Model Transformations Require Formal Semantics

Yu Sun, Zekai Demirezen, Tomaz Lukman, Marjan Mernik, Jeff Gray

Model Transformations Require Formal Semantics

[1]Yu Sun, [1]Zekai Demirezen, [2]Tomaž Lukman, [3,1]Marjan Mernik, [1]Jeff Gray

[1] University of Alabama at Birmingham
Dept. of Computer and Information Sciences
{yusun,zekzek,gray}@cis.uab.edu

[2] Jožef Stefan Institute
Dept. of Systems and Control
tomaz.lukman@ijs.si

[3] University of Maribor
Faculty of Elec. Eng. and Computer Science
marjan.mernik@uni-mb.si

Abstract

Despite the increasing interest in model-driven engineering, there are many open issues that need to be addressed to advance the technology and promote its adoption. This position paper outlines several current limitations of model transformation, with a specific emphasis on model optimization. A primary shortcoming that can be found in many model transformation approaches and tools is the lack of formal semantics to define the meaning of a modeling abstraction. This inadequacy is the source of many problems surrounding the practice of model engineering.

1. Introduction

Program transformation [2] is a widely used technique to support software evolution and maintenance, which transforms an input source to a modified target. Within the context of Model-Driven Engineering (MDE) [1], model transformation [3] plays a similar role of importance. Instead of transforming source code, a model transformation can be applied to evolve models that represent a higher level of abstraction. Although program and model transformation process different software artifacts, they share several similarities. This paper identifies several problems of model transformation and suggests that the more established area of program transformation may provide a solution context.

A special type of program transformation is called rephrasing [2], which transforms a program into a different program in the same language (i.e., the source and target programs are written in the same programming language). Similarly, in model transformation, it is also common to transform a model to another model in the same domain (i.e., both the source and target models conform to the same metamodel), which is called an endogenous transformation [3]. Such transformations represent refinements that are realized at the same level of abstraction. An example of rephrasing, which is especially interesting for our discussion, is the modification of a particular source code (or model) to support some desired optimization.

One essential requirement of optimization is to ensure that the semantics of the program (or model) is preserved in the whole process of optimization. That is, the target program (or model) must have the same meaning as the source input. One approach to achieve this level of correctness is to make a formal proof based on the semantics of the desired optimization. We conducted a small experiment to implement optimization for a simple domain-specific language (DSL) [4] and a domain-specific modeling language (DSML) [20] in order to make analogies between program optimization and model optimization. We discovered that the more mature foundation of programming language theory could be used to define the semantics of a DSL such that a formal optimization proof is realizable. However, for DSMLs, it was more challenging (and in some modeling tools not even possible) to prove optimization at the modeling level due to a lack of formal semantics for the modeling language.

The paper is organized as follows: The next two sections introduce current approaches to define semantics for DSLs and DSMLs. A case study on optimizing a simple DSL and DSML is given in Section 4 to show that DSMLs often lack a formal approach to define semantics. In Section 5, some other problems raised by the lack of formal semantics in DSMLs are proposed for discussion, pointing toward future work. Finally, a conclusion offers a summary of our position.

2. Methods for Representing Semantics

There are several different methods that have been proposed to describe the semantics of programming languages in a concise, formal and complete manner [9]. They also have been successfully applied to define the semantics for DSLs. The following lists three main approaches to specify programming language semantics that may also be useful for defining the semantics of DSMLs.

Denotational Semantics

Denotational semantics [8] is a semantic definition technique, which is based on mathematical constructs. In denotational semantics, each language element is associated to a mathematical object by mapping functions. Concise and rigorous definition of objects and functions provides an excellent way to represent the meaning of constructs. Although the denotational semantics of DSMLs could be defined in terms of state changes, manipulating mathematical objects rather than DSML constructs lead to difficulties and complexity during implementation.

Operational Semantics

Operational semantics [10] can be used to specify the meaning of a programming language in terms of program execution on abstract machines. Semantic definitions are composed of rules, which describe specific effects of language constructs on an abstract machine. Each rule consists of preconditions that have to be met for the rule to apply and affects the transformation of the current state in some way. Applying the rules of operational semantics yields new states that gradually extend and finally replace the syntactic structures by auxiliary constructs and values. The final states of this transition system only contain values; they represent the result of the specification. Most of the DSML platforms that initiated a formal way of specifying the semantics have employed an operational style of the semantics definition (for more information, please see Section 4).

Attribute Grammars

An attribute grammar [11] is a formal technique used to specify static semantics as an extension of a context-free grammar. This technique can be used to specify dynamic semantics as an interpretation of constructs by defining a translation into lower level code. Attribute grammars are mostly utilized to check the correctness of the static semantics like variable type checking, compatibility between procedure definition and call. Attribute grammars form one of the essential parts of compilers and bring benefits such as automatic construction of compilers, interpreters and other language-based tools. However, the large number of rules required for a complete definition of a language may offer challenges when using attribute grammars to define a DSML.

3. DSML Platforms and Semantics

Compared with the rich and mature techniques and tools to define the semantics for programming languages, semantic specification and supporting tools in the context of DSMLs is still an open area. There have been a few initial approaches for semantic specification within DSML platforms, which generally provide a mechanism for specifying a mapping between the abstract syntax and the semantic domain. There are initial studies that extend denotational semantics (i.e., the denotational metamodeling [19] approach) or attribute grammars. In this section, approaches that are based on operational semantics are discussed.

Atom3 [12] is a visual metamodeling tool that uses graph grammars to represent models. In addition to syntax definition, the tool includes code generation, model optimization and simulator specification facilities. In Atom3, operational semantics of models are defined as graph transformations based on graph rewriting. Graph transformations are composed of rules that map a source graph to a target graph. Although the tool enables definition of semantics in an abstract manner, relying on graph grammars typically introduces performance issues, which inherit from the time complexity of the graph matching algorithm [1].

Kermeta [13] is a model-driven toolsuite, which has been designed to provide tools to build models and actions in the same meta layer. To provide this functionality, the tool composes action metamodels with existing metalanguages (e.g., EMOF) and enables imperative control structures and iterators. This built-in support for specification of operational semantics enables the simulation and testing of metamodels. However, the necessity of defining the behavior of each concept in an imperative way results in code, which is written in the style of a general-purpose programming language.

AMMA [17] **and GME** [18], which are two of the most known and mature non-commercial metamodeling tools, do not support the formal specification of DSML semantics natively. However, attempts were made for each tool to define the operational semantics of DSMLs via Abstract State Machines (ASM) [15]. These attempts were very similar and specified the semantics as operational rules through a sequence of state transitions on the ASM. The main difference was that the attempt in GME used an external tool (Microsoft's ASML) to define the semantics [7]. The ASML tool was connected via transformations in the GREAT [16] transformation language. The effort to add semantics to AMMA integrated the semantics definition directly within the platform, rather than an external tool [14].

4. Example Program and Model Optimization

We implemented a simple Robot language [5] that can be used to control the movement of a virtual robot. The grammar and its denotational semantics for the Robot language are shown in Figure 1. In the semantics part, functions C and P map to a mathematical object, which maps the Robot position to a new position based on a command sequence. In order to demonstrate the problem, we use both a DSL and DSML to define this language and show the optimization problem by comparison. In Figure 2 (left column) a simple program in the Robot language is shown.

Suppose that we desire to optimize the program in Figure 2 to increase the efficiency of its execution. In Optimization 1, the sequence of moves can be rearranged so that the same type of moves are adjacent. In Optimization 2, some combinations of moves have no effect and can be eliminated (e.g., *down up*). The rationale behind Optimization 1 is that the robot can move faster if there is no need to change the direction. Intuitively, both are correct and reasonable optimizations. However, we believe that all transformations, in particular optimizations, should be based on proofs that utilize formal language semantics. In the case of the Robot DSL, the proof was not hard because a formal semantics definition is available [5]. The proof is given in Figure 3. To implement the optimization at the DSL level, several tools are available, such as program transformation systems [2] and language construction environments [4]. Most of these tools transform the program by syntax pattern matching. For example, in order to realize Optimization 1, a transformation rule could transform the program pieces that match the pattern *down up down* into *down down up*. By executing the rule until no more changes can be made, the same type of move can be constructed. For Optimization 2, a transformation rule can match *down up* and eliminate the contradicting moves.

P ::= begin C end

C ::= left | right | up | down | C1C2

P : Program → Int*Int

P ⟦begin C end⟧ = **C** ⟦C⟧(0,0)

C :: Command → Int*Int → Int*Int

C ⟦left⟧(x,y) = (x+Δx,y+Δy) where Δx=-1 and Δy=0

C ⟦right⟧(x,y) = (x+Δx,y+Δy) where Δx=+1 and Δy=0

C ⟦down⟧(x,y) = (x+Δx,y+Δy) where Δx=0 and Δy=-1

C ⟦up⟧(x,y) = (x+Δx,y+Δy) where Δx=0 and Δy=+1

C ⟦C1 C2⟧(x,y) = let (x+Δx1, y+Δy1) = **C** ⟦C1⟧(x,y) in

let (x+Δx1+Δx2, y+Δy1+Δy2) = **C** ⟦C2⟧(x+Δx1, y+Δy1) in

(x+Δx1+Δx2, y+Δy1+Δy2)

Figure 1. Context-free Grammar and Denotational Semantics for the Robot Language

Original Program	**Program Optimization 1**	**Program Optimization 2**
begin down up down left end	begin down down up left end	begin down left end

Figure 2. Program Instances written in Robot Language

To prove "begin C1 C2 end" = "begin C2 C1 end"

We have to show that:

P ⟦begin C1 C2 end⟧= **P** ⟦begin C2 C1 end⟧

In other words, we have to prove:

C ⟦C1 C2⟧(0,0) = **C** ⟦C2 C1⟧(0,0)

Since:

C ⟦C1 C2⟧(0,0) = let (Δx1, Δy1) = **C** ⟦C1⟧(0,0) in
let (Δx1+Δx2, Δy1+Δy2) = **C** ⟦C2⟧(Δx1, Δy1) in
(Δx1+Δx2, Δy1+Δy2)

C ⟦C2 C1⟧(0,0) = let (Δx2, Δy2) = **C** ⟦C2⟧(0,0) in
let (Δx2+Δx1, Δy2+Δy1) = **C** ⟦C1⟧(Δx2, Δy2) in
(Δx2+Δx1, Δy2+Δy1)

Also:

(Δx1+Δx2, Δy1+Δy2) = (Δx2+Δx1, Δy2+Δy1)
(due to associativity of +)

We can get:

C ⟦C1 C2⟧(0,0) = **C** ⟦C2 C1⟧(0,0)

Therefore:

P ⟦begin C1 C2 end⟧= **P** ⟦begin C2 C1 end⟧

Figure 3. Proof for Optimization 1

To achieve the same formal analysis capabilities at the DSML level is still a major challenge, due to the lack of formal semantic specification for DSMLs. The metamodel for the Robot language is simple: a `robot` with two attributes to record the coordinates can contain one or more `movement` commands. The `direction` attribute in the movement can be one of the four - `left, right, up, and down`. Figure 4 is the metamodel for the Robot language.

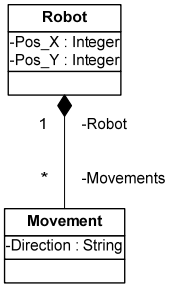

Figure 4. Metamodel for Robot Language

In many modeling tools, the semantics of the language is defined solely by a model interpreter that translates the model representation into some other representation. The model interpreter traverses each element of the model and generates corresponding code. In this experiment, we implemented the Robot language in GEMS [6] and the model interpreter was written in Java. Figure 5 is an excerpt of the interpreter code. As can be seen, the semantics of each modeling element is defined by Java code. In this case, if the element is an action *`left`*, executing the code in the *`visitLeft`* method will change the corresponding coordinates and display them. Thus, to understand the meaning of a modeling language requires that the Java code representing the interpreter should be comprehended fully. This represents a poor solution to semantics and does not provide a representation that can be easily understood and processed (i.e., the concerns of the programming language hide the intention of the domain). This manner of representing semantics also does not allow semantic reasoning based on proofs. Several existing modeling tools force this style of code generation on the user with the resulting undesirable situation with respect to clear semantics of the DSML. Although some initial promising attempts focus on directly using or mapping the model concepts to existing models of computation (e.g., action semantics, denotational semantics, Abstract State Machines (ASM) [7]), they are not at the same level as the domain represented by the DSML. Because these alternative models of computation are at different levels of abstraction and represented in different technical spaces, there exist many challenges in providing a semantic mapping.

```
public void visitLeft(Left tovisit) {

  int temp_x = Integer.parseInt((String)(tovisit.
           getParent().getAttribute("Pos_x")));
  int temp_y = Integer.parseInt((String)(tovisit.
           getParent().getAttribute("Pos_y")));

  MakeAction((Robot)(tovisit.getParent()),
                     tovisit, temp_x, temp_y);
  displayRobotPosition(tovisit);
  visitContainer(tovisit);

}
```

Figure 5. An Excerpt of the Robot Interpreter

Due to these limitations, it is challenging to base model optimizations on DSML semantics. Without the ability to prove properties of an optimization, the resulting model instance may represent an incorrect transformation from the source model. The situation is somewhat similar to the state of programming language research during the early 1960s, before formal semantic approaches were invented, when researchers used concrete operational semantics. During those times, the meaning of a language construct was described by translation to machine code or by the interpreter's code. Different formal semantic methods were developed with the aim to prove the properties of language constructs and to automatically generate compilers or interpreters, as well as other language tools (e.g., editors, debuggers, test engines).

5. Other Problems Related with Semantics

Without a formal and uniform specification for DSML semantics, several challenges are essential to the success of MDE. The following discussion represents our position for the workshop, which claims that program transformation systems and traditional language engineering tools may offer insight into a solution to some of these challenges.

How to improve readability of DSMLs

The semantics embedded in the model interpreter in the form of code is hard to comprehend. In addition, different modeling tools have different mechanisms to traverse the model instance, which makes the understanding process more difficult. For instance, a GEMS model interpreter works in a depth-first search pattern, but GME traverses the model in the order of types of the model elements.

How to automatically generate model interpreters

Without a formal and unique representation for semantics, a model compiler or interpreter cannot be generated automatically. For DSLs, a number of tools exist to support the generation of compilers according to the syntax and semantics specifications, such as Lisa [23], ASF+SDF [24], ANTLR [25]. A Lisa specification for the Robot language is shown in Figure 6. This robot language specification can be processed by Lisa to produce several generated files that are translated to a general-purpose language to provide executabiltiy. However, no such mechanism is generally available for DSMLs. A further consequence is that various other language-based tools such as debuggers, test engines also cannot be generated automatically.

```
lexicon {
       keywords  begin | end
       operation left | right | up | down
       ignore [\0x0D\0x0A\ ]
   }
   attributes int *.inx; int *.iny;
              int *.outx; int *.outy;
   rule start {
    START ::= begin COMMANDS end compute {
        START.outx = COMMANDS.outx;
        START.outy = COMMANDS.outy;
        COMMANDS.inx = 0;
        COMMANDS.iny = 0;
    };
```

Figure 6. An Excerpt of the Robot Specification in Lisa

How to verify model compiler correctness

Model compilers and interpreters are implemented mainly by general-purpose programming languages. Hence, verifying a model transformation is very difficult, if not impossible.

How to prove properties of domain concepts

Proving properties about concepts and relationships in the domain is not possible due to the lack of formal semantics of DSMLs. For instance, in the Robot case, the concept that two movements can be switched (proved in Figure 4) is an important property for this domain. However, an equivalent proof on the DSML level is difficult if verification, optimization, and parallelization of models are typically expressed through general-purpose programming languages.

How to make model transformation languages connected with a semantic definition

Currently, many model transformation languages (e.g. ATL [21], C-SAW [22]) exist, and have shown initial success in different aspects of model transformation. Although many of the languages are declarative and at a high level of abstraction, they do not support formal specification of the semantics or validation of the transformation. The rules are based on a developer's subjective decisions, which are not reliable.

6. Conclusion

Due to the lack of formal semantics for DSMLs, the real meaning of a modeling language is available only in associated model interpreters. As a consequence, model transformations cannot be verified for preserving the semantics, which is a serious shortcoming compared to the capabilities offered by textual DSLs that are defined through grammars and language definition tools. At the workshop, we will outline these challenges and indicate how lessons from the areas of program transformation and language definition might address some of the challenges of representing the semantics of modeling languages.

Acknowledgements

This work supported by NSF CAREER award CCF-0643725.

References

1. Schmidt, D. "Model-Driven Engineering," IEEE Computer, vol. 39 no. 2, pp. 25-32 (2006).
2. Visser, E. "A Survey of Rewriting Strategies in Program Transformation Systems," Workshop on Reduction Strategies in Rewriting and Programming, Electronic Notes in Theoretical Computer Science, vol. 57, Utrecht, The Netherlands (2001).
3. Mens, T., Gorp, P. "A Taxonomy of Model Transformation," Proceedings of the International Workshop on Graph and Model Transformation, vol. 152, pp. 125-142 (2005).
4. Mernik, M., Heering, J., Sloane, T. "When and How to Develop Domain-Specific Languages," ACM Computing Surveys, vol. 37, issue 4, pp. 316-344 (2005).
5. Wu, X., Mernik, M., Bryant, B., Gray, J. "Implementation of Programming Languages Syntax and Semantics," Encyclopedia of Information Science and Technology, 2nd Edition, IGI Global (2008).
6. GEMS Project http://www.eclipse.org/gmt/gems
7. Chen, K., Sztipanovits, J., Neema, S., Emerson, M. Abdelwahed, S. "Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages," Proceedings of the Fifth ACM International Conference on Embedded Software, pp. 35-43 (2005).
8. Stoy, J. "Denotational semantics: The Scott-Strachey Approach to Programming Language Theory," MIT Press, Cambridge, MA (1977).
9. Nielson, H., Nielson, F. Semantics with Applications: A Formal Introduction, Wiley Professional Computing, Wiley, Revised Edition (1999).
10. Plotkin, G. "The Origins of Structural Operational Semantics," Journal of Logic and Algebraic Programming, vol. 60-61, Pp. 3-15 (2004).
11. Knuth, E. "Semantics of Context Free Languages," Mathematical Systems Theory, vol. 2, no. 2, pp. 127-145 (1968).
12. Syriani E., Vangheluwe, H. "Programmed Graph Rewriting with Time for Simulation-Based Design," First International Conference on Model Transformation, pp. 91-106 (2008).
13. Muller, P., Fleurey, F., Jezequel, J. "Weaving Executability into Object-Oriented Meta-languages," In Proceedings of MoDELS, pp. 264-278 (2005).
14. Di Ruscio, D., Jouault, F., Kurtev, I., Bezivin, J., Piearantonio, A. "Extending AMMA for supporting dynamic semantics specifications of DSLs," LINA Research Report number 06.02, University of Nantes (2006).
15. Borger, E. "High-Level System Design and Analysis using Abstract State Machines," In Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods, pp. 1-43 (1998) .
16. Agrawal, A. "Graph Rewriting and Transformation (GReAT): A Solution for The Model Integrated Computing (MIC) Bottleneck," International Conference on Automated Software Engineering, pp. 364-368 (2003).
17. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P. "Model-based DSL Frameworks," In Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 602-616 (2006).
18. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T. "Model-integrated Development of Embedded Software," Proceedings of the IEEE, vol. 91(1), pp. 145-164 (2003).
19. Hausmann, J. "Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages," Doctoral thesis, University of Paderborn, Paderborn, Germany (2005).
20. Jackson, E. "The Software Engineering of Domain-Specific Modeling Languages: A Survey Through Examples," Technical Report, Institute For Software Integrated Systems (ISIS), ISIS-07-807, March (2008).
21. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I. "ATL: A Model Transformatin Tool," Science of Computer Programming, vol. 72, nos. 1/2, June 2008, pp. 31-39 (2008).
22. Gray, J., Lin, Y., and Zhang, J. "Automating Change Evolution in Model-Driven Engineering," IEEE Computer, Special Issue on Model-Driven Engineering (Doug Schmidt, ed.), vol. 39, no. 2, February 2006, pp. 51-58 (2006).
23. Mernik, M., Korbar, N., Zumer, V. "LISA: A Tool for Automatic Language Implementation," ACM SIGPLAN Notices, vol. 30(4) pp. 71–79 (1995).
24. Brand, M., Heering, J., Klint, P., Olivier, P. "Compiling Language Definitions: The ASF+SDF Compiler," ACM Transactions on Programming Languages and Systems, vol. 24, no. 4, pp. 334-368 (2002).
25. Parr, T. The Definitive ANTLR Reference: Building Domain-Specific Languages, The Pragmatic Programmer (2007).
26. Varro, G., Schurr, A., Varro, D. "Benchmarking for Graph Transformation," IEEE Symposium on Visual Languages and Human-Centered Computing, pp. 79-88 (2005).