Toward Denotational Semantics of Domain-Specific Modeling Languages for Automated Code Generation

Danielle Gaither, Barrett R. Bryant

Dept. of Computer Science and Engineering
University of North Texas
Denton, TX USA

{dcg0063, barrett.bryant}@unt.edu

Abstract. One goal of model-driven development (MDD) is automated code generation, which is ultimately a type of model transformation. Current approaches to such transformations are often rule-based, implying a focus on operational semantics. We explore an approach based on denotational semantics in this paper. First, we construct a denotational semantics for elements of a modeling language based on the metamodel of that language. These denotational semantics are then implemented in an established programming language. Once the denotational semantics of the modeling language are created, we discuss ways to integrate our approach to automated code generation into existing domain-specific modeling (DSM) tools.

Keywords: domain-specific modeling, denotational semantics, model-driven development

1 Introduction

An important aspect of automated code generation is establishing a semantics for the source language. The main types of semantics associated with programming languages are axiomatic, operational, and denotational semantics. Axiomatic semantics primarily concern themselves with logical transformation of predicates. Operational semantics interpret a program as an algorithm. Denotational semantics map components of a program to mathematical functions [1].

Most approaches to model-based code generation to date have involved rule-based model transformations, which could be considered equivalent to an operational semantics of the model. This approach has two significant disadvantages: (1) the rules tend to be specified in an ad hoc manner, making formal analysis difficult, if not impossible, and (2) the approach does not lend itself to composition of models from different domains.

Denotational semantics can overcome these disadvantages because of its mathematical foundations. Mathematical functions certainly lend themselves to formal

analysis as well as to composition. Denotational semantics have an additional advantage of being independent of their implementations [2]. Khaligh and Radetzki [3] observe that denotational semantics can be difficult to implement due to their lack of obvious mapping to typical implementation strategies.

We have chosen to base our work in this paper on a domain-specific language (DSL) created by Marques et al. [4] for developing role-playing games (RPGs) for mobile phones. The language allows for a variety of semantic concepts to be portrayed and for certain concepts to be verified. The latter is particularly important, as one major advantage of MDD is its amenability to formal verification.

Since this paper is not about the RPG DSL *per se*, but rather about ways of analyzing and transforming it, this paper will focus on a simplified subset of the language, which we will call MicroRPG. Each game contains: exactly one map of a specified height and width; exactly one hero with a name, a position, a given number of health points, and a status; exactly one goal; and zero or more traps that decrease the hero's health points. The game ends when the hero either wins by reaching the goal or loses by running out of health points. The metamodel for MicroRPG is shown in Figure 1.

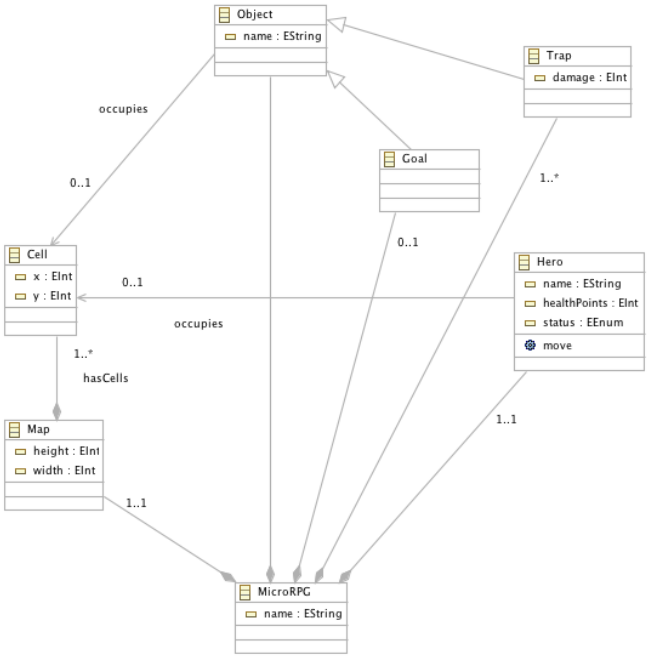

Figure 1, MicroRPG metamodel.

2 Denotational Semantics of Modeling Languages

2.1 Background

The approach to model-based code generation has some overlap with the field of compiler construction, especially concerning syntax analysis and semantic analysis. In

the case of modeling languages, syntax analysis is performed by checking whether a given model instance conforms to the model's associated metamodel, which functions similarly to a grammar for a traditional programming language. For example, a MicroRPG model with two heroes would not be syntactically correct, since the MicroRPG metamodel specifies one Hero per MicroRPG. Modeling tools typically have built-in functions to check the correctness of a given model's syntax.

Syntax analysis is followed by semantic analysis. Syntax specifications for traditional programming languages have been sufficiently standardized that a grammar specification can be dropped into a parser generator. While automated tool construction has enjoyed some success in traditional programming languages, especially with the use of attribute grammars [5] [6], the semantics of modeling languages generally lack such formalization [1]. We will focus on graphical modeling languages, since they present the most significant differences from traditional programming languages.

2.2 Denotational Semantics of MicroRPG

The syntax rules for MicroRPG are established by the metamodel as shown in Figure 1. The syntax domains for MicroRPG are: MicroRPG, GameMap, Cell, Hero, Object, Trap, and Goal, which correspond with the classes in Figure 1.

The semantics we provide will preserve the items in the metamodel, as well as their relationships to each other. Semantic domains include the syntax domains, as well as functions that return the attributes specified in Figure 1.

Each MicroRPG has exactly one Hero. Each Hero begins the game in a specified location, a specified health value, and a status of "Won," "Lost," or "InProgress."

Each MicroRPG contains exactly one GameMap of a specified width and height (both integers). Each Cell in a Map is identified by an (*x*, *y*) value between (1, 1) and (*width*, *height*).

Each Cell contains at most one Object, which can be either a Goal or a Trap. Each Map must have exactly one Goal and zero or more Traps. Each Trap has a location and a damage value.

If the Hero occupies the same space as a Trap, the Trap's damage value is subtracted from the Hero's health. If the Trap's damage exceeds the Hero's health, the Hero loses the game.

The Hero can move up, down, forward, or backward one space at a time. If the Hero tries to move in an inappropriate direction, the Hero will stay in the currently occupied space. If the Hero occupies the same space as the Goal, the Hero wins the game.

Our first example function will involve moving the Hero. The map is *n* cells wide by *m* cells high. The Hero may move one space horizontally or vertically at a time. For this example, the Hero's location will be (1, 1), and the hero will move up. Assume that the target space (in this case, location (1, 2)) does not contain the goal or a trap. Since not all moves are legal, e.g., trying to move down from (1,1), it is necessary to check the validity of a potential move. In a modeling context this can be specified using constraints, as is currently done in Object Constraint Language or Alloy, for example. In this instance, the *move* function is checked by a function called *tryMove*.

$$move: GameMap \times Cell \times Direction \rightarrow Cell$$
$$move\ [\![gameMap\ (1,1)\ Up]\!]$$
$$= tryMove\ [\![(x1, y1)(x2, y2)]\!]$$
$$= tryMove\ [\![(1,1)\ (2,1)]\!]$$
$= \text{if } x2 < 1 \text{ or } x2 > width(gameMap) \text{ or}$
$y2 < 1 \text{ or } y2 > height(gameMap) \text{ then } (x1, y1)$
else (x2, y2)
$= (x2, y2)$
$= (2, 1)$

A more complex example function will involve the Hero encountering a Trap. This occurs when the Hero and the Trap occupy the same Cell. We will assume that this Trap causes one point of damage. If the hero has only one health point remaining, the hero loses the game. Otherwise, the hero simply loses one health point. For this example, we will assume that the hero only has one health point remaining.

$$objectEncounter: Object \times Hero \rightarrow Hero$$
$$objectEncounter[\![trap\ hero]\!]$$
$= \text{if } damage(trap) \geq health(Hero) \text{ then}$
$Hero\{name(hero), position(hero), 0, \text{Lost}\} \text{ else}$
$Hero\left\{\begin{matrix} name(hero), position(hero), \\ health(hero) - damage(trap), \text{InProgress} \end{matrix}\right\}$
$= Hero\{name(hero), position(hero), 0, \text{Lost}\}$

The game is over, and the hero is unsuccessful.

3 Haskell Implementation

Since Haskell is amenable to formal verification and a popular choice for writing language implementations, it has been chosen to implement the semantics specified in this paper. While space limitations prohibit a complete listing of the implementation, some sample functions will be displayed here. The syntax domains translate easily to data or type declarations in Haskell. For example, the declaration for the Hero is:

```
data Hero = Hero {
    name :: String, position :: Cell, health :: Int,
         status :: Status}
    deriving (Show,Read,Eq)
```

The following method shows what happens when the Hero encounters an Object.

```
objectEncounter :: Object -> Hero -> Hero
objectEncounter (Goal{location=l})
    (Hero {name=n,position=p,health=h,status=s})
         = (Hero n p h Won)
objectEncounter (Trap {location=l,damage=d})
    (Hero {name=n,position=p,health=h,status=s})
         | d >= h = Hero n p 0 Lost
         | otherwise = Hero n p (h-d) InProgress
```

Note how the implementation components correspond to similar components of the metamodel in Figure 1 and to the semantics discussed in Section 2 of this paper.

4 Integration With Existing Tools

Marques et al. [4] used the Eclipse Modeling Framework (EMF) for their work, which was also used for the work in this paper. When creating an executable denotational semantics, functional languages are popular choices for this task [6] [7] [8]. This is not surprising, as the expression of denotational semantics can certainly be thought of as a DSL, and indeed, as a model [9], and functional languages have been used to create many DSLs. Eclipse currently has plugins for Scala [10], Haskell [11], Scheme [12], Clojure [13], and OCaml [14] development.

5 Related Work

While there are many publications dealing with the semantics of metamodels, relatively few of those publications concern themselves with denotational semantics. Mannadiar and Vangheluwe [15] have discussed using denotational semantics for code generation but note the same difficulties mentioned previously in this paper.

Many publications discuss the use of denotational semantics in the design of DSLs, but not for modeling or metamodeling as such. Vytiniotis et al. [16] treat the denotational semantics themselves as a model to be implemented in Haskell, writing statically-verifiable contracts based on denotational semantics. Peyton Jones, Eber, and Seward [17] used denotational semantics to determine the values of various financial contracts.

Some work has discussed using denotational semantics with the Unified Modeling Language (UML). Kim and Carrington [18] use denotational semantics to derive operational semantics. However, Mannadiar and Vangheluwe [15] raise objections to the use of UML for automated code generation, as they claim UML is too broad to generate well-designed code in many instances.

6 Conclusions and Future Work

We have presented a way to map metamodel components to denotational semantics, which easily lend themselves to implementation in a functional language. We have taken an example of a domain-specific modeling framework, created a denotational semantics for that framework, and shown part of an implementation that maps closely to the metamodel. Given that denotational semantics of programming languages are amenable to both composition and formal analysis techniques, we expect these benefits to carry over to modeling languages as well. An important future step will be implementing the mapping between model components and code in an automated fashion.

7 References

[1] L. Allison, A practical introduction to denotational semantics, Cambridge: Cambridge University Press, 1986.

[2] D. A. Schmidt, Denotational semantics: a methodology for language development, London: Allyn & Bacon, 1986.

[3] R. S. Khaligh and M. Radetzki, "A metamodel and semantics for transaction level modeling," in Forum on Specification and Design Languages, Oldenburg, Germany, 2011.

[4] E. Marques, V. Balegas, B. Barrocca, A. Barisic and V. Amaral, "The RPG DSL: a case study of language engineering using MDD for genearting RPG games for mobile phones," in Workshop on domain-specific modeling, Tucson, AZ, 2012.

[5] J. Heering and P. Klint, "Semantics of programming languages: a tool-oriented approach," ACM SIGPLAN Notices, vol. 35, no. 3, pp. 39-48, 2000.

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

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

[8] B. Denckla, P. J. Mosterman and H. Vangheluwe, "Towards an executable denotational semantics for causal block diagrams," in OOPSLA Workshop in Domain-Specific Modeling, San Diego, 2005.

[9] P. Hudak, "Modular domain specific languages and tools," in Software Reuse, Victoria, Canada, 1998.

[10] "Scala IDE for Eclipse," [Online]. Available: http://scala-ide.org/. [Accessed 14 February 2013].

[11] "EclipseFP: The Haskell plug-in for Eclipse," [Online]. Available: http://eclipsefp.github.com/. [Accessed 14 February 2013].

[12] "SchemeWay - Scheme plugins for Eclipse," [Online]. Available: http://sourceforge.net/projects/schemeway/. [Accessed 14 February 2013].

[13] L. Petit, "Counterclockwise," [Online]. Available: http://code.google.com/p/counterclockwise/. [Accessed 14 February 2013].

[14] N. Bros and R. Cerioli, "OcaIDE," 2007. [Online]. Available: http://www.algo-prog.info/ocaide/. [Accessed 14 February 2013].

[15] R. Mannadiar and H. Vangheluwe, "Domain-specific engineering of domain-specific languages," in Workshop on domain-specific modeling, Reno, NV, 2010.

[16] D. Vytiniotis, S. Peyton Jones, D. Ros´en and K. Claessen, "HALO: Haskell to logic through denotational semantics," in Principles of Programming Languages, Rome, 2013.

[17] S. Peyton Jones, J.-M. Eber and J. Seward, "Composing contracts: an adventure in financial engineering - Functional pearl," in International Conference on Functional Programming, Montreal, 2000.

[18] S.-K. Kim and D. Carrington, "A Formal Model of the UML Metamodel: The UML State Machine and Its Integrity Constraints," in Intl. Conf. B and Z Users, Grenoble, France, 2002.