Automatic Generation of Model Traversals from Metamodel Definitions

Tomaž Lukman
Jožef Stefan Institute
Jamova 39, 1000 Ljubljana Slovenia
tomaz.lukman@ijs.si

Marjan Mernik
University of Maribor
Smetanova 17, 2000 Maribor Slovenia
marjan.mernik@uni-mb.si

Zekai Demirezen, Barrett Bryant
University of Alabama at Birmingham
Dept. of Computer and Information Sciences
Birmingham, AL 35294-1170
{zekzek, bryant}@cis.uab.edu

Jeff Gray
University of Alabama
Department of Computer Science
Tuscaloosa, AL 35487-0290
gray@cs.ua.edu

ABSTRACT

Developing software from models is a growing practice and there exist many model-based tools (e.g., model editors, model interpreters) for supporting model-driven engineering. Even though these tools facilitate the automation of software engineering tasks and activities, such tools are typically engineered manually. In this paper, a simple technique is described that enables automatic generation of model traversals. Semantic rules can be inserted into a traversal algorithm to provide meaning to the modeling language. The combination of automated traversal generation with attached semantic rules can generate a model interpreter that can translate a model into some other representation.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors

General Terms

Algorithms, Languages, Theory.

Keywords

Metamodeling, Semantics, Attribute Grammars, Domain-Specific Languages.

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 [1]. In particular, domain-specific modeling (DSM) is a modeling approach that provides languages that fit the domain of an end-user by offering intentions, abstractions, and visualizations for domain concepts [2]. These languages can either be textual, which we will refer to as domain-specific languages (DSLs); or visual, which we will refer to as domain-specific modeling languages (DSMLs). Computer scientists may also benefit from the higher abstractions provided by DSMLs (e.g., describing the deployment of an application with a modeling language, compared to handcrafting thousands of lines of XML to represent the same intention).

However, the potential for impact of DSM is reduced due to the imprecise nature in which such languages are defined. The large majority of such 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 language, but also limits opportunities for automating the generation of various language tools (much like the adoption of grammars provided a reference point for generation of compiler and other tools for programming languages).

ACMSE '10, April 15-17, 2010, Oxford, MS, USA.

The advantages of formal specification of programming language semantics are well-known [3, 4]. The meaning of a program can be precisely and unambiguously defined, which offers a unique possibility for automatic generation of compilers or interpreters. In our previous work [5, 6], we investigated how the attribute grammar specification formalism for programming languages can be used to generate many other language-based tools, such as: syntax-directed editors, visualizers, animators, debuggers, and testing tools. In most of these cases, the core language definition must be augmented with tool-specific information (e.g., mapping information in debuggers). In other cases, a fragment of a formal language definition (e.g., regular definitions for language knowledgeable editors) is enough for automatic tool generation. It is also possible to extract implicit information from the formal language definition (e.g., dependencies among attributes in semantic functions for a dependency graph viewer) to automatically generate the desired tool. This earlier work was particularly important in identifying generic (fixed) and specific (variable) parts from which language-based tools can be automatically generated [5]. The approach was applied to GPLs (general-purpose languages), as well as to domain-specific languages (DSLs) [7].

Unfortunately, the formal specification of the syntax and semantics of modeling languages have not matured deeply. While the syntax of modeling languages is commonly specified by metamodels, an appropriate and standard formalism for specifying (behavioral) semantics of modeling languages does not yet exist. This is an important research topic in MDE and several proposals have already emerged [8-10] that focus on the first aforementioned benefit (i.e., that the meaning of a model is precisely and unambiguously defined), while no proposal exists yet for automatic generation of model interpreters, debuggers, simulators and verification tools.

In this paper, we describe our investigation into a formalism to specify the semantics of modeling languages from which different language-based tools (e.g., interpreter, code generator) can be automatically generated. Particularly, we introduce a technique for automatic generation of model interpreters. Building model-based tools from scratch is time consuming and error prone, which makes maintenance very costly. From our experience and that reported by others from industry, building model interpreters for various DSMLs requires significant effort [11, 12]. Moreover, to be productive, a software developer needs other tools (e.g., simulator, verifier, debugger, etc.). The lack of appropriate tools may cause newly developed DSMLs to become obsolete.

The next section outlines the literature in several categories that relate to the topic of the paper. The core of the paper is contained in Section 3, which introduces the specific ideas of our approach. A small case study is discussed in Section 4, followed by concluding comments in Section 5.

2. RELATED WORK

Some initial work on the generation of various tools for modeling languages exists. An important basis for the automatic generation of model-based tools is the way that the DSML semantics are defined. Different approaches for defining the semantics of DSMLs have been proposed; these differ in their applicability and potential to leverage automatic (or at least semi-automatic) language tool generation.

A common way of defining the semantics is through *translation semantics*, where the abstract syntax of the main DSML is mapped into the abstract syntax of an existing formal language that has well-defined and understood semantics. An advantage of this approach is that the DSML can reuse 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 demanding to correctly map the constructs of the DSML into the constructs of the target language. Another challenge 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 DSML can understand the result. One concrete approach that uses translation semantics is called *semantic anchoring* [8], which uses the well-known Abstract State Machines (ASM) formalism to define the semantics. This solution maps the abstract syntax of the DSML, which was defined in the GME (Generic Modeling Environment) metamodeling tool [13], 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 Moses toolsuite [14], 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 toolsuite generates animation and debugging tools for visual models. The work presented in [10, 15] 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 [16] present a semantics definition framework based on a transition system, where transitions can be defined with Prolog, Schema or ASM. The framework enables semi-automatic generation of visual interpreters and debuggers.

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 users with a programming background. A well-known representative of this approach is the Kermeta tool [17], which extends an abstract metalayer with an imperative action language to weave a semantic definition within the metamodel. The built-in support for specification of operational semantics in Kermeta enables the automatic generation of simulation and testing tools.

Semantics can also be specified through *rewriting systems,* where the system typically consists of rewrite rules. The existing approaches in this category often employ graph rewriting where the semantics can be specified in an operational fashion through the graphical definition given by graph grammars. Graph rewriting specification was employed in the AToM^3 tool [18, 19], which uses (triple) graph grammars as rewriting rules. One of the interesting features of AToM^3 is that the definition of rewriting

rules is given through concrete syntax that makes semantic specification especially amenable for domain experts. AToM^3 can use graph grammar definitions to generate visual model simulators and implement model optimizations and code generation.

3. EXTENDED METAMODELS

This section introduces a novel approach for specifying the semantics of modeling languages. The basic essence of our idea is to extend a metamodel with the semantic description. This is similar to the idea of attribute grammars [20] in the field of programming languages, where semantics are defined on top of syntax definitions. Our idea is a natural step for MDE engineers and a seamless integration with the current practice of defining the abstract syntax of modeling languages. The proposed approach consists of two parts: 1) the specification of the semantic information in the metamodel, and 2) an automatic synthesis of this information so that model-based tools can be generated. Both of these parts will be presented in the following subsections, with the emphasis on the algorithm that allows automatic traversal of models and the evaluation of the meaning of a specific model.

3.1 Lessons learned from attribute grammars

With attribute grammars, traversal algorithms can range from a simple and non-efficient tree traversal (where nodes are visited as long as non-evaluated attributes exist in a tree), to more sophisticated and efficient traversals (where nodes are visited in a more optimal way). The visiting order can be computed from the fact that an attribute depends on another attribute, which is attached to a different node; this node has to be visited and its attribute evaluated before the attribute that depends on it. Note that circular dependencies among attributes are not allowed. Such algorithms (e.g., for non-circular attribute grammars, ordered attribute grammars [21]) already exist for trees, but have not yet been developed for graphs, which are more challenging due to possible cycles in graph structures.

3.2 Model semantics specification

The semantics of the modeling language that extend the metamodel is given through semantic rules and semantic attributes. The semantics rules carry semantic information that specifies how semantic attributes are calculated when the semantics of a model is being computed. Semantic rules are specified through an action language, which is currently Java. The attributes are used to compute the meaning of the current model. Attribute occurrences in model elements are initially undefined and are then gradually defined and redefined by the traversal algorithm to obtain/compute the meaning of the whole model. Because the semantics of particular metamodel elements will be stored as attribute occurrences in a graph representing a particular model, which conforms to that metamodel, the meaning of the model will be obtained by evaluating attribute occurrences. The semantic description is still compositional because the meaning of subparts will be stored in attribute occurrences representing those subparts. Hence, the meaning of the entire model will be stored in attribute occurrences of the root element, as is the case of attribute grammars. In this manner, a model engineer is relieved from the error prone and effort demanding task of writing model interpreters/code generators, since with such a semantics specification an automatic generation is possible. Moreover, such description would also allow automatic generation of other model-based tools (e.g., debugger, simulator) in the same manner as we developed language-based tools in our previous efforts [5, 22].

3.3 Automatically deriving model traversals

To achieve automatic generation of modeling tools, the key challenge is to develop a suitable traversal algorithm over models (graphs) for evaluating attribute occurrences. This traversal algorithm is dependent on the metamodel, which describes the structure of a modeling language (i.e., concepts in a domain and their relationships), and on the tool API, which provides access to the internal representation of models. Many metamodeling environments (e.g., GME-Generic Modeling Environment [13], GMF-Graphical Modeling Framework [23], and GEMS-Generic Eclipse Modeling System [24]) have different mechanisms to store metamodels and provide different APIs to query and navigate models. Because of this, a general traversal algorithm should consist of a fixed (tool-independent) and a variable (tool-dependent) part. In this manner, our technique could be more easily adopted by existing and future tools. In the next subsections, an initial traversal algorithm for extended metamodels is defined within EMF (Eclipse Modeling Framework), which is central to modeling in the Eclipse platform.

3.3.1 Discussion of the traversal algorithm

The initial traversal algorithm is simple and will be gradually optimized, similar to the case in the now mature field of attribute grammars. The main requirements/challenges for the traversal of a model that should be observed by our algorithm are: (i) all model elements should be visited; (ii) the traversal should not fall into an endless loop, because of the cycles that occur in models; (iii) a clear beginning of a model should be determined, similar to a root in the case of trees; (iv) independent of the fact that models can be disconnected (i.e., some nodes of the model might not be connected to others), a traversal of the model and computation of the meaning should be possible.

When reflecting on our experience in implementing model interpreters, we made several interesting observations. We used these observations in development of our model traversal generation algorithm. The traversal is automatically generated from the information found in the definition of the abstract syntax of the DSML (i.e., found in the metamodel). Because of this fact, each model that conforms to the metamodel can be traversed. The generated traversal satisfies all the requirements stated in the previous paragraph. Due to our experience with the EMF metamodeling environment, we initially proposed an algorithm that works on metamodels specified with ECore, which is an adapted version of the EMOF (Essential Meta-Object Facility) standard for the EMF environment and is widely used in industry and academia. In the future, we plan to extend our technique to other means (facilities/languages) for specifying metamodels.

The major observation we made was that developers navigate through a model based on the nodes (we will refer to them as metaclasses) and connections (we will refer to them as relationships) between these nodes, which are specified in the metamodel. ECore only allows the following metarelationships between two metaclasses (or the same metaclass): unidirectional association, bidirectional association, composition and

generalization. Based on these elements, which are used in the definition of a metamodel and the already mentioned observations, we defined the characteristics (found in a metamodel specified by ECore) that influence the model query/traversal and corresponding patterns of navigation. These characteristics, which are visualized in Figure 1, are described in the following paragraphs.

(a) Root metaclass

A very common characteristic of metamodels defined in ECore is the fact that there is one metaclass, which serves as a container for other metaclasses (i.e., other metaclasses are connected to this metaclass with a composition metarelationship). This metaclass can be considered as the root of the metamodel and the instance of it as the root of the model. The root defines the start of the traversal model. If we visualize such a metamodel via a model editor that was generated by the GMF (Graphical Modeling Framework) tool, then the root metaclass is visualized as the diagram on which the visual representations of the instances of the contained metaclass are displayed.

(b) Unidirectional metarelationship

The traversal on two metaclasses (or one self-related metaclass) that are connected by a unidirectional metarelationship is straightforward. The navigation will follow the path into which the arrow points. The multiplicity on the side of the arrow has to be considered, since it indicates if a list of instances (which can be limited or unlimited) or just an instance (which can be mandatory or optional) has to be traversed.

(c) Bidirectional metarelationship

Bidirectional metarelationships can be considered as an equivalent to two unidirectional metarelationships between two metaclasses (or the same metaclass) in both ways. Therefore, the traversal is executed in both directions.

(d) Composition metarelationship

When encountering composition metarelationships, the traversal will go from the metaclass that is attached to the filled diamond to the metaclass attached to the other end of the metarelationship. Multiplicities must also be considered.

(e) Generalization metarelationship

When dealing with generalization metarelationships, one has to be careful. We decided to traverse the generalization/specialization structure from the most general metaclass found at the top of the inheritance tree to the leaves of the inheritance tree. When dealing with a generalization metarelationship, we refer to the more general metaclass as the supermetaclass and the more concrete metaclass (i.e., the one that inherits from the other) as the submetaclass. The supermetaclass can be navigated/traversed from any other metaclass that is connected to the supermetaclass with a composition (being the contained), a unidirectional metarelationship (being the target), or a bidirectional metarelationship. When considering the actual model, it is necessary to find out if an instance of a supermetaclass is also an instance of any of the submetaclasses that inherit from it. If this is the case, then the submetaclass in the inheritance tree should also be considered for traversal. Because the traversal is carried out from the top of the inheritance tree, there is no need to traverse the inheritance tree bottom-up (i.e., starting from the leaves).

(f) Root metaclass with a self-containing metarelationship

Some metamodels have a root metaclass, which is connected to itself by a composition metarelationship. This characteristic is used to enable a hierarchical decomposition in models that conform to such a metamodel. Such models have a diagram, which has a hierarchy of subdiagrams of the same type, when visualized. The traversal of this characteristic requires a recursive algorithm, which stops when it finds the root of the model. The root of the model is the instance (class) that has no parent instances to which it is connected by a composition.

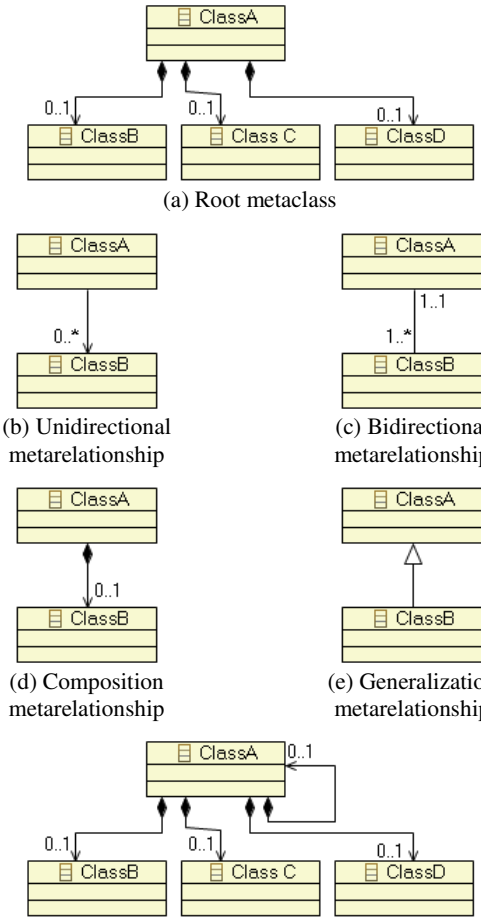

Figure 1: Characteristics influencing model traversal

To be able to take advantage of the identified characteristics, only metamodels that satisfy certain constraints should be used. These constraints are:

- The model root must not be connected to itself with any kind of metarelationship, except a composition metarelationship.
- Bidirectional metarelationships are not used (they are usually not used in metamodeling with ECore). Instead, two unidirectional metarelationships that are navigable in opposite ways are used, which makes the generation less complex. An alternative would be to convert bidirectional metarelationships to unidirectional metarelationships automatically before executing the algorithm.

3.3.2 An algorithm for model traversal generation

Based on the identified characteristics, it is possible to define an algorithm that automatically constructs a model traversal for any metamodel that satisfies the identified constraints.

The generated traversal satisfies all the identified requirements mentioned in Section 3.3.1, although it may not be optimal. The outline of the algorithm for constructing such a traversal is given in Listing 1.

```
modelRoot = findAbsoluteModelRoot();
visit(modelRoot);

// recursive visiting function
function visit(currentModelElement)

// evaluate the current node
evaluateInheritedAttributes();

// visit possible generalizations - top down
if currentModelElement has subMetaclass
    subMetaclassOfCurrent
      = getSubMetaclass(currentModelElement);
    visit(subMetaclassOfCurrent);
end-if

// visit possible generalizations - bottom up
if currentModelElement has superMetaclass
    //no computation needed since the inheritance tree
    //is already traversed in a top down manner
end-if

// visit possible unidirectional relationships
foreach unidirectionalRel in currentModelElement
    if unidirectionalRel.source == currentModelElement
        evaluateInheritedAttributes();
        foreach modelElement in unidirectionalRel.target
            visit(modelElement);
        end-foreach
        evaluateSynthesizedAttributes();
        markPathAsVisited();
    end-if
end-foreach

// visit possible composition relationships
foreach compositionRel in currentModelElement
    if compositionRel.source == currentModelElement
        evaluateInheritedAttributes();
        foreach modelElement in compositionRel.target
            visit(modelElement);
        end-foreach
        evaluateSynthesizedAttributes();
        markPathAsVisited();
    end-if
end-foreach
evaluateSynthesizedAttributes();

end-function
```

Listing 1: Outline of our model traversal generation algorithm

4. EXAMPLE TRAVERSAL GENERATION

To demonstrate the proposed semantic definition and automatic generation of a model interpreter, the widely known and well-understood example of a finite state machine (FSM) modeling language is used. A metamodel of the FSM, which is annotated with the semantic information, can be found in Figure 2.

From the metamodel (Figure 2), it can be observed that the structure is quite simple and that a state machine (abstracted by the 'StateMachine' metaclass) consists of two main model elements that have to be traversed, namely states (represented by the 'AbstractState' metaclass) and transitions (represented by the 'Transition' metaclass). In the metamodel we can see that each of the metaclasses has its own semantic rule that is used to manipulate the values of semantic attributes and to compute the meaning of the whole model. Before the beginning of the traversal, the attributes 'envs' and 'envt' are initialized. Both attributes store actual states and transitions in the model. While visiting states and transitions, useful information (e.g., name of a state, source and destination of a transition) is stored and finally used in a compute function (not shown in the paper) that outputs the meaning of the model.

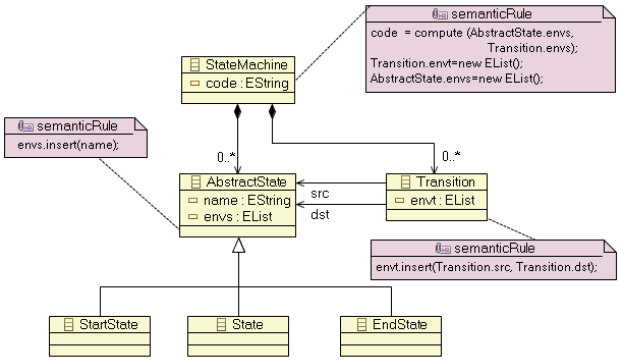

Figure 2: A metamodel of a FSM annotated with the proposed semantic information (semantic attributes and semantic rules)

According to our algorithm and additional computation (evaluation of semantic information), a model interpreter can be automatically generated. An excerpt of the pseudocode for the FSM example can be seen in Listing 2. This example demonstrates an automatic generation of a model interpreter.

```
modelRoot = StateMachine();
modelRoot->TransitionList.envt=new EList();
modelRoot->AbstractState.envs=new EList();

foreach AbstractState in modelRoot->states
    if notVisited()
       AbstractState.envs.insert(AbstractState.text);
       markPathAsVisited();
    end-if
end-foreach

foreach Transition in modelRoot->states
    if notVisited()
       Transition.envt.insert(Transition.src,
                              Transition.dst);
       markPathAsVisited();
    end-if
end-foreach

modelRoot.code = compute(AbstractSate.envs,
                         Transition.envt);
```

Listing 2: An excerpt of an automatically generated interpreter for the FSM example presented in Figure 2

5. CONCLUSION

Model-driven engineering offers a capability to use higher level models to define design intentions, which can then be generated to lower level representations (e.g., source code). However, the implementation of translators to perform the generation is often a manual process. This paper describes our algorithm for generating model traversals from metamodel definitions. Semantic rules and attribute evaluation can be performed during the traversal navigation to define the meaning of a particular model, based on the metamodel semantics. Future work includes the implementation of mature tool support in EMF-based modeling tools.

6. ACKNOWLEDGMENTS

This work was supported in part by an NSF CAREER award (0643725).

7. REFERENCES

[1] D. C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer. vol. 39, no. 2, pp. 25-31.

[2] J. Sprinkle, M. Mernik, J.-P. Tolvanen and D. Spinellis. 2009. Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software. vol. 26, no. 4, pp. 15-18.

[3] D. Harel and B. Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer. vol. 37, no. 10, pp. 64-72.

[4] H. R. Nielson and F. Nielson. 1992. Semantics with Applications. A Formal Introduction. John Wiley & Sons.

[5] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray and H. Wu. 2005. Automatic generation of language-based tools using the LISA system. IEE Proceedings Software. vol. 152, no. 2, pp. 54-69.

[6] M. Mernik, M. Lenic, E. Avdicausevic and V. Zumer. 2002. LISA: An Interactive Environment for Programming Language Development. Proceedings of the 11th International Conference on Compiler Construction (Grenoble, France, 2002). pp. 1-4.

[7] M. Mernik, J. Heering and A. M. Sloane. 2005. When and how to develop domain-specific languages. ACM Computing Surveys. vol. 37, no. 4, pp. 316-344.

[8] K. Chen, J. Sztipanovits, S. Abdelwalhed and E. Jackson. 2005. Semantic anchoring with model transformations. Model Driven Architecture - Foundations and Applications (Nuremberg, Germany, 2005). pp. 115-129.

[9] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bezivin and A. Pierantonio. 2006. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical report. INRIA and LINA.

[10] J. E. Rivera and A. Vallecillo. 2007. Adding behavioral semantics to models. Proceedings of the International Enterprise Distributed Object Computing Conference (EDOC 2007) (Annapolis, Maryland, USA, 2007). pp. 169-180.

[11] J. Gray, J. P. Tolvanen, S. Kelly, A. Gokhale, S. Neema and J. Sprinkle. 2007. Domain-Specific Modeling. Handbook of Dynamic System Modeling. Boca Raton, Florida: CRC Press.

[12] S. Kelly and R. Pohjonen. 2009. Worst Practices for Domain-Specific Modeling. IEEE Software. vol. 26, no. 4, pp. 22-29.

[13] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle and G. Karsai. 2001. Composing Domain-Specific Design Environments. IEEE Computer. vol. 34, no. 11, pp. 44-51.

[14] Y. Jin, R. Esser and J. W. Janneck. 2004. A method for describing the syntax and semantics of UML statecharts. Software and Systems Modeling. vol. 3, no. 2, pp. 150-163.

[15] J. R. Romero, J. E. Rivera, F. Durán and A. Vallecillo. 2007. Formal and tool support for model driven engineering with Maude. Journal of Object Technology. vol. 6, no. 9, pp. 187-207.

[16] D. A. Sadilek and G. Wachsmuth. 2009. Using Grammarware Languages to Define Operational Semantics of Modelled Languages. Objects, Components, Models and Patterns (Zurich, Switzerland, 2009). pp. 348-356.

[17] P. A. Muller, F. Fleurey and J. M. Jézéquel. 2005. Weaving executability into object-oriented meta-languages. Model Driven Engineering Languages and Systems (Montego Bay, Jamaica, 2005). pp. 264-264.

[18] J. de Lara and H. Vangheluwe. 2008. Translating model simulators to analysis models. Fundamental Approaches to Software Engineering (Budapest, Hungary, 2008). pp. 77-92.

[19] J. de Lara, H. Vangheluwe and M. Alfonseca. 2004. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. Software and Systems Modeling. vol. 3, no. 3, pp. 194-209.

[20] D. Knuth. 1968. Semantics of context-free languages. Theory of Computing Systems. vol. 2, no. 2, pp. 127-145.

[21] J. Paakki. 1995. Attribute grammar paradigms - a high-level methodology in language implementation. ACM Computing Surveys. vol. 27, no. 2, pp. 196-255.

[22] H. Wu, J. Gray and M. Mernik. 2008. Grammar-driven generation of domain-specific language debuggers. Software: Practice and Experience. vol. 38, no. 10, pp. 1073-1103.

[23] R. C. Gronback. 2009. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional.

[24] J. White, D. C. Schmidt and S. Mulligan. 2007. The Generic Eclipse Modeling System. Model-Driven Development Tool Implementer's Forum at the 45th International Conference on Objects (Zurich, Switzerland, 2007).