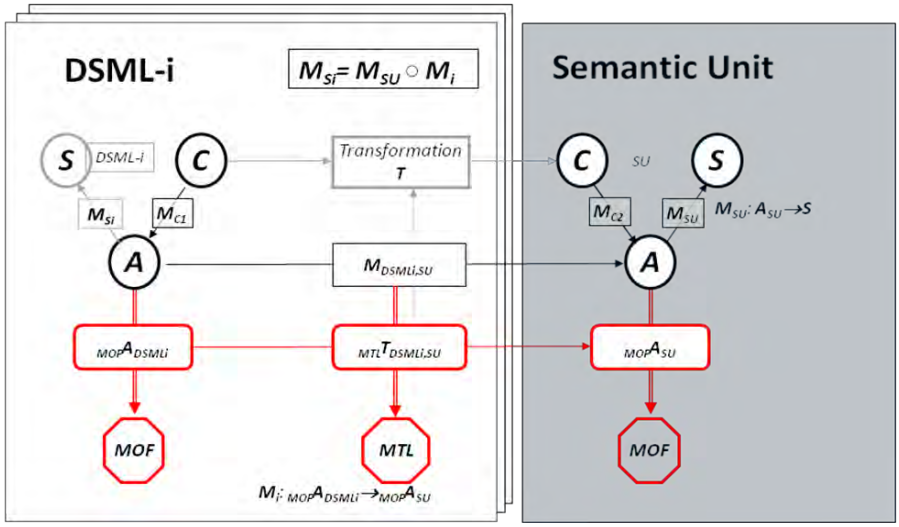


Fig. 2. Defining semantics via a transformation and a semantic unit

For specifying the semantic unit, a tool has been created that uses the Abstract State Machine Language (ASML) [38] to represent semantic units. ASML allows building these semantics units using the Abstract State Machine concepts [7] (i.e., essentially as transition systems with sophisticated data structures representing the state of the system). A number of prototype model transformations have been built that show how a non-trivial DSML (e.g., a Statechart-like language) can be formally defined via the transformation [9]. These form the initial components of a tool suite where one can define the abstract syntax of a language, together with its semantics using semantic units and transformations. An interesting property of ASML is that it is executable, thus one can rapidly prototype and experiment with DSMLs by executing their models as ASML "programs."

In this approach, the main complexity is in the model transformation process, and semantic units are typically simple. A semantic unit is a subject of reuse: it is designed to be used with different DSMLs. Because of this desired property, all of the semantic (and possibly syntactic) variations are kept in the transformation part. Note that the semantic unit can be expressed

in any formalism that does not have to be executable. ASML was used in the projects described above and is suitable for execution and test generation, but formalisms better suited for model checking (e.g., nuSMV [12]) can be used as well.

3.2. Definition via an Interpreter

The approach described in the previous section is well-suited for cases when one semantic unit can serve a number of DSMLs and all the semantic variations can be captured in the transformation. However, this is not the case for many DSMLs, most notably the 20+ variants of the Statechart notation [5]. In this case, another approach is to simplify the translation part and define the semantics using an *interpreter* that directly executes the models.

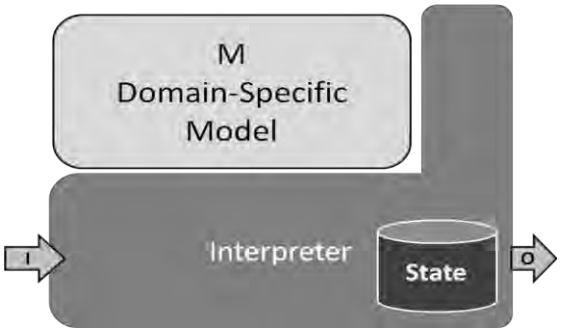


Fig. 3. Defining semantics via an interpreter

Formally, an interpreter is a mapping *i* that depends on the model M, and implements $i(M) : I \times S(M) \rightarrow O \times S(M)$, where *I* is the input event alphabet, *O* is the output event alphabet, and *S* is the set of the internal states of the interpreter, also dependent on the model. The concept is illustrated in Figure 3. The model is a read-only data structure that controls the interpreter's behavior, while the state is updated by the interpreter as it processes inputs. Of course, an interpreter is not different conceptually from a semantic unit, but typically much more complex.

Such interpreters can be defined in any executable language, including conventional languages. This has advantages: (1) any developer skilled in the implementation language can understand the specification of the semantics, (2) all the formal reasoning and analysis tools available for the implementation language can be used, (3) fast prototyping of semantics is feasible, (4) program verification, debugging, and testing tools available for the implementation language can be immediately used. The disadvantages of the approach are: (1) reasoning about programs is typically more difficult than reasoning about models, (2) verifying an interpreter + model assembly is inefficient, as the resulting system has many more states than strictly needed

by the model, and (3) treating non-deterministic behavior as complex, because a concrete interpreter is always deterministic.

We have used this interpreter-based approach to define the semantics of two Statechart variants: (1) UML State machines, and (2) Matlab Stateflow. Each has a specification of about 100 pages in English, and for some subsets formal specifications exist, but are documented in journal papers. We have defined a common data structure (an abstract syntax) for the models, and coded the interpreter in pure Java (only the core libraries were used). The code for the abstract syntax part was about 600 lines; functional code common across the two variants required about 250 lines; the Stateflow-specific code had about 600 lines; and the UML State machine variant had about 400 lines. All the code was reviewed by 3-4 programmers and thoroughly tested and compared to existing tools using carefully chosen examples (models and input/output sequences). Our experience indicates that such interpreter-based specification is feasible, and can be quite compact.

3.3. Challenges

When defining the semantics of DSMLs, several challenges arise, some of which are listed below.

Existence of valid models. One can define an abstract syntax with very restrictive well-formedness constraints, such that no valid models can be constructed. In the case of a complex DSML, it may become a challenge to recognize such a problem.

Existence of valid models that generate an acceptable behavior. A secondary problem is to verify if a valid model exists that generates an acceptable behavior, which, for instance satisfies certain properties (e.g., deadlock freedom). It is a defect of the semantics definition if such a model cannot be constructed.

Composability. In a project, multiple DSMLs are often used. Syntactic composition can be simple, but composition of semantics needs to be investigated more thoroughly as a core research topic.

Efficiency of verification with interpreters. The interpreter-based method has a shortcoming: the system has much more states than the original model, so its verification is more complex. We need techniques to introduce abstractions over the states of an interpreter-based system to reduce the complexity.

Reusability. One goal of the semantic units was reusability, and the same applies to the interpreter-based approach. We were able to take advantage of the features of the implementation language (namely, inheritance and polymorphism) when developing the interpreters for the Statechart variants, but the question arises regarding how this can be extended to other cases.

Dissemination. Definition of the semantics for a DSML must be published in a form supporting review by the stakeholders. A key research question regards the best way of disseminating or sharing such specifications.

4. A DSML with Metamodel-Based Semantics

Recent advances in unified communication, mobile technology, and the desire for collaborators from geographically dispersed teams to coordinate their communication activities are becoming commonplace. There is a strong demand for an easy and flexible way of building user-centric communication services that effectively shields users of these systems from the heterogeneity of communication technologies, and that supports the dynamic nature of communication-based collaboration. Many existing communication service frameworks are custom-built, inflexible, costly, and technology specific. They provide little or no support for user-driven specification, adaptation and coordination of communication services performed in response to changes in highly dynamic environments (e.g., those found in disaster management and healthcare).

To address the aforementioned problems, Deng et al. [14] proposed the *Communication Virtual Machine technology* which consists of an interpreted DSML, the *Communication Modeling Language (CML)*, and a semantic rich platform to execute the communication models, the CVM. In this section we present an extension of CML called the *Workflow Communication Modeling Language (WF-CML)* that better supports the dynamic coordination of communication services. WF-CML defines communication-specific abstractions of workflow concepts found in many of the major general-purpose workflow languages, including UML activity diagrams [41], YAWL [1], and Windows Workflow Foundation [39]. The definition of WF-CML includes the metamodel and the dynamic semantics. Due to space limitations, we only present a subset of the metamodel and an overview of the dynamic semantics, yet to be completed.

4.1. Motivating Scenario

To further motivate the need for WF-CML, we present a scenario developed at the Miami Children's Hospital [8]. The following are the actors in the scenario: A Discharge Physician (DP), a Senior Clinician (SC), a Primary Care Physician (PCP), a Nurse Practitioner (NP) and the Attending Physician (AP). Patient Discharge Scenario:

(1) On the day of discharge, Dr. Burke (DP) establishes an audio communication with Dr. Monteiro (SC) to discuss the discharge of baby Jane. During the conversation, Dr. Burke composes a discharge package, `DisPkg_1`, referred to as a *form*, and sends it to Dr. Monteiro to be validated. The `DisPkg_1` form consists of a `RecSum-Jane.txt` (text file), summary of patient's condition; `xRay-Jane.jpg`, an x-Ray of the patient's heart, (non-streamfile); and a `HeartEcho-Jane.mpg` (video clip), an echocardiogram (echo) of the patient's heart. After `DisPkg_1` is sent, Dr. Burke contacts Dr. Sanchez (PCP) to join the conversation with Dr. Monteiro to discuss the

patient's condition. During the conversation, Dr. Monteiro validates `DisPkg_1` and sends it to Dr. Burke.

(2) Since the form `DisPkg_1` is received within 24 hours and is validated, Dr. Burke then sends it to Nurse Smith (NP) and Dr. Wang (AP) (If the form had not been validated and received within 24 hours, the workflow requires that Dr. Burke send out an interim discharge note (`InterimNote_1`)). At the same time, Dr. Burke continues his conference with Drs. Monteiro and Sanchez.

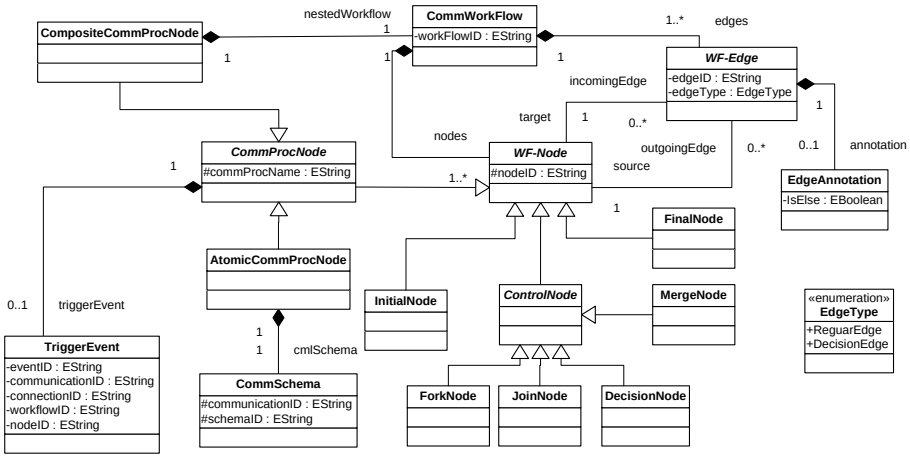


Fig. 4. Partial Abstract Syntax for WF-CML

4.2. Metamodel

The metamodel for WF-CML consists of the abstract syntax, represented as a UML class diagram, and the static semantics defined using OCL. Figure 4 shows a partial class diagram of the abstract syntax for WF-CML. The complete class diagram and static semantics can be found on the project's web page[1].

A WF-CML model is a graph (`CommWorkFlow`) consisting of nodes (`WF-Node`), edges (`WF-Edge`), and trigger events (`TriggerEvent`) as shown in Figure 4. The nodes are described as follows: `InitialNode` and `FinalNode` – signify the beginning and ending of a model representing the coordination of communication processes. `CommProcNode` (communication process node) - is either an atomic communication model (`AtomicCommProcNode`) or a nested workflow model (`Composite-CommProcNode`) and has zero or one trigger event associated with the node. The atomic communication model represents a model created using pre-

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

workflow CML. `DecisionNode`, `ForkNode`, `JoinNode` and `MergeNode` - express control flow between communication processes. There are two types of edges (decision and regular). A decision edge is annotated with zero or more atomic events. If there is no event annotation on the decision edge, it is considered an *else* edge.

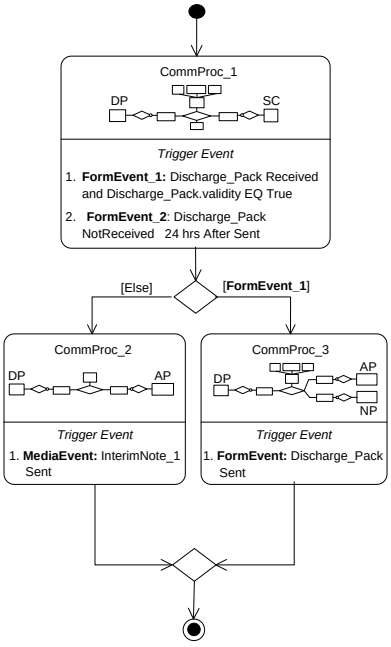


Fig. 5. WF-CML Model for Scenario

Figure 5 shows the WF-CML model for the scenario described in the previous section. The CML model in `CommProc_1`, top node in the figure, specifies the communication between the DP and the SC, the user ids and names are instantiated when the WF-CML model is executed by Dr. Burke, and he loads the contact information for the SC. There are two types defined for this communication, a form type (`Discharge_Pack`) and a built-in media type (`LiveAudio`). The trigger event in `CommProc_1` states that this node is exited when a validated patient form of type `Discharge_Pack` is received, in this case `DisPkg_1`, and it is validated; or the patient form is not received 24 hours after being sent.

4.3. Dynamic Semantics

The semantic rules of WF-CML extend the semantic rules for CML [54]. We first provide an overview of the semantic rules for realizing CML models followed by the semantics rules for WF-CML models.

CML:

$((CI_{in}, DI_{in}), CSP_Env_i) => ((CI_{out}, DI_{out}), Script_{out}, Event_{out}, CSP_Env_{i+1})$

where:

(CI_{in}, DI_{in}) - input control and data instances capturing a user's communication needs to be realized by the communication service.

CSP_Env_i - state of the CS process including the state of the executing control and data instances, (CI_i, DI_i), negotiation state, Neg_i, and media transfer state, MT_i.

(CI_{out}, DI_{out}) - updated control and data instances generated during the transition.

$Script_{out}$ - communication control script generated, including (re)negotiation and media transfer scripts, executed by the CVM middleware.

$Event_{out}$ - output event generated during the execution of the CS process, including media events or negotiation events.

CSP_Env_{i+1} - updated environment of the CS process. The structure is similar to CSP_Env_i stated above.

WF-CML:

$(Event_{in}, WF_Env_i) => ((CI_{out}, DI_{out}), WF_Env_{i+1})$

where:

$Event_{in}$ - an input event that may trigger the execution of the next node in the WF-CML model. These events include negotiation events, data transfer events and exception events.

WF_Env_i - the current configuration of a process executing the WF-CML model (WF_Proc). Its state is defined as $(WF_{exec}$, CS_Procs, Curr_CS),

where:

WF_{exec} - the currently executing WF-CML model in the WF_Proc process.

CS_Procs - a list of executing CS processes in the executing WF_Proc process.

Curr_CS - currently active CS processes with respect to the WF Proc process.

$WF\ Env_{i+1}$ - the updated configuration of the WF Proc process.

The rules describing the semantics for CML and WF-CML models may be applied to the motivating scenario presented in Section 4.1 as follows. The WF-CML model is processed using the semantics rule for WF-CML and shown in Tables 1 and 2. Table 1 shows the left-hand side of the rule and Table 2 the right-hand side of the rule. The input to the rule, shown in the third row in Table 1 (i.e., when i = 0), includes: (1) the null event, and (2) the workflow environment (WF_{exe}). The current workflow environment includes: (a) the WF-CML model shown in Figure 5, (b) the list of executing processes (`CS_Procs`), which is empty, and (c) the currently active CS processes in the workflow (`Curr_CS`), which is null. The output of the rule includes: (1) the control instance and data instance pair (CI, DI) to be processed by the CML semantic model, (2) the currently executing WF-CML model (Figure 5), (3) the

list of executing processes which is `Comm_Proc_1`, the top node in Figure 5, and (4) the currently active node in the WF-CML model, `Comm_Proc_1`.

Table 1. Left-hand side of the semantic rule used for WF-CML.

i	$Event_{in}$	WF_Env_i		
		WF_{exec}	CS_Procs	Curr_CS
0	null	WF-CML model (see Figure 5)	empty	null
…				
k	FormEvent_1	WF-CML model (see Figure 5)	CommProc_1 (see Figure 5, top node)	CommProc_1 (see Figure 5, top node)
…				

Table 2. Right-hand side of the semantic rule used for WF-CML.

i	(CI_{out}, DI_{out})	WF_Env_{i+1}		
		WF_{exec}	CS_Procs	Curr_CS
0	(M_1, null)	WF-CML model (see Figure 5)	CommProc_1 (see Figure 5, top node)	CommProc_1 (see Figure 5, top node)
…				
k	(M_p, null)	WF-CML model (see Figure 5)	CommProc_1, CommProc_3 (see Figure 5)	CommProc_3 (see Figure 5, bottom right)
…				

The (CI, DI) model pair extracted from the WF-CML model is processed by the semantic rule for CML. The left-hand and right-hand sides of the rule are shown in Tables 4 and 5, respectively. Table 3 shows some of the CML models used during the realization of the communication. The third row of Table 4, where j = 0, shows the input model pair of (M_1, null). Table 3 shows that M_1 is a model representing the communication between two persons and the connection (C1) supports the transmission of live audio and a patient discharge form. The media and form types on the connection are not labeled. We use the pair (null, null) in Table 4 to represent the initial models in the system. The initial states for the negotiation and media transfer state machines are the negotiation ready state (`Neg_Ready`) and the media transfer ready state (`MT_Ready`), respectively. After applying the rule, Table 5 shows the output generated and the updated state of the system. The models generated are the same as the input models because these models are used during negotiation; the script generated creates a connection with the remote party in the connection, Dr. Monteiro, and sends the control model (M_1); the event generated (`Neg_Initiated`) reflects that negotiation has started. The entries in the table for j=1 and j=2 represents the negotiation process. The application of the rule shown in Tables 4 and 5 with the row labeled j=2 shows the application of the rule to enable live audio.

Table 3. Some of the CML models used in the motivating scenario.

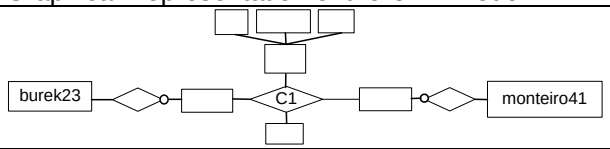
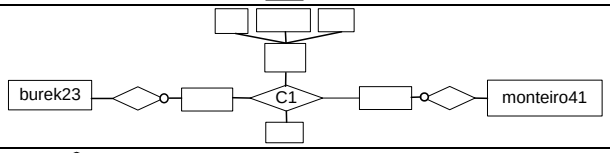
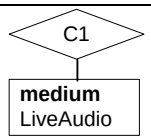
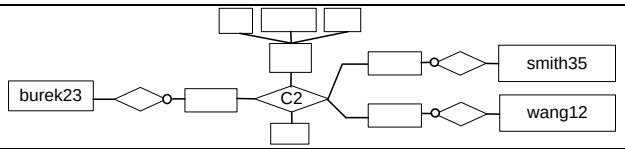
Model ID	Graphical Representation of the CML model
M_1 (control instance)	
M_2 (control instance)	
M_3 (data instance)	
…	
M_p	

Table 4. Left-hand side of the semantic rule used for CML.

j	(CI_{in}, DI_{in})	CSP_Env_i			Comments
		(CI_i, DI_i)	Neg_i	MT_i	
0	$(M_1, null)$	(null, null)	Neg_Ready	MT_Ready	M_1 is the control instance model created by the local participant, Dr. Burke
1	$(M_2, null)$	$(M_1, null)$	WaitingSameCI	MT_Ready	M_2 is the control instance model received by Dr. Burke's CVM from the remote participant, Dr. Monteiro
2	(M_2, M_3)	$(M_2, null)$	Neg_Ready	MT_Ready	M_3 is the model that represents the activation of live audio
…					

During the communication for `CommProc_1` an event will eventually be triggered that moves the workflow onto the next node. In the scenario, the `FormEvent_1` is triggered, as shown in Table 1 row labeled i = k. The right-hand side of the WF-CML rule in Table 2 shows that both `CommProc_1` and `CommProc_3` are now active and the currently active node with respect to workflow is `CommProc_3`. Two communication processes are active since our semantics do not force the termination of a communication after the workflow model moved on to the next node. Note that the control model (M_p) is now processed by the CML semantic rule which establishes a new connection with two participants, Nurse Smith and Dr. Wang.

Table 5. Right-hand side of the semantics rule used for CML.

j	(CI_{out}, DI_{out})	$Script_{out}$	$Event_{out}$	CSP_Env_{i+1}		
				(CI_{i+1},DI_{i+1})	Neg_{i+1}	MT_{i+1}
0	(M_1, null)	createConnection (“C1”); sendSchema (“C1”, “burke23”, “monteiro41”, “M_1, null”)	Neg_ Initiated	(M_1, null)	Neg_ Initiated	MT_ Ready
1	(M_2, null)	sendSchema (“C1”, “burke23”, “monteiro41”, “M_2, null”); addParticipant (“C1”, “monteiro41”)	Neg_ Complete	(M_2, null)	Neg_ Complete	MT_ Ready
2	(M_2, M_3)	enableInitiator (“C1”, “LiveAudio”); sendSchema (“C1”, “burke23”, “monteiro41”, “M_2, M_3”)	Enable_ Stream	(M_2, M_3)	Neg_ Ready	Stream_ Enabled
…						

4.4. Challenges

WF-CML supports the execution of communication models in a distributed environment, where participants in the communication are allowed to change the currently executing communication process. The complexity of executing WF-CML models directly provide us with the following challenges: (1) What notation should be used to define the dynamic semantics (e.g., operational, denotational, or axiomatic)? (2) How to define the environments for a communication process and workflow process? (3) How can the semantics be extended to support dynamic adaptation of the WF-CML?

5. Model-Based Verification Tools

MDE provides a context in which formal specification and verification techniques can be applied. There is evidence that this is already taking place (e.g., see [11], [23], [30], [34], [47]). With respect to the UML, in the late nineties the precise UML (pUML) group helped raise awareness of the need for more formal descriptions of UML semantics to enable rigorous analysis of structural and functional properties of systems captured in UML models. Over the last decade, we have seen a significant number of papers on using relatively mature formal verification techniques to analyze properties described in particular UML models (e.g., there has been significant work on using model checking techniques to analyze UML state machine models, and Petri net variants to analyze activity models).

Despite the focused attempts, there are very few UML-based verification tools that can be described as usable by practitioners. In the following, we discuss some of the opportunities for applying verification techniques in MDE and discuss some of the challenges. For the most part, the opportunities and challenges are presented in terms of UML modeling issues, primarily because this is one of the more widely used (and misused) MDE languages, and there is a dire need for practical UML-based verification tools.

5.1. Towards Usable UML-based Verification Tools

The UML has reached a level of maturity that now allows us to reach for some of the lower hanging fruit (not necessarily the same as low-hanging fruit!) where application of rigorous verification techniques are concerned. One of the frustrating experiences that a modeling student or practitioner learning a language such as the UML goes through is determining if his/her model is, in some sense, a valid description. In the case of students, the only feedback that they often receive is the instructor's grade of their work. There is a need to provide modelers, in particular, UML modelers, with some means of checking the validity of their model.

An obvious approach is to provide some support for executing or animating models. The Colorado State University (CSU) UMLAnT (UML Animation and Testing) tool provides a means for dynamically analyzing (testing) UML design models. A UMLAnT design model consists of class diagrams with operations specified in a Java-like action language called JAL [15]. UMLAnT is an Eclipse plug-in that provides support for (1) generating test inputs that satisfy criteria based on coverage of elements in a sequence diagram that describes the scenarios that will be exercised in a test, (2) executing the design model using test inputs (a test input is an operation with parameter values), and (3) showing execution progress in terms of sequence diagrams and changes to object configurations. We are currently updating the tool to the latest version of Eclipse and improving its robustness.

We are also developing lightweight scenario-based analysis techniques that allow developers to check whether a scenario describing a desired or undesired behavior is supported by a model [56]. The technique provides a less expensive way of analyzing a system in the cases where exhaustive formal analysis is not possible or cost-effective. In the approach we are developing, a behavior is described as a sequence of snapshots, where a snapshot is an object configuration that conforms to a class diagram. A class model with operations specified in the OCL is transformed to a class model, called a Snapshot Model, that characterizes all possible behaviors (sequences of snapshots). A verifier then provides scenarios (expressed as sequence diagrams) and the analysis tool we are developing checks whether these scenarios conform to the Snapshot Model.

One of the problems that our analysis approaches and those developed by other researchers face is that they do not handle incomplete models well. This is one of the challenges that we are currently tackling in our analysis work. Another aspect that requires attention is ensuring consistency of behavioral and structural concepts across different modeling views. This is a particularly challenging problem in the UML, and is sometimes one of the reasons practitioners limit their use to one or two UML diagram types (typically class diagrams, sequence diagrams or state machine diagrams). One of the problems that hinders research in this area is the size of the UML language (as reflected in its metamodel) - this makes it very difficult to determine precisely the consistency relationships that must hold across elements in different diagrams. Furthermore, it has not been verified that the UML metamodel is a valid description that can be relied upon correctly to define these relationships. A good usability challenge problem for verification tools is finding an answer to the question “is the UML metamodel correct?”

5.2. Formal Verification Challenges: Transformations, Semantic Variations, and Models@Run.Time

The previous subsection identified some obvious opportunities for applying verification techniques in the MDE context. That was just the tip of the iceberg; there are other more challenging verification problems that should be tackled in MDE. A challenging problem concerns verification of model transformations [35]. In a recently published paper on testing model transformations, we highlighted some of these challenges [4]. One of the major problems concerns generating an adequate set of test models. Generating test inputs for programs that use inputs with simple structures is challenging in itself; when the inputs are models with complex structures the challenges are greater.

Another problem that must be considered is the variety of semantics that can be associated with languages such as the UML. In the UML, some parts of the semantics are intentionally left undefined to allow users to tailor semantics to their needs. While formal methods purists may argue for defining a single semantics for the UML, the practical reality is that different groups

use the UML differently, and this need must be supported. It is highly unlikely that a single verification approach would meet all structural, functional and behavioral analysis needs. To tackle this problem we have started a research initiative called GeMoC (Generic Model of Computation) with the goal of developing a verification framework that can be used in a modeling environment that supports a variety of semantics (or models of computation).

An emerging MDE research area that attempts to extend the use of models to runtime management is models@run.time [6]. There has been significant work on using models to support runtime adaptation of software. Verifying adaptations at runtime is a particularly challenging problem that groups working in this area are currently addressing.

6. Semantics-Based Tools in Domain-Specific Modeling

As mentioned in Section 2, a formal description of a modeling language allows for the automatic generation of supporting tools that are based on the modeling language semantics. This section motivates the need for such tool generation by summarizing our previous work in generating debuggers and testing engines for domain-specific languages (DSLs) [31], [37]. Our framework (Figure 6) for automatic generation of DSL debuggers and test engines reuses existing GPL tools [55]. The framework consists of a mapping process that records the correspondence between the DSL program and the generated GPL code, a tool methods mapping that specifies how DSL tool actions are mapped to GPL tool actions (e.g., a DSL debugging command might request execution of several GPL debugging commands), and a tool results mapping, which specifies how obtained results should be displayed to the end-user using only DSL abstractions.

Existing approaches for defining the formal semantics of programming languages can be used to specify the semantics of DSMLs. However, a critical point of this work is that a semantics definition should be model-based. To fulfill this objective and accomplish transparency of low-level formalisms, three steps are followed. The first step focuses on the methodology to specify state transitions to show dynamic behavior of meta-elements. The second step concerns the visual language to control the sequence of the defined state transitions and runtime configurations. The third step includes transformation of specifications into the different language-based tools. The combination of all outcomes of these steps will form the semantic framework. Figure 7 shows an outline of the approach. The first part of the figure demonstrates abstract syntax and static semantic definitions; current platforms provide a means for specifying these definitions. The second part depicts the dynamic semantics specification technique based on activity diagrams and graph grammars. These tools are used to define a sequence of state transitions. The last part shows specification of verification properties within domain boundaries. Finally, all these specifications can be transformed into the different language-based tools (e.g., interpreter, code generator, simulator, verifier).

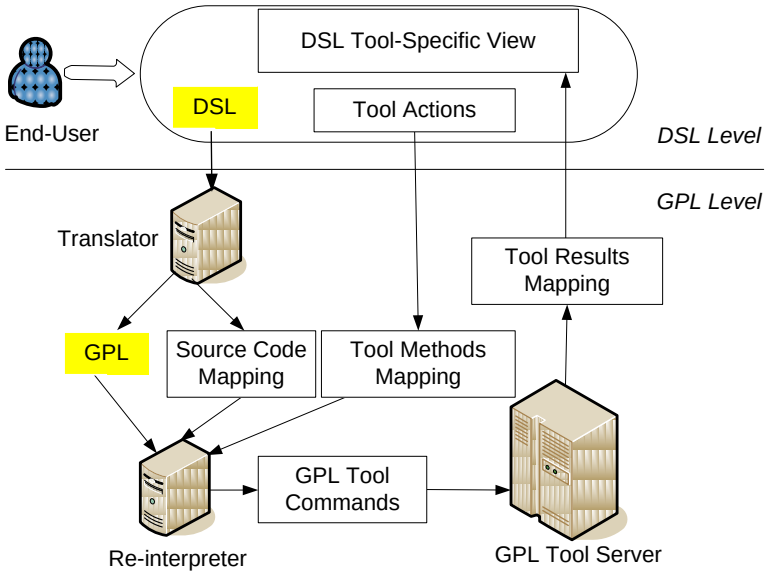


Fig. 6. The framework for automatic generation of DSL tools

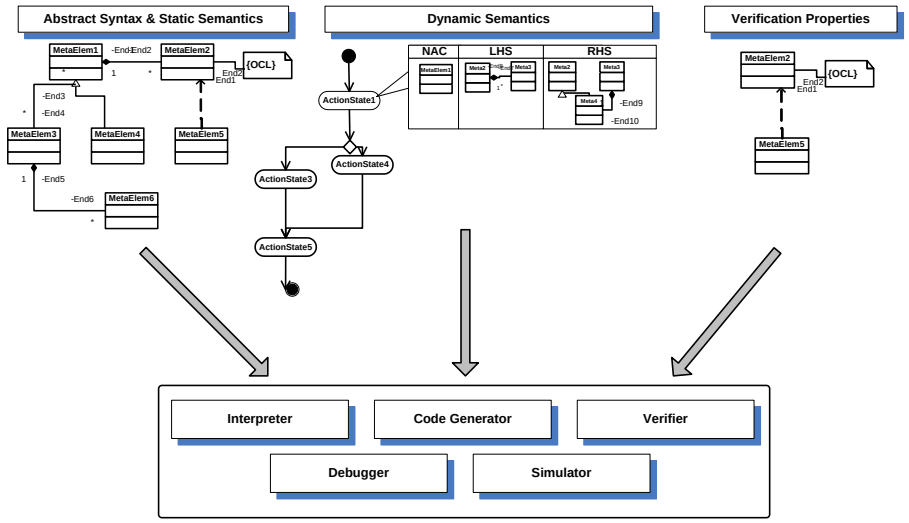


Fig. 7. Semantics-Based Tool Generation

In [13], we performed some experiments on semi-automatic generation of tools for modeling languages and focused on how to specify the behavioral semantics of a DSML by a sequence of graph transformation rules, enabling transformation of a modeling language specification into the model checking tool Alloy [28]. In our initial study, we demonstrated specification of sequential system semantics that connects the initial model to possible result models. First, we focused on how to specify the behavioral semantics of a DSML by a

sequence of graph transformation rules. While each graph transformation rule represents a state change of a sequential system, a sequence of state changes is defined by an activity diagram. Sequence definitions control what state transition is to be fired, in what order, and what condition. All these steps are mapped into a transition system that is used to generate a state space. We provided an example to demonstrate semantics definition of a DSML and verification of an assertion in one of the model checking tools (i.e., Alloy). The activities investigated in our initial work can be summarized by the following items:

1. mapping metamodel elements to Alloy abstract signatures,
2. mapping model elements to Alloy concrete signatures,
3. mapping graph transformation rules to Alloy predicates, and
4. mapping verification tasks to Alloy asserts.

Abstract signatures are used to define the meta-layer of the models. To define a model layer in Alloy, these abstract signature definitions are extended into concrete signatures. Each model element is mapped into an appropriate concrete signature in Alloy. Behavioral specifications, which we define by means of graph transformation rules, are mapped into Alloy predicates. Each task defined in a semantics definition is transformed into an Alloy predicate having two parameters, g and g', representing the current state and the next state. Finally, the assertions that would be satisfied at the final states are transformed into Alloy assert definitions.

Although our current investigation was performed manually, it demonstrated how DSML designers can define semantic and verification specifications using visual models. We are currently investigating how to generalize and automate this process.

7. Conclusions

DSMLs allow end-users and domain experts to specify the core essence of a problem using visual abstractions that are close to the problem space of a specific domain. A key research challenge in the adoption of such modeling languages concerns the manner in which the semantics of each DSML is specified. Typically, the behavioral semantics of a DSML is described within individual hard-coded model interpreters. Such a representation of the semantics is not specified in a manner that is ameliorable to formal analysis and generation of model-based tools. As such, the utility of a DSML is hampered due to the lack of a single representation that formally denotes the semantics of the language. This paper has described several research projects that investigate and develop a formal, yet widely usable, means to specify DSML semantics. Our future work is automatic generation of model interpreters, simulators, debuggers and verifiers from such semantic specifications, which would have significant impact on the current practice of model-driven engineering in terms of automating many tasks that are currently done ad hoc in a manual hand-crafted manner.

Acknowledgments. This work was supported in part by United States National Science Foundation awards CAREER CPA-1052616, CCF-0811630, CCF-1018711, HRD-0833093, OISE-0730065, and OISE-0968596, and programs by the Defense Advanced Research Project Agency: the Evolutionary Design of Complex Systems and Model-based Integration of Embedded Systems, by the National Science Foundation's ITR program, by Boeing and General Motors through the ESCHER initiative, and NASA's Robust Software Engineering program.

References

1. van der Aalst, W. M. P., ter Hofstede, A. H. M.: YAWL: Yet Another Workflow Language. Information Systems, Vol. 30, No. 4, 245–275. (2003)
2. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Language for Model Transformations. Journal on Software and System Modeling, Vol. 5, No. 3, 261–288. (2006)
3. Álvarez, J. M., Evans, A., Sammut, P.: Mapping between Levels in the Metamodel Architecture. In: Gogolla, M., Kobryn, C. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 2185. Springer-Verlag, New York, 34-46. (2001)
4. Baudry, B., Ghosh, S., Fleury, M., France, R., La Traon, Y., Mottu, J.-M.: Barriers to Systematic Model Transformation Testing. Communications of the ACM, Vol. 53, No. 6, 139-143. (2010)
5. von der Beeck, M.: A Comparison of Statecharts Variants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.): Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, Vol. 863. Springer-Verlag, Berlin, 128–148. (1994)
6. Blair, G., Bencomo, N., and France, R. R.: Models @ run.time. Computer, Vol. 42. No. 10, 22–27. (2009).
7. Börger, E.: The Origins and Development of the ASM Method for High-Level System Design and Analysis. Journal of Universal Computer Science, Vol. 8, No. 1, 2-74. (2002)
8. Burke, R. P., White, J. A.: Internet Rounds: A Congenital Heart Surgeon's Web Log. Seminars in Thoracic and Cardiovascular Surgery, Vol. 16, No. 3, 283–292. (2004)
9. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.): Model Driven Architecture - Foundations and Applications. Lecture Notes in Computer Science, Vol. 3748. Springer-Verlag, Berlin, 115-129. (2005)
10. Chen, K., Sztipanovits, J., Neema, S.: Compositional Specification of Behavioral Semantics. In Proceedings of DATE '07, Design, Automation and Test in Europe, IEEE, Nice, France, 906-911. (2007)
11. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using the OCL Environment. Electronic Notes in Theoretical Computer Science, Vol. 102, 99 – 110. (2004)
12. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. International Journal on Software Tools for Technology Transfer, Vol. 2, 2000. (2000)
13. Demirezen, Z., Mernik, M., Gray, J., Bryant, B. R.: Verification of DSMLs Using Graph Transformation: A Case Study with Alloy. In Proceedings of the 6th

International Workshop on Model-Driven Engineering, Verification and Validation, ACM, Denver, Colorado. (2009)
14. Deng, Y., Sadjadi, S.M., Clarke, P. J., Hristidis, V., Rangaswamy, R., Wang, Y.: CVM - A Communication Virtual Machine. Journal of Systems Software, Vol. 81, No. 10, 1640–1662. (2008)
15. Dinh-Trong, T T., Ghosh, S., France, R. B.: A Systematic Approach to Generate Inputs to Test UML Design Models. In Proceedings of ISSRE '06, the 17th International Symposium on Software Reliability Engineering, IEEE, Raleigh, North Carolina, 95–104. (2006)
16. Di Ruscio, D., Jouault, F., Kurtev, I., Bezivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report, INRIA/LINA, http://hal.archives-ouvertes.fr/ccsd-00023008/en. (2006)
17. Ducasse, S., Girba, T., Kuhn, A., Renggli, L.: Meta-Environment and Executable Meta-Language using Smalltalk: An Experience Report. Software and Systems Modeling, Vol. 8, No. 1, 5-19. (2009)
18. Eker, J., Janneck, J. W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming Heterogeneity - The Ptolemy Approach, Proceedings of the IEEE, Vol. 91, No. 1, 127-144. (2003)
19. Engels, G., Hausmann, J., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 1939. Springer-Verlag, New York, 323-337. (2000)
20. Ermel, C., Holscher, K., Kuske, S., Ziemann, P.: Animated Simulation of Integrated UML Behavioral Models Based on Graph Transformation. In Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE Computer Society, Dallas, Texas, 125-133. (2005)
21. Esser, R., Janneck, J.W.: Moses - A Tool Suite for Visual Modeling of Discrete-Event Systems. In Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01), IEEE Computer Society, Stresa, Italy, 272-279. (2001)
22. Gargantini, A., Riccobene, E., Scandurra, P.: A Semantic Framework for Metamodel-Based Languages. Automated Software Engineering, Vol. 16, No. 3, 415-454. (2009)
23. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling, Vol. 4, No. 4, 386-398. (2005)
24. Gray, J., Tolvanen, J.P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. In Fishwick, P. A. (ed.): Handbook of Dynamic System Modeling, CRC Press, Boca Raton, Florida. (2007)
25. Hahn, C.: A Domain Specific Modeling Language for Multiagent Systems. In Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems, Estoril, Portugal, 233-240. (2008)
26. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"?, Computer, Vol. 37, No. 10, 64-72. (2004)
27. Henriques, P. R., Pereira, M. J. V., Mernik, M., Lenič, M., Gray, J., Wu, H.: Automatic Generation of Language-Based Tools using the LISA System. IEE Proceedings Software, Vol. 152, No. 2, 54-69. (2005)
28. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, 256-290. (2002)

29. Kelly, S., Tolvanen, J-P.: Domain-Specific Modeling: Enabling Full-Code Generation, John Wiley and Sons. (2008)
30. Knapp. A.: A Formal Semantics for UML Interactions. In: France, R. B., Rumpe, B. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 1723. Springer-Verlag, New York, 116–130. (1999)
31. Kosar, T., Oliveira, N., Mernik, M., Varanda Pereira, M. J., Črepinšek, M., da Cruz, D., Henriques, P. R.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
32. de Lara, J., Vangheluwe, H., Alfonseca, M.: Metamodelling and Graph Grammars for Multi-Paradigm Modelling in AToM 3. Software and Systems Modeling, Vol. 3, No. 3, 194-209. (2004)
33. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments, Computer, Vol. 34, No. 11, 44-51. (2001)
34. Lilius, J. and Porres Paltor, I. Formalising UML State Machines for Model Checking. In: France, R. B., Rumpe, B. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 1723. Springer-Verlag, New York, 430–445. (1999)
35. Lin, Y., Zhang, J., Gray, J.: A Testing Framework for Model Transformations. In Beydeda, S., Book, M., Gruhn, V. (eds.), Model-driven Software Development, Springer, Heidelberg, Germany, 219-236. (2005)
36. Mathworks: Matlab Simulink/Stateflow Tools, http://www.mathworks.com (2010)
37. Mernik, M., Heering, J., Sloane, A. M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, No. 4, 316-344. (2005)
38. Microsoft Corporation: The Abstract State Machine Language, http://research.microsoft.com/en-us/projects/asml. (2010)
39. Microsoft Corporation: Windows Workflow Foundation, http://msdn.microsoft.com/en-us/vbasic/cc506054 (2010)
40. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-Languages. In: Briand, L. C., Williams, C. (eds.): Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, Vol. 3713. Springer-Verlag, Heidelberg, Germany, 264-278. (2005)
41. Object Management Group. Unified Modeling Language: Superstructure, Version 2, http://www.omg.org/spec/UML/2.3. (2010)
42. Porubän, J., Forgáč, M., Sabo, M., Běhálek, M.: Annotation Based Parser Generator. Computer Science and Information Systems, Vol. 7, No. 2, 291-307. (2010)
43. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and Tool Support for Model Driven Engineering with Maude, Journal of Object Technology, Vol. 6, No. 9, 187-207. (2007)
44. Sadilek, D. A., Wachsmuth, G.: Using Grammarware Languages to Define Operational Semantics of Modelled Languages. In: Oriol, M., Meyer, B. (eds.): Objects, Components, Models and Patterns, Lecture Notes in Business Information Processing, Vol. 33. Springer-Verlag, Heidelberg, Germany, 348-356. (2009)
45. Scheidgen, M., Fischer, J.: Human Comprehensible and Machine Processable Specifications of Operational Semantics. In: Akehurst, D. H., Vogel, R., Paige, R. F. (eds.): Model Driven Architecture - Foundations and Applications, Lecture Notes in Computer Science, Vol. 4530. Springer-Verlag, Heidelberg, Germany, 157-171. (2007)

46. Schmidt, D. C.: Guest Editor's Introduction: Model-Driven Engineering. Computer, Vol. 39, No. 2, 25-31. (2006)
47. Shah, S., Anastasakis, K., Bordbar, B.: From UML to Alloy and Back Again. In Proceedings of MoDeVVa '09, the 6th International Workshop on Model-Driven Engineering, Verification and Validation, ACM, Denver, Colorado, USA, 1–10. (2009)
48. Smith, G.: The Object-Z Specification Language, Kluwer Academic Publishers. (2000)
49. Soden, M., Eichler, H.: Towards a Model Execution Framework for Eclipse. In Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture, ACM, Enschede, Netherlands, 1-7. (2009)
50. Sprinkle, J., Mernik, M., Tolvanen, J.-P., Spinellis, D.: Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software, Vol. 26, No. 4, 15-18. (2009)
51. Sunyé, G., Pennaneac'h, F., Ho, W.-M., Le Guennec, A. and Jézéquel, J.-M.: Using UML Action Semantics for Executable Modeling and Beyond. In: Dittrich, K. R., Geppert, A., Norrie, M. C. (eds.): Advanced Information Systems Engineering, Lecture Notes in Computer Science, Vol. 2068. Springer-Verlag, Heidelberg, Germany, 433-447. (2001)
52. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. Computer, Vol. 30, No. 4, 110-111. (1997)
53. Varró, D.: A Formal Semantics of UML Statecharts by Model Transition Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.): Graph Transformation, Lecture Notes in Computer Science, Vol. 2505. Springer-Verlag, Heidelberg, Germany, 378-392. (2002)
54. Wang, Y., Wu, Y., Allen, A., Espinoza, B., Clarke, P.J., Deng, Y.: Towards the Operational Semantics of User-Centric Communication Models. In Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, vol.1, pp.254-262. (2009)
55. Wu, H., Gray, J., Mernik, M.: Grammar-Driven Generation of Domain-Specific Language Debuggers. Software: Practice and Experience, Vol. 38, No. 10, 1073-1103. (2008)
56. Yu, L., France, R. B., Ray, I.: Scenario-Based Static Analysis of UML Class Models. Model-Driven Engineering Languages and Systems. Lecture Notes in Computer Science, Vol. 5301. Springer-Verlag, New York, 234–248. (2008)

Barrett R. Bryant is Professor and Associate Chair of Computer and Information Sciences at the University of Alabama at Birmingham (UAB). He received his B. S. in computer science from the University of Arkansas at Little Rock in 1979 and his Ph. D. in computer science from Northwestern University in 1983, after which he joined UAB. His research interests include theory and implementation of programming languages, formal specification of software systems, and component-based software engineering. He is a member of EAPLS, and a senior member of ACM and IEEE.

Jeff Gray received the BSc and MSc degrees in Computer Science from West Virginia University in 1991 and 1993, and the Ph.D. in Computer Science from Vanderbilt University in 2002. He is currently Associate Professor of Computer Science at the University of Alabama. Jeff's research

interests are in the general areas of software engineering and programming languages, and in the specific areas of model-driven engineering, aspect orientation, and software evolution. He is a member of the IEEE and ACM.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998, respectively. He is currently Professor of Computer Science at the University of Maribor. He is also Visiting Professor of Computer and Information Sciences at the University of Alabama at Birmingham, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modeling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Peter J. Clarke received his BSc. degree in Computer Science and Mathematics from the University of the West Indies (Cave Hill) in 1987, MS 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 software testing, software metrics, model-based testing, model-driven software development and domain-specific modeling languages. He is currently Associate Professor of Computing and Information Sciences at Florida International University. He is a member of the ACM (SIGSOFT, SIGCSE, and SIGAPP); IEEE Computer Society; and a member of the Association for Software Testing (AST).

Robert France is Professor of Computer Science at Colorado State University. His research interests are in the area of Software Engineering, in particular formal specification techniques, software modeling techniques, design patterns, and domain-specific modeling languages. He is an editor-in-chief of the Springer journal on Software and System Modeling (SoSyM), a Software Area Editor for IEEE Computer, and is a past Steering Committee Chair of the MoDELS/UML conference series. He was also a member of the revision task forces for the UML 1.x standards. He was awarded the Ten Year Most Influential Paper award at MODELS in 2008.

Gabor Karsai is Professor of Electrical Engineering and Computer Science at Vanderbilt University and a senior research scientist in the Institute for Software-Integrated Systems at Vanderbilt. He conducts research in model-integrated computing. Karsai received a Technical Doctorate degree in Electrical Engineering from the Technical University of Budapest, Hungary, and a PhD in Electrical Engineering from Vanderbilt University. He is a member of the IEEE Computer Society.

Received: January 14, 2011; Accepted: March 11, 2011.