Verification of DSMLs Using Graph Transformation: A Case Study with Alloy

Zekai Demirezen[1], Marjan Mernik[1, 2], Jeff Gray[1], Barrett Bryant[1]

[1]Department of Computer and Information Sciences
University of Alabama at Birmingham, Birmingham, AL 35294-1170
{zekzek, gray, bryant}@cis.uab.edu

[2]Faculty of Electrical Engineering and Computer Science
University of Maribor, 2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

Abstract. Domain-Specific Modeling Languages (DSMLs) enable domain experts to participate in software development tasks and to specify their own programs using domain abstractions. Many Model-Driven Engineering (MDE) platforms primarily concentrate on structural aspects of DSMLs and only provide techniques to define abstract and concrete syntax. Only a few platforms provide built-in support for specification of behavioral semantics and verification tasks. In this paper, we focus on how to specify the behavioral semantics of a DSML by a sequence of graph transformation rules. We also discuss how to transform a DSML specification into Alloy, a model checking tool. These transformations demonstrate that DSML models specified in a visual notation can be verified by means of existing model checking tools.

Keywords: model checking, verification, domain-specific modeling languages, operational semantics, graph transformation systems, activity diagram.

1 Introduction

Model-Driven Engineering (MDE) has been shown to increase productivity and reduce development costs [1]. The concepts advocated by MDE focus on abstractions tied to a specific domain that provide tailored modeling languages for domain experts. Domain-Specific Modeling Languages (DSMLs) [2], used within the MDE context, enable end-users who are domain experts to participate in software development tasks and to specify their own programs using domain concepts in the problem space, rather than programming language concepts in the technical solution space. However, there remain several challenges that drive new research in DSMLs. For example, simulation, code generation, model checking and different kinds of analysis require a precise definition of the semantics of a DSML that is not provided sufficiently in many modeling toolsets.

Due to the complexity of the software engineering process, formal verification tools are often required to detect design errors, which are difficult to recognize by checking manually. In this context, model checkers [3] provide a technique for finite state systems to detect design errors automatically. In most erroneous situations, these checkers find counter examples to reveal error cases. Although formal verification

tools enable analysis of the correctness of a system, the low-level details of verification tools make it challenging to use by end-users. One solution to overcome this situation is the use of automated transformations that map a designer's high level definitions into the representation used by a model checker.

The purpose of the study described in this paper is to demonstrate how DSML designers can define semantic and verification specifications using visual models. The long-term goal of our project is a language design approach that addressees the automated model checking of DSML instances. The paper discusses how to specify behavioral semantics by a sequence of graph transformation rules and how to map DSML definitions into Alloy. These mappings demonstrate that DSML models designed by a visual notation can be verified by existing model checking tools.

The rest of the paper is organized as follows. Section 2 explains the syntax and semantic specifications of DSMLs using graph grammars. This section also briefly introduces Alloy and discusses the mapping of DSML specifications into Alloy models. Section 3 demonstrates the idea of DSML model checking in Alloy with a case study. Section 4 lists related work in this area. The paper closes with the concluding remarks in Section 5.

2 Specification of DSMLs

DSMLs, like any other language, consist of definitions that specify the abstract syntax, concrete syntax, static semantics and behavioral semantics of a language. Specification of abstract syntax includes the concepts that are represented in the language and the relationships between those concepts. In MDE, domain metamodels are often used to define the structural rules for the abstract syntax. Concrete syntax definition provides a mapping between meta-elements and their textual or graphical representations. Well-formedness rules, which represent the static semantics of a language, can be defined to check model consistency. Such rules are often specified in constraint languages (e.g., OCL) that enforce rules among metamodel elements. The runtime behavior of each syntactical meta-element defined in the DSML represents the behavioral semantics of the language, which is often challenging to specify.

Behavioral semantics of DSMLs can be represented by a sequence of state transition rules. This approach divides all semantic concerns into discrete states and transition relations. All of the defined state changes represent operational semantics of the domain elements. In particular, in-place model transformations [4] represent an approach for designing state transitions. This technique is similar to the Structural Operational Semantics (SOS) defined by Plotkin [5], who proposes SOS to give computational state transitions by means of the abstract syntax of a language. Therefore, SOS defines an abstract behavior for an abstract syntax that allows model checking, correction of proofs and other verification activities.

2.1 Specifying Dynamic Semantics using Graph Grammars

One of the main characteristics of the in-place model transformation is that target and source models are always instances of the same metamodel. An in-place model

transformation rule is defined as L: [NAC]*LHS->RHS, where L is the rule label, LHS denotes the left-hand side rule stating the precondition pattern to trigger the rule; the RHS represents the right-hand side rule that specifies the final model part after execution of a rule. NAC is the optional negative condition that disables the rule if it is satisfied. Graph grammars [6] provide visual rules to specify in-place transformations based on precondition and postcondition steps. The notation proposed by AGG [7] to model graph transformations is used to define these rules visually. AGG is a rule-based visual language supporting an algebraic approach to graph transformation. The AGG visual structure enables DSML designers to define transformation rules in a model-driven manner.

Each AGG transformation specifies the runtime behavior for one of the state transitions. However, to give the complete semantics for a DSML, a sequence of state changes also needs to be defined. These sequence definitions control what state transition is to be fired, in what order, and what condition. An activity diagram is an appropriate state machine to define these transition sequences. It enables the design of simple and compound states, branches, forks, and joins. When the activity of a state completes, a transition enables the flow to pass to the next activity. Although flow may continue as sequential transitions, branches can exhibit alternate paths.

2.2 Verifying Properties of a DSML using Alloy

Alloy [8] is a structural language based on first-order logic, which provides effective techniques for model checking. An Alloy model consists of Signatures, Relations, Facts, Predicates, and Asserts. Signatures represent the concepts of the domain, such as the meta-elements. Signatures consist of relations, which are similar to the meta-attributes. Alloy provides Facts to enable users to define constraints about the Signatures and the Relations. Facts are similar to well-formedness rules of metamodels. Alloy consists of a consistency checker and counter extraction tool. The consistency checker operates on Predicates, which are defined to analyze the model during its evolution. The counter extraction tool utilizes Asserts to check model states to find a counterexample. All of these definitions enable checking the reachability of a given configuration through a finite sequence of steps.

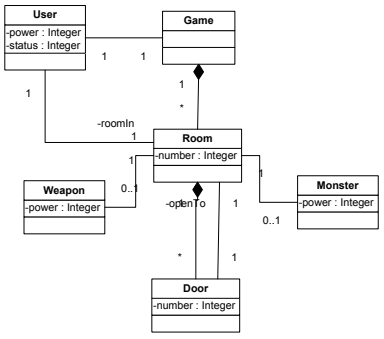
2.3 Mapping a DSML Model to an Alloy Model

Automated model checking of a DSML requires interoperation of existing model checking tools with the syntax and semantics of a DSML. To enable this capability, syntax and semantics specifications, and an instance model specified by these specifications, must be converted into the formalism expected by an underlying model checking tool. Next, the properties that the model must satisfy need to be stated by a logical formalism expressed in the format expected by the verification tool. Graph transformations (described using AGG) can be transformed into an Alloy Model [9]. Metamodel specifications can be mapped into Alloy signatures and predicates. The transformation steps proposed in this paper are summarized by the following items:

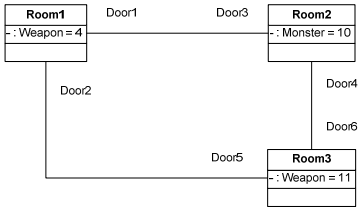
- mapping metamodel elements to Alloy abstract signatures,
- mapping model elements to Alloy concrete signatures,
- mapping graph transformation rules to Alloy predicates, and
- mapping verification tasks to Alloy asserts.

3 Case Study: The Maze Game

The Maze Game example has a simple metamodel shown in Figure 1a. From this metamodel definition, a maze consists of rooms, which can be connected to each other. Each room can contain a weapon and/or a monster with the power attribute. This modeling language is used to generate a game, enabling users to type commands to move in the maze and finish the game without being killed by monsters. A model instance describes a specific maze configuration. Collecting weapons during game-play increases a user's power, which can be used to kill monsters. A model instance is shown in Figure 1b. In the context of this domain, the semantics definition is required for the user movement between rooms. These movements result in state transitions within the model instance. This example contains several situations that demonstrate graph transformation rules and verification of movements using Alloy.



a. Maze Game Metamodel



b. Maze Game Instance with 3 Rooms, 6 Doors, 2 Weapon, and 1 Monster

Fig. 1. Metamodel and Model Instance

The behavioral semantics of a user move is shown in Figure 2. This activity consists of three tasks and two decision points. The `CheckMove` decision point and the `ChangeRoom`, `SubstractMonsterPower` tasks are shown in Figure 2. The LHS definition of `CheckMove` searches for the matching room and door combination that a user wants to move. If this matching is satisfied, it returns true to enable branching to the `ChangeRoom` task. Otherwise, the `IllegalMove` status is set. The `ChangeRoom` task searches for the matching room-door combination and moves the user into the new room. The `SubstractMonsterPower` task locates the room-monster combination and then subtracts the Monster's power from the User's power.

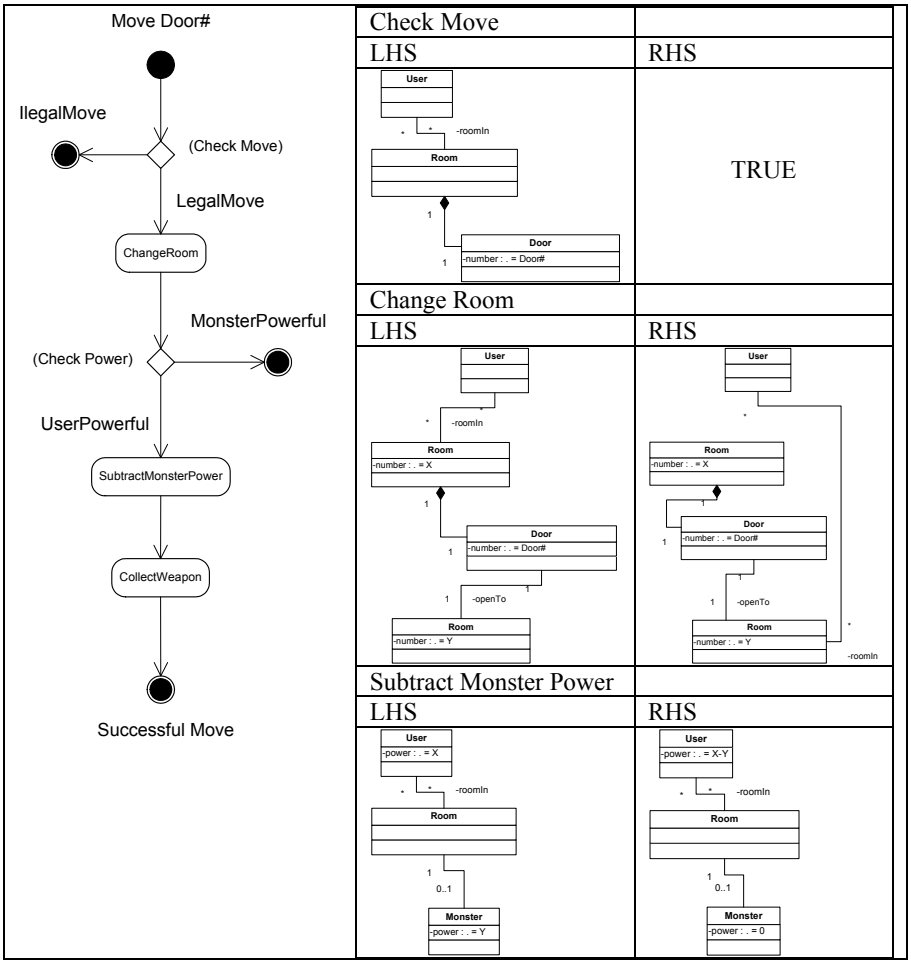


Fig. 2. User Move Semantics

3.1 Mapping Metamodel Elements to Alloy Abstract Signatures

Each meta-element in the Maze Game metamodel is transformed into an Alloy abstract signature definition. A part of the specification is shown in Figure 3. This is an enhanced model of the Maze Game metamodel. For example, the `User` signature includes `roomIn`, as well as other attributes to track user position and status during the execution of a game. This detailed definition represents the runtime metamodel [10].

```
abstract sig Game{
  rooms:set Room,
  user: one User
}
abstract sig Room{
  number:one Int,
  doors:set Door,
  monster:one Monster,
  weapon:one Weapon
}
abstract sig Door{
  number:one Int,
  openTo:one Room
}
```

```
abstract sig User{
  power:one Int,
  roomIn:one Int,
  status:one Int
}

abstract sig Weapon{
  power:one Int
}
abstract sig Monster{
  power:one Int
}
```

Fig. 3. Maze Game Alloy Abstract Signatures

3.2 Mapping Model Elements to Alloy Concrete Signatures

In the previous subsection, each meta-element in the Game metamodel is defined as an abstract signature. Abstract signatures are used to define the meta-layer of the Maze Game. 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. Figure 4 shows model elements of the Maze game, which appeared initially in the diagram in Figure 1b. To create the initial model instance in the Alloy system, the `InitGame` predicate is specified. This predicate establishes all the relations between signature definitions and creates a Maze Game Instance with three Rooms, six Doors, two Weapons, and one Monster.

```
one sig R1,R2,R3 extends Room{}
one sig D1,D2,D3,D4,D5,D6 extends Door{}
one sig M1 extends Monster{}
one sig G1,G2 extends Weapon{}
one sig U1 extends User{}

pred initGame(g:Game){
  D1.number=1 && D2.number=2 && D3.number=3 &&
  D4.number=4 && D5.number=5 && D6.number=6 &&

  D1.openTo=R2 && D2.openTo=R3 && D3.openTo=R1 &&
  D4.openTo=R3 && D5.openTo=R1 && D6.openTo=R2 &&

  g.rooms=R1+R2+R3 && R1.number=1 && R2.number=2 &&
  R3.number=3 && M1.power=10 && G1.power=4 && G2.power=11 &&

  g.user=U1 && R1.weapon=G1 && R1.doors=D1+D2 &&
  U1.power=9 && U1.status=0 && R2.monster=M1 &&
  R2.doors=D3+D4 && R3.weapon=G2 && R3.doors=D5+D6 && U1.roomIn=1
}
```

Fig. 4. Maze Game Alloy Concrete Signatures and InitGame Predicate

3.3 Mapping Graph Transformation Rules to Alloy Predicates

All previous signature definitions show static parts of the Maze Game. Behavioral specifications, which are defined 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 of the Maze Game. The body of the predicate implements the LHS rule and provides matching expressions based on the current state. The predicate also contains an RHS expression to define the next state that enables the state transition as a result of task execution. If the graph transformation rule includes the NAC part, the predicate specification also contains the NAC definition.

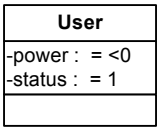
```
pred changeRoom(g:Game, g':Game, rNo:Int){
   (g.user.roomIn!=g'.user.roomIn) &&
   one room: g.rooms | one door:room.doors | one nextRoom: door.openTo |
             g.user.roomIn=room.number &&
   nextRoom.number=rNo && g'.rooms=g.rooms  &&
   g'.user.power=g.user.power && g'.user.roomIn=rNo)
}

pred subtractMonsterPower(g:Game, g':Game){
   one room: g.rooms | one monster:room.monster |
   g.user.roomIn=room.number && g'.rooms=g.rooms  &&
   g'.user.power=g.user.power-monster.power &&
   g'.user.roomIn=g.user.roomIn
}

pred checkMove(g:Game, rNo:Int){
   one room: g.rooms | one door:room.doors |
   one nextRoom: door.openTo |
   g.user.roomIn=room.number && nextRoom.number=rNo
}

run{
   one init:Game | one g2:Game | one g3:Game | one g4:Game
   |initGame[init] && ((!checkMove[init,2] && IllegalMove[init,g2]) ||
   ((checkMove[init,2]&& changeRoom[init,g2,2] &&
   ((checkMonsterPower[g2] && subtractMonsterPower[g2,g3] &&
   collectWeapon[g3,g4]) || (!checkMonsterPower[g2] && dead[g3,g4])))
} for 5 but 7 int
```

Fig. 5. Maze Game Alloy Predicates

Figure 5 shows `ChangeRoom` and `SubtractMonsterPower` predicates as examples. Predicate `SubtractMonsterPower` consists of a query part to find the current room in the Game, and a translation part to subtract monster power from the user's power and leave other parts untouched. To complete the semantics specification mapping to Alloy, we need to define branch definitions and show the sequence of predicates in the run predicate. The `CheckMove` branch task is shown as an Alloy predicate in Figure 5. The `CheckMove` predicate queries the current room and checks whether that room has the door with the number queried. The run command shows the execution sequence of each task defined by predicates. The run is executed on the predicates to show the instance for which predicates are true. Execution of the run can be restricted to the maximum number of objects with an argument. This scope declaration binds the size of the instances or counter examples.

3.4 Mapping Verification Tasks to Alloy Asserts

The verification task checks whether the given configuration is reachable from the initial graph. In the maze game, the designer defines one task to check the status of the user when he/she runs out of weapon power. This specification is transformed into an Alloy assert definition as shown in Figure 6. The `UserStatus` assert states that if power is negative, user status must be dead. The Alloy system searches the state space to find a counterexample to reveal the design problem, if it exists.




```
assert UserStatus(g:Game){
  g.user.power<0 &&
  g.user.status=1
}
```

Fig. 6. Maze Game UserStatus Assert

4 Related Work

Several techniques for the analysis of graph transformation systems exist. Varro [11] focused on automated formal verification of graph transformation systems and proposed optimization techniques that operate on dynamic parts of models. This research provided reductions in the state space during model checking. Heckel et al [12] discussed states and transitions in the context of graph systems. According to this idea, graphs are shown as states and rules are defined as state transitions. GROOVE [13] was developed based on Heckel's idea. GROOVE provides model verification activities using graph transformation rules. Rensink [14] focused on the state explosion problem in model checking and studied allocation and deallocation problems. Eshuis and Wieringa [15] proposed model verification techniques for activity diagrams. Their tool translates an activity diagram into an input format used by a model checker. Additionally, Baresi and Spoletini [9] demonstrated how the Alloy tools can be used in graph transformation systems. Their study defined transformation techniques between transformation rules and the Alloy transition system.

5 Conclusion

This paper presented a case study for the verification of simple models using Alloy. We introduced the transformation steps from DSML specifications to Alloy models by an example. Although the metamodel, model, and semantic specification are mapped into a model checker model, Alloy suffers from the state explosion problem, which yields limitations during model checking. In future work, we will study optimization techniques during mapping steps to reduce state explosion. Our future plan also includes automatic transformation of DSML specifications into Alloy

models. Our current investigation was performed manually and on a very simple example. The process needs to be generalized. Through further automation of these ideas, we plan to demonstrate that models designed by visual notations within MDE boundaries can be automatically verified by the use of existing model checking tools.

Acknowledgments. This work was supported in part by NSF CAREER award (CCF-0643725).

References

1. Schmidt, D.C.: Model-Driven Engineering. In: IEEE Computer, Volume 39, Issue 2, pp. 25-31, (2006)
2. Sprinkle, J., Mernik, M., Tolvanen, J-P., Spinellis, D.: What Kinds of Nails Need a Domain-Specific Hammer? In: IEEE Software, Volume 26, Issue 4, pp. 15-18, (2009)
3. Clarke, E.M.: The Birth of Model Checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, Volume 5000, Springer, Heidelberg, pp. 1–26, (2008)
4. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. In: IBM Systems Journal, Volume 45 , Issue 3, July, pp. 621-645, (2006)
5. Plotkin, G. A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, (1981)
6. de Lara, J., Vangheluwe, H.: Translating Model Simulators to Analysis Models. In: Proc. of Fundamental Approaches to Software Engineering (FASE 2008), Volume 4961 of LNCS, Springer, pp. 77-92, (2008)
7. Beyer, M.: AGG1.0. Tutorial, Tech. Univ. of Berlin, Dept. of Computer Science, (1992)
8. Jackson, D., Shlyakhter, I., Sridharan, M.: A Micromodularity Mechanism. In: Proceedings of the 8th European Software Engineering Conference (ESEC 2001), Vienna, Austria, pp. 62–73, (2001)
9. Baresi, L., Spoletini, P.: On the Use of Alloy to Analyze Graph Transformation Systems. In: Proceedings of the Fifth International Conference on Graph Transformation (ICGT 2006), Volume 4178 of LNCS, Springer, pp. 306–320, (2006)
10. Hausmann, J.H.: Dynamic Meta Modeling. A Semantics Description Technique for Visual Modeling Languages. PhD thesis, Universität Paderborn, Germany (2005)
11. Varro, D.: Automated Formal Verification of Visual Modeling Languages by Model Checking. In: Journal of Software and Systems Modeling, Volume 3, Issue 2, pp. 85-113, (2004)
12. Heckel, R., Ehrig, H., Wolter, U., Corradini, A.: Integrating the Specification Techniques of Graph Transformation and Temporal Logic. In: Proc. Mathematical Foundations of Computer Science (MFCS'97), Volume 1295, Springer, Bratislava, pp. 219–228, (1997)
13. Rensink, A.: The GROOVE simulator: A Tool for State Space Generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE), Volume 3062, LNCS, pp. 479–485, (2004)
14. Rensink, A.: Model Checking Graph Grammars. In: Leuschel M, Gruner S, Lo Presti S (eds) Proc. of the 3rd Workshop on Automated Verification of Critical Systems (AVOCS2003), Technical Report DSSE–TR–03–2, pp.150-160, (2003)
15. Eshuis, R., Wieringa, R.: Tool Support for Verifying UML Activity Diagrams. In: IEEE Transactions on Software Engineering, Volume 30, Issue 7, pp. 437-447, (2004)