MODEL CHECKING EARLY REQUIREMENTS SPECIFICATIONS IN ALLOY

Ton Long Phuoc*, Nguyen Minh Hai

Department of Information and Technology, Industrial University of HCMC, 12 Nguyen Van Bao, Ward 4, Go Vap District, Ho Chi Minh City

Email: tonlongphuoc@iu.edu.vn

Received: 15 June 2015; Accepted for publication: 28 July 2016

ABSTRACT

Automation generated source code and verifying are essential sector for software engineering. There are many ways to generate source code and verify from the specification languages. In this paper, we propose an approach that automatically generated code from a specification language Alloy. From this specification language, we will describe how to translate from one language to the Java source. An application in this paper is a gardening game program. Applied after the findings will be organized according to the MVC (Model-View-Controller) architectural pattern. Besides, we will also verify the identity of the structure of the application and the content of the Alloy specification. We built an tool as GmDSL, we have verified the application in GmDSL. The application was created from the tool also shows the correctness of the early constraints. Simultaneously, we also compares be verified through the GmDSL tool with NuSVM tool.

Keywords: MVC, DSL, alloy, code generation.

1. INTRODUCTION

One of the main activities of any development process aimed at creating useful software is domain modeling. Domain is the business processes being automated or the real world problem being solved by a software program. Having a good understanding of a software domain is essential for the success of any software development project. Software domains, being part of the real world, are hard to be captured directly. They have to be modeled using appropriate abstractions. Because of the great amount of knowledge related to any subject area, a domain model has to be a selection of that knowledge and unnecessary details have to be eliminated. Selection and abstraction of this knowledge requires communication between software specialists and the domain experts, who are specialists in the subject area of the software being developed.

Alloy is a lightweight modeling language, based on first order relational logic [1]. It is supported by an efficient tool called Alloy Analyzer.
In this paper, we are going to present how Alloy can be used as a language for domain modeling to make Domain Specific Languages (DSLs). With regard to software automation, computer game development has no exception. The development of a game application at least involves the following blocks: a) defining game rules, game characters, game objects, game scenes, etc.; b) describing platforms the game is supposed to run on; c) verifying as to the accuracy of the model. Suppose we can define the domain of games concretely, we may develop a DSL and its associated engine to automate most of these blocks in order to speed up the development of all the games in this domain [2].

DSL in this paper purpose of built a tool that combines the application specification languages with Alloy and the automatic code generation engine. The DSL allows verified the model that the user specified. At the same time, execute the following languages arise will be organized into the MVC architecture [3]. This means, the user specified content language applications Alloy (similar to the way the Object Constraint Language). Besides, DSL will build an engine from the pre-defined rules, which arise from the executable code for the application's content.

For example, concepts that are conceptually represented in GmDSL should directly addresses components or objects that matter in games (e.g., tools, characters, scenes). We identify the following problems for such a DSL: i) High-level specification of mobile games; ii) Automated code generation following the MVC; iii) Verifying as to the accuracy of the model, i.e allows a DSL engine to generate executable code. In this paper, we present GmDSL that addresses these three problems. We allow users to specify the game application with Alloy. We also report prototypes we have built for GmDSL. It build a game application is organized according to the MVC architecture in this paper.

The outline of this paper is as follows. Section 2 gives the related work. Section 3 is the background. Section 4 is the motivation and research problems - it presents some problems of convert from specification to Java language and how to verify the accuracy of them? Section 5 is the core of our paper - it proposes a framework and shows how the proposed framework addresses the problems identified. Section 6 draws some result of out framework. Section 7 draws some concluding remarks and points out future work.

2. RELATED WORK

Alloy as a modeling language has been used in a wide range of software engineering research studies. According to Jackson [1], these case studies and their tools for verifying and analyzing software systems can be listed as follows.

- Using Alloy in model driven development: this includes using Alloy in verifying MDA transformations and verifying models developed using other languages, such as UML, by transforming them to Alloy models [15].
- Comparing Alloy with other modeling languages such as Z or UML/OCL [9].
- Using Alloy in business process modeling: Current business process modeling tools and languages can only perform a syntactic check on the form of model. Alloy can be used to check the semantics of the business process models.

The works listed above have used Alloy as a formal language for the verification of the models but, in this paper we claim that this language can be used as a model finder as well. We
have used this capability to create instances of domain models and to generate code for the application.

One possible use of the output Alloy model is to sanity-check invariants or preconditions: the Alloy Analyzer is used to visualize instances that satisfy the invariants and/or preconditions. By manually viewing several satisfying instances, the user can verify that all of these are indeed of the expected structure. Unexpected structures indicate that the invariants/preconditions are underspecified or incorrect. In this usage, it is often necessary to write additional predicates within the Alloy Analyzer to restrict the search space. Another related use case is to visualize the post-condition for a method call with given arguments. If the Alloy Analyzer can find multiple instances for a method call with exactly specified input heap and arguments, this indicates that the post-condition does not exactly specify the result of the method. Such underspecification may be intentional, for example if the method allocates temporary helper objects without mentioning them in the contract. But it can also indicate unintentional underspecification. According Milicevic’s group [16], who present αRby, an embedding of the Alloy language in Ruby. The benefits of having a declarative specification and modeling language (backed by an automated solver) embedded in a traditional object-oriented imperative programming language. This approach aims to bring these two distinct paradigms (imperative and declarative) together in a novel way. They argue that having the other paradigm available within the same language is beneficial to both the modeling community of Alloy users and the object-oriented community of Ruby programmers. Their embedding is specific in that one of its main objectives is to preserve as much as possible the original character of the chosen specification language (Alloy), in terms of both syntax and semantics. In Figure 1, the syntax of Alloy and αRby.

| Description              | αRby                                                                 | Alloy                |
|--------------------------|----------------------------------------------------------------------|----------------------|
| Equality                 | var1 = var2                                                          | var1 = var2          |
| Sigs and Fields          | sig S [f: one (S) ** Int]                                            | sig S {f: one S -> Int} |
| Fun declaration return   | fun f[0: S] [set S] {}                                               | fun f[0: S]: set S {} |
| Fun/ Pred calls          | f1(x)                                                                | f1[x]                |
| Set comprehension        | S.select{|s| p1(s)}                                                  | {s:S | p1[s]}         |
| Quantifiers              | all(s: S) { p1(s) and p2(s) }                                       | all s: S { p1[s] and p2[s] } |
| Illegal Ruby operators   | x.in?(y), x.not_in?(y)                                               | x in y, x !in y      |
|                          | not x > y                                                           | x !> y               |
|                          | x ** y                                                              | x -> y               |
|                          | x.(y)                                                               | x.y                  |
|                          | x.size                                                              | #x                   |
|                          | y if x                                                               | x => y               |
|                          | if x then y else z                                                  | x => y else z        |
|                          | S.<! f, f.> Int                                                     | S.<! f, f.> Int      |
| Operator airty mismatch  | x.closure, x.rclosure                                               | ^x, *x               |

*Figure 1.* Examples of differences in syntax between αRby and Alloy [16].
3. BACKGROUND

3.1. Domain Specific Languages

Domain Specific Languages: A Domain Specific Language (DSL) is primarily designed to be used in a certain area - domain, abstracting away from the software implementation making implementation easier [4]. In addition, DSLs also called programming language specialization in a narrow context or a language focused on a narrow field. Though this abstraction is designed to aid the developer, the language should merely be domain complete and not be capable of solving any computational problem. DSLs have existed for many years. Languages that were created for particular domains include Fortran used to allow direct mathematical formula, Structured Query Language (SQL) for database access and manipulation, and Algol for algorithm specification. In recent times, the use of DSLs have been proposed and used in different domains including the production of rich web applications [5], smashups of web APIs and services [6], and system integration [7]. Because of the complexities in mobile software development, we notice there are problems for abstraction in the development of mobile devices.

3.2. Alloy

In software engineering, Alloy is a specification language for modeling indicates the binding of the complex structures in software technology [8]. Alloy Analyzer tool is a simple model based on the first logical hierarchy. Mathematical foundation of this language affected by the permitted margin performance. Although the syntax of this language more responsible than other languages such as restrictions on language subjects.

Alloy is a lightweight modeling language based on the first order relational logic [9]. In an Alloy module, there are a number of signatures which define sets of atoms. The definition of a signature may contain a number of fields which define relations between atoms of signatures. Signatures also serve as types, and sub-typing is possible through signatures extension. Signature extensions define new signatures as subsets of the main signature. There are also ways to define constraints in the model, using constraint paragraphs. There are four kinds of constraint paragraphs: Fact, Predicate, Function, Assertion. A model in Alloy can be instantiated as a collection of instances. Instances are binding of values to variables. The Alloy Analyzer finds instances of a model automatically by assigning values to variables satisfying the constraints defined. Model analysis involves constraint solving, and the analyzer embodies a SAT solver. It provides visualization for making sense of solutions and counterexamples it finds. Instructions to Alloy Analyzer to perform its analysis are called commands. A run command causes the analyzer to search for an instance that witnesses the consistency of a function or a predicate. A check command causes it to search for a counterexample to show that an assertion does not hold. Searching for instances is done within finite bounds, specified by the user as scope. So, when the search fails, it does not mean that there is no instance satisfying the model (i.e., the model is inconsistent).

3.3. SAT Solver

Alloy source files and their translation to Boolean Satisfiable (SAT) problems [10]. To use the Analyzer as a back-end engine, tools such as Jalloy [11] have resorted to generating Alloy source and feeding it to the Analyzer's parser, this process of generating textual models is typically slow, awkward, and error-prone. Even when accurate, the generation can yield models
that the Analyzer cannot parse efficiently, e.g. A model with a single monolithic formula not decomposed into predicates, functions, or lets.

3.4. Kodkod Logic

Kodkod is a new tool designed relations clearly as a component plug-ins that can easily be integrated as a platform of other tools. Kodkod outperforms the Analyzer dramatically on problems involving partial instances, and, due to improvements in the core technology that we describe, outperforms Alloy even on the problems for which Alloy was designed. It also outperforms other SAT-based logic engines [12].

The logic accepted by Kodkod is a core subset of the Alloy modeling language. Kodkod formulas, like Alloy formulas, are constraints on relational variables of any arity. In Alloy, these relational variables are divided into “signatures” (for unary relations) and “fields” (for non-unary relations), and the signatures establish a type hierarchy. Kodkod, in contrast, makes no special distinction between unary and nonunary relational variables, and all relations are untyped. In addition, the purely syntactic conventions of the Alloy modeling language \- predicates, functions, facts, etc \- are not part of the Kodkod logic. Think of the Kodkod logic as Alloy, stripped of its syntactic sugar [12].

4. MOTIVATION AND RESEARCH PROBLEMS

4.1. Example

In this section, we present some typical problems. They can be solved with different technologies. We first describe the basic case study. Then we look for corresponding characteristics in order to establish a list of generic problems.

In this subsection, we briefly describe an example in which a developer needs to build a gardening game that allows a single player to act as a gardener [19]. First, She picks up a few gardening tools (shovels, scissors, sprinkler and more) and starts the game with a collection of plants: rose, tulip, sunflower, daisies, etc. The game comes up with a set of rules governing how she could grow up her garden using resources. The gardener is provided with a initial resource of 1,000 points. Then, she uses this resource to buy flower seeds (roses are worth 10 points, tulips 15 and sunflowers 10 points). She sows her garden with the seeds. She grows them and finally has them harvested. This order of gardening activities must be respected. In terms of tools, the gardener has a shovel (used for sowing) that costs 10, a sprinkler (used for growing) 15 and scissors (used for harvesting) 10 points. Additional logic of this game can be summarized in the following.

- The activity of sowing costs the gardener some amount of resource: roses 5 points, sunflowers 8 and tulips 7.
- A flower needs to be grew within 10 seconds after it has been sowed or it will have no value.
- A flower needs to be watered within 10 seconds after it has been grew or it will loss 3 points.
- The activity of growing (watering) adds the flower some amount of resource: rose 5 points, sunflower 10 and tulip 7.
– A flowers will change status of harvest if sunflowers has some amount of resource: rose 20 points, tulips 15 and tulips 20.
– A flower need to be harvesting with 8 seconds after it has been subtracted 2 points or it will added points to the gardener.
– The gardener will win the game when she obtains 3,000 points. The game is over when she loses all her points. Now, the game developer needs to build this game application.

Suppose that the game developer has a DSL framework. It translates what she describes the game conceptually and semantically by Alloy into executable code [13] and verifies them. We highlight the following problems the game developer may get into. Distribute for items with subsection 4.2.

4.2. Research Problems

Based on the analysis of the concrete problems for the aforementioned example in above subsection, we generalize our research problems as follows.

- **High-level specification of mobile games.** Game applications will be represented in a high-level, declarative language such as Alloy. Specifically, game rules are described as Alloy facts, concepts as Alloy signatures.

- **Verifying the specification of mobile games.** The specification of the game will verify before generated source code. How to verify the accuracy of them?

- **Automated code generation.** How do tool convert the whole of the specification on the implementation language? The conversion required to ensure coherence and rationality to the context.

5. FRAMEWORK

To solve the problems mentioned above, we propose a process for the DSL tools [2] include the following components: First, the user will specify the objects in the game as well as of the operating scenario in the game, such as seeding, nurtured, harvesting, when the game will win, fail... All these specifications will be specified in the language Alloy. Then, it will verify these specifications by SAT solver and NuSMV [17] tool. In there, the rule for contraints on the action between objects are organized according to two following formats: Player rules allow users to define, making the transition from model to enforce language become tighter while ensuring the game has the same function with the original specification. Other rules: As the rule is specified for all gaming applications. The design will enhance GmDSL calculated automatically at the same time elements to expand the scope of the program application. Additionally, the tool generates code GmDSL after execution are organized by the MVC architecture, makes expanding or switching between platforms becomes easier [3].

5.1. Software Architecture

In GmDSL, the users who want to design the gardening game such application will perform the following steps: Figure 2 An outline of GmDSL.
Step 1 Alloy Editor: Users design the game through tool making graphical models and editing the content of game by text. Then additional components such scenario parameters declared for the game: title, scorewin, scorefail, etc. Users design the characters in the game such as tool: shovels, scissors, sprinkler, etc. The characters are cultivation (product): rose, tulip, sunflower, etc. Which character is full of attributes such as status, bonus. The action in rule combine tool and flower. This specifications will be declared by Alloy languages. It will be drafted by the user. First, we introduce Character, Player, Role, Action and Resource, with Action and Resource being used to behavior the contents of a Character.

```
open util/ordering [Time] as TO
sig Time 

sig GameState 

one sig WON, GAMEOVER, PLAYING extends GameState 

one sig Farmer 
resource: set (Int one -> Time),
game_state: set (GameState one -> Time) { 
    all t: Time | resource.t >= 0 && resource.t <= 5 }

sig Flower 
    score: set (Int one -> Time) { 
        all t: Time | score.t >= 0 && score.t <= 3 }

sig Rose, Tulip extends Flower 

sig Tool 
    cost: set (Int one -> Time) { 
        all t: Time | cost.t >= 0 && cost.t <= 3 }

sig Hoe, Sprinkler extends Tool 
```

Figure 3. Concepts of the gardening game expressed in Alloy.

The declaration above statement describes the content of the game application (Figure 3). Which describes the components of the game such as: Resources for the main characters, the supporting characters such kinds of flowers and tools for gardening.
Figure 4. Rules of the gardening game expressed in Alloy.

These are the statements specification for the rules of the gardening game as Figure 4. These rules will be specified by the fact and predicate in Alloy language. These rules will be converted to the Rule Engine. Since then, the rules will support for verifying the behavior of the objects in the game application.

**Step 2 Verifying (Alloy API):** Select inspection tools and semantic translation to check syntax and generate source code for game. The editor will check syntax the content by Alloy grammar and synch between conceptual model with text. Then, using Alloy analyzer to verify the specifications before using tool nuSMV.

**Step 3 Mapping:** Choose incurred to generate the code for the class.

**In Alloy:**

1. Sig StateGame <-> Class StateGame
2. Sig StateWin, StateFail, StateNormal extends StateGame

**In Java:**

1. Class StateWin extends StateGame
2. Class StateFail extends StateGame
3. Class StateNormal extends StateGame
**Step 4 Finish:** Compile and execute programs. This is the interface for the design of applications with GmDSL. In this interface the user can design concept model and specification of the rules. In particular, these rules will be checked the validity of the semantic processor and combined with the model to generate source code for the program. In the 4 is the architect of our games.

![Software Architecture of the Game](image)

**5.2. Code generation**

We also developed a simple tool that allows a software developer can specify an application by text. They declare the rules according our grammar. Then, the engine will check the coherence and rationality between model and the rules, converted into executable code. In step 4, when the user has it taken steps in GmDSL, the program will automatically generate source code for the program. In which the structure of the program will be divided into 3 packages. View package will focus on the part of the game interface, the package will focus on the control handle and package model rule would specify the character in the game. In Figure 5, The source code generated from Alloy will organized MVC architecture.

In the Controller package contains classes such as Universe, Evaluator and Boundary. This is dynamic universe.

In the View package contains classes such as start-up scene, main scene and the finish scene.

In the Model package package contains classes such as flower, tool, farmer.

**5.3. Verifying the dynamics of the game data**

When the user changes the architecture of game application such as changing the primary character, changing the behavior of secondary characters, the rules... The system will automatically verify with the rules arising from Alloy as Figure 6. This means, the system will turn to take steps
by step. Firstly, changing a user event data structure leads to change. And then, the boundary will be update. Finally, with the Formula rules and kodkod APIs will verify through the Evaluator.

Refresh Boundary: Once the specifications are built, the next step is to keep track of the data structure in game event. To store the state of the program, we demonstrate the object graph using relations. A part of the tool is syn of the data structures. This part will be modified by the user to map the Java data structure to its Alloy representative. Basically, the user indicates the mapping intended between the Java data structures and Alloy signatures. This manual configuration can be skipped too using techniques described in [14]. Once the part responsible for translating the data structure to Alloy and Kodkod is constructed, the program can traverse data structures automatically and translate them to Alloy and Kodkod. This is the procedure we called mapping formula in Figure 5. For each object that appears in pre- or post-states there would be one signature instance in Alloy. Before and After running each method of the program (when the game event is activity), the abstraction part is called to keep track of the object graphs and write them to be sent to the SAT solver. That means the reverse action of abstraction and translates a repaired data structure back to Java. During translation of the program state to Kodkod (Evaluator). For each relation, there would be a boolean indicating if this relation is relaxed. If the relation is not relaxed, it would be bounded exactly to its set in synchronization. For example, when a flower is sowing, farmers began watering flower which is growth (achievement scores for flower). That mean, the data structure of the flower will change on scores.

Figure 6. The processing of the GmDSL tool.

6. EXPERIMENTAL RESULTS

Figure 7. The description of checking property with SAT solver on Alloy Analyzer 4.2.
In this section, we report the results obtained from conducting several experiments. We have verified the applications of games on the SAT solver, and the results were satisfactory the requirements such as, the correctness of the specification, visual representation, etc. The results is described in Figure 6.

We performed the experiments on Windows 8 with Intel Core i5-3340M, 2.70 GHz and 8GB memory. We compare our approach with the model checking tool NuSMV, an open source tool for the model checking on finite state systems. The property for comparison is described in Figure 8, the result of NuSMV [20] tool for model checking.

\[
\begin{align*}
\text{EF}((\text{state}(\text{pushl}, \text{REG}, \text{NULL}, \text{NULL})) \\& \text{EF}((\text{state}(\text{movl}, \text{r}, \text{esp}, \text{s_f}s\_0 \_s, \text{NULL})))) \\
\text{EF}((\text{state}(\text{pushl}, \text{REG}, \text{NULL}, \text{NULL})) \\& \text{EF}((\text{state}(\text{movl}, \text{r}, \text{esp}, \text{s_f}s\_0 \_s, \text{NULL})))) \\
\text{EF}((\text{state}(\text{call}, \text{REG}, \text{NULL}, \text{NULL})) \\& \text{EF}((\text{state}(\text{movl}, \text{r}, \text{esp}, \text{s_f}s\_0 \_s, \text{NULL})))) \\
\text{EF}((\text{state}(\text{call}, \text{REG}, \text{NULL}, \text{NULL})) \\& \text{EF}((\text{state}(\text{movl}, \text{r}, \text{esp}, \text{s_f}s\_0 \_s, \text{NULL}))))
\end{align*}
\]

Figure 8. The description of checking property with NuSMV.

| Tools     | Vars | Primary vars | Clauses | Duration | Bitwidths |
|-----------|------|--------------|---------|----------|-----------|
| GmDSL (SAT) | 38,182 | 1,370        | 124, 206 | 636 ms   | 6         |
| NuSMV     | 29,192 | 1,103        | 120, 139 | 643 ms   | 6         |

Figure 9. The results of verification from GmDSL and NuSMV.

In general, as presented in Figure 9, our approach produces the better results compared with NuSMV. The reason is that our approach combine the feature of SAT solver for solving the path condition which allows our approach to generate test-cases and follow the appropriate paths. However, the main drawback here is the computational limitation of solver. Current solvers mostly cover only linear constraints for arithmetic. For non-linear constraints, we targets to adopt raSAT.

7. CONCLUSION AND FUTURE WORK

In this paper, we propose a new approach of automatically generating code and verifying from the specification language Alloy. By adopting Alloy as a domain modeling language, we can easily visualize models using Alloy Analyzer which can be understood by the domain
experts. In addition, we can implement their instances and check their correctness using assertions. One of the main features of this approach for verification is that it requires no test cases implying. We can present both behavioral and structural aspects of the domain at the same time. In GmDSL, we also verify the specifications for game by Alloy with SAT and NuSMV tool. We realize that the GmDSL tool used Alloy language specification gives better results than the NuMVC tool.

However, our approach also encounters some problems. The problem with overriding could be solved by adding a new field to the child's fields. For handling the scalability we divide our model into separate modules using Domain Specific Languages as mentioned in Section 3.

**Future Work:** For the future works, we will complete our tool GmDSL with the following objectives.

- Improve engine specification drafting and upgrade tool specification verification of applications.
- Handle additional semantics for the specification of the rule in more complete ways by another programming languages.
- Develop a tool for compiling engine design classes to execute Java language or C\{\#\} language.

**REFERENCES**

1. Jackson D. - Software Abstractions, Logic, Language, and Analysis. volume 2, MIT Press, (2006), 36-40.
2. Mernik M. and Heering J. and Sloane A. M. - When and how to develop domain-specific languages. ACM Computing Surveys 37(4) (2005), 316-344.
3. Yacoub S. M., and Ammar H. H. - Pattern-oriented analysis and design: composing patterns to design software systems. Addison-Wesley Professional, 2004, pp. 288-292.
4. Fowler M. Domain – Specific languages, volume 2,Addision Walley, 2010, pp. 27-32.
5. Oussena S., Kramer D., Clark T. - MobDSL: A Domain Specific Languages for multiple mobile platform deployment. In proceedings of the 2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications (NESEA), IEEE Computer Society (2010), 1-7.
6. Buchli J., Frigerio M., and Caldwell D. G. - A Domain Specific Languages for kinematic models an. M d fast implementations of robot dynamics algorithms, arXiv preprint arXiv, 20, (2013), 42-50.
7. Tuleu A. and Nordmann A. and Wrede S. - A Domain-Specific Languages and Simulation Architecture for the Oncilla Robot. In Proceedings of the ICRA 2013 Workshop on Developments of Simulation Tools for Robotics & Biomechanic, IEEE Computer Society, Karlsruhe, Germany, May (2013), 25-26.
8. Anastasakis K., Bordbar B., Kauster J. M. - Analysis of Model Transformations via Alloy. In Proceedings of the 4th Model-Driven Engineering, Verication and Validation workshop, Springer, Nashville, USA, October (2007), 47-56.
9. Jackson D. - A Comparison of Object Modelling Notations: Alloy, UML and Z. Technical report, MIT, August (1999), 8-14.
10. Zhang L. and Malik S. - Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In Proceedings of the conference on Design, Automation and Test in Europe, IEEE Computer Society, Munich, Germany (2003), 10880-10889.

11. M. Vaziri and D. Jackson. - Checking properties of heap-manipulating procedures with a constraint solver. Tools and Algorithms for the Construction and Analysis of Systems, Springer, (2003), 505-520.

12. Torlak E. and Dennis G. - Kodkod for Alloy users. In Proceedings of the First ACM Alloy Workshop, Springer, Portland, USA, December (2006), 45-53.

13. Bedekar M., Birla A. A. - MobiDSL: a Domain Specific Language for Mobile. In Proceedings the 9th OOPSLA workshop on Domain-Specific Modeling (DSM’09), ACM, Birla, India, (2009), 130-150.

14. Khalek S. A., Yang G., Zhang L., Marinov D., and Khurshid S. - Testera: A tool for testing java programs using alloy specifications. In Proceedings of the 26th IEEE/ACM international conference on automated software engineering, IEEE Computer Society, Washington D.C, USA, (2011), 608-611.

15. Georg G., Anastasakis K., Bordbar B., and Ray I. - UML2Alloy: A Challenging Model Transformation. In Proceedings ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems, Springer, Nashville, USA, (2007) 3-5.

16. Milicevic A. and Efrati I., and Jackson D. - Rby An Embedding of Alloy in Ruby. In Abstract State Machines, Alloy, B, TLA, VDM, and Z, Springer, 2014, pp. 56-71.

17. Cimatti A., Clarke E. M., Giunchiglia E., Giunchiglia F., Pistore M., Roveri M., Sebastiani R., and Tacchella A. - NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Proceeding of International Conference on Computer-Aided Verification (CAV 2002), Copenhagen, Denmark, July, 2002, pp. 27-31.

18. To K. and Ogawa M. - SMT for Polynomial Constraints on Real Numbers, Tools for Automatic Program Analysis TAPAS 2012, Elsevier ENTCS 289 (2012) 27-40.

19. Ton L. P. and Truong T. M. - Linking Rules and Conceptual Model in a Domain Specfic Language. In Proceedings of 9th International Conference on Advanced Computing and Applications (ACOMP), IEEE Computer Society, Ho Chi Minh City, Vietnam, 2015, pp. 35-42.

20. Cimati A. and Clarke E. and Giuchiglia F. and Roveri M. - NuSMV: a new symbolic model checker, International Journal on Software Tools for Technology Transfer, Springer 4 (2) (2000) 410–425.