DynAlloy as a Formal Method for the Analysis of Java Programs

Juan P. Galeotti and Marcelo F. Frias

Department of Computer Science
School of Exact and Natural Sciences
University of Buenos Aires
Argentina
e-mail: {jgaleotti, mfrias}@dc.uba.ar

Abstract. DynAlloy is an extension of the Alloy specification language that allows one to specify and analyze dynamic properties of models. The analysis is supported by the DynAlloy Analyzer tool. In this paper we present a method for translating sequential Java programs to DynAlloy. This allows one to use DynAlloy as a new formal method for the analysis of Java programs. As an application showing the utility of this formal method toward this task, we present JAT, a tool for automated generation of test data for sequential Java programs, implemented on top of the DynAlloy Analyzer.

1 Introduction

Alloy [9] is a relational modeling language. Its simplicity, object-oriented flavor, and automated analysis support, have made this formal language appealing to a growing audience. The Alloy Analyzer [10] transforms Alloy specifications (models) in which domains' sizes are bounded to a fix scope, into propositions that are later fed to SAT-solvers such as Berkmin [6] or MChaff [15]. Then, given an assertion to be verified in the model, the Alloy Analyzer attempts to produce a model of the specification that violates the assertion. If no such model is found within the provided scopes, we can gain more confidence that the analyzed property holds in the model. Of course, a counterexample suffices to show that the model is flawed. We will include a description of Alloy's syntax and semantics in Section 2. It is nevertheless worth mentioning at this point that Alloy models are static. That is, while Alloy functions seem to model an input–output behavior by relating input and output variables, the classical first-order semantics prevents actual state change or evolution.

The DynAlloy specification language was first introduced in [4] as an extension of the Alloy language allowing us to cope with the lack of dynamics of Alloy. DynAlloy's semantics is based on dynamic logic [8], making then possible to specify atomic actions (and complex actions from these) that actually modify the state. It also allows one to assert properties about these actions by means of partial correctness assertions [2]. In [3] we presented the DynAlloy Analyzer, which allowed us to effectively analyze DynAlloy specifications.

Please use the following format when citing this chapter:

Galeotti, J.P., Frias, M.F., 2006, in IFIP International Federation for Information Processing, Volume 227, Software Engineering Techniques: Design for Quality, ed. K. Sacha, (Boston: Springer), pp. 249–260.

Contributions of the Paper

- From the foundational point of view, this paper introduces a translation of sequential Java programs to DynAlloy. This translation provides us with a new formal method for the analysis of Java programs using the DynAlloy Analyzer.
- In the applications side, we introduce JAT (Java Automated Testing), an application of DynAlloy to the automated generation of test data for sequential Java programs. JAT allows a user to generate test data according to various structural testing criteria such as statement coverage, branch coverage or path coverage. JAT also provides the user with information about non reachable code, and profits from the existence of invariants and pre conditions written in JML [14], and previous partial test suites, if these are available.

The paper is structured as follows. In Sections 2 and 3 we give brief introductions to Alloy and DynAlloy, respectively. In Section 4 we show how to translate sequential Java programs to DynAlloy. In Section 5 we present JAT, compare it with related work, and evaluate its performance through examples. Finally, in Section 6 we present our conclusions.

2 The Alloy Specification Language

In this section, we introduce the reader to the Alloy specification language by means of an example. This example intends to illustrate the standard features of the language and their associated semantics, and will be used in further sections.

Suppose we want to specify systems handling lists. We might recognize that, in order to specify lists, a data type for the data stored in the lists is necessary. We can then start by indicating the existence of a set (of atoms) for data, which in Alloy is specified using a *signature*:

sig *Data* { }

In this signature we do not assume any properties about the structure of data.

With data already defined, we can now specify what constitutes a list. A possible way of defining lists is by saying that a list consists of a datum, and an attribute *next* relating the current node to the remaining part of the list:

sig *List* { val : lone *Data*,
next: lone *List* }

The modifier "lone" in the above definition indicates that attributes "val" and "next" may relate a list with at most one element. These are *partial* functions from *List* to *Data* and *List* to *List*, respectively.

Alloy allows for the definition of signatures as subsets of the set denoted by another "parent" signature. This is done via *signature extension*. For example, one could define other (perhaps more complex) kinds of lists as extensions of the *List* signature:

sig *Empty* extends *List* {}
sig *TwoList* extends *List* { val2: Data }

```
problem ::= decl* form
decl ::= var : typexpr
typexpr ::=
type
| type → type
| type ⇒ typexpr

form ::=
expr in expr (subset)
|!form (neg)
| form and form (conj)
| form or form (disj)
| all v : type/form (univ)
| some v : type/form (exist)

expr ::=
expr + expr (union)
| expr & expr (intersection)
| expr − expr (difference)
|~ expr (transpose)
| expr.expr (navigation)
| *expr (refl. trans. closure)
| ^expr (transitive closure)
| {v : t/form} (set former)
| Var

Var ::=
var (variable)
| Var[var] (application)
```

Fig. 1. Grammar of Alloy

As specified in these definitions, *Empty* and *TwoList* are special kinds of lists. In *TwoList*, a new attribute val2 is added to each list. As the previous definitions show, signatures are used to define data domains and their structure. The attributes of a signature denote *relations*. For instance, the "val" attribute in signature *List* represents a binary relation, from list atoms to atoms from *Data*. Given a set L (not necessarily a singleton) of *List* atoms, L.next denotes the relational image of L under the relation denoted by next. Signature extension, as we mentioned before, is interpreted as inclusion of the set of atoms of the extending signature into the set of atoms of the extended signature.

In Fig. 1, we present the grammar and the (informal) semantics of Alloy's relational logic, the core logic on top of which all of Alloy's syntax and semantics are defined. Adding a bit more of notation, given singleton unary relations $A = \{ a \}$ and $B = \{ b \}$, we define $A \rightarrow B = \{ \langle a, b \rangle \}$. Given a binary relation R, we define the update of R by the pair $A \rightarrow B$ by

$$R \mathrel{++} (A \rightarrow B) = \{ \langle x, y \rangle \in R : x \neq a \} \cup \{ \langle a, b \rangle \} \ .$$

So far, we have just shown how the structure of data domains can be specified in Alloy. These models can be enriched with the addition of *operations*, *properties* and *assertions*. Following the style of Z specifications, operations in Alloy can be defined as expressions, relating states from the state spaces described by the signature definitions. Primed variables are used to denote the resulting values, although this is just a convention not reflected in the semantics. In order to illustrate the definition of operations in Alloy, consider, for instance, an operation that specifies the appending of a datum to the front of a list (usually called Cons):

```
pred Cons(d : Data, l, l' : List){
    l'.val = d and l'.next = l}                    (1)
```

As the reader might expect, a model can be enhanced by adding properties (axioms) to it. These properties are written as logical formulas called *facts* in Alloy. We reproduce some here. It might be necessary to say that lists are acyclic

```
fact AcyclicLists{ all l : List | l !in l.(^next) }
fact OneEmpty{ one Empty }.
```

The keyword "one" states that the set (unary relation) Empty is a singleton. More complex facts can be expressed by using the quite considerable expressive power of the relational logic. Assertions are the *intended* properties of a given model. Consider, for instance, the following simple Alloy assertion, regarding the presented example:

assert ToEmpty{ all l: *List* | l != Empty implies Empty in l.^next }

This assertion states that non empty lists (eventually) reach the empty list. Assertions are used to check specifications. Using the Alloy analyzer it is possible to validate assertions by searching for possible (finite) counterexamples for them under the constraints imposed in the specification.

3 The DynAlloy Specification Language

DynAlloy is an extension of the Alloy modeling language. It allows us to define atomic actions that modify the state, and build more complex actions from the atomic ones. Atomic actions are defined by means of pre and post conditions given as Alloy formulas. For instance, atomic actions that retrieve the first element in a list, or remove the front element from a list are specified by

```
act Head(l : List, d : Data)          act Tail(l : List)
  pre = { l != Empty }                  pre = { l != Empty }
  post = { d' = l.val }                 post = { l' = l.next }
```

The primed variables d' and l' in the specification of actions Head and Tail denote the value of variables d and l in those states reached *after* the execution of the actions. While actions may modify the value of all variables, we assume that those variables whose primed versions do not occur in the post condition retain their corresponding input values. Thus, Head modifies the value of d, but l keeps its initial value. This allows us to use simpler formulas in pre-post conditions.

Equally important, DynAlloy allows us to assert properties about complex actions by means of partial correctness assertions. For instance,

```
{ l != Empty }
  Head(l, d);
  Tail(l)
{ Cons(d',l',l) }
```

The syntax of DynAlloy's formulas extends the one presented in Fig. 1 with the addition of the following clause for building partial correctness statements:

$$formula \;\; ::= \;\; \ldots \mid \{formula\}\ program\ \{formula\} \text{ “partial correctness”}$$

Figure 2 shows how complex actions are built from atomic ones. Figure 3 describes the semantics of DynAlloy.

One of the important features of Alloy is the automatic analysis possibilities it provides. Similarly, in [3] we show how to translate DynAlloy specifications to Alloy specifications in order to achieve analyzability. We reproduce the fundamental aspects

act	$::=$	$p\{pre(\overline{x})\}\{post(\overline{x})\}$	"atomic action"
	$\mid$	$formula?$	"test"
	$\mid$	$act + act$	"non-deterministic choice"
	$\mid$	$act ; act$	"sequential composition"
	$\mid$	act^*	"iteration"

Fig. 2. Grammar for DynAlloy's Actions

$$M[\{\alpha\}p\{\beta\}]e = M[\alpha]e \implies \forall e' \left(\langle e, e'\rangle \in P[p] \implies M[\beta]e'\right)$$

$$P : program \rightarrow \mathcal{P}\,(env \times env)$$

$$\begin{array}{lcl} P[\langle pre, post\rangle] & = & \{\, \langle e, e'\rangle : M[pre]e \wedge M[post]e' \,\} \\ P[\alpha?] & = & \{\, \langle e, e'\rangle : M[\alpha]e \wedge e = e' \,\} \\ P[p_1 + p_2] & = & P[p_1] \cup P[p_2] \\ P[p_1 ; p_2] & = & P[p_1] ; P[p_2] \\ P[p^*] & = & P[p]^* \end{array}$$

Fig. 3. Semantics of DynAlloy.

of this translation below, and refer the reader to [3] for optimizations. We define below a function $wlp : program \times formula \rightarrow formula$ that computes the weakest liberal precondition [2] of a formula according to a program (composite action). We will in general use names $x_1, x_2 \ldots$ for program variables, and will use names $x'_1, x'_2, \ldots$ for the value of program variables *after* action execution. We will denote by $\alpha|_x^v$ the substitution of all free occurrences of variable x by the fresh variable v in formula α.

When an atomic action a specified as $a\{pre(\overline{x})\}\{post(\overline{x}, \overline{x'})\}$ is used in a composite action, formal parameters are substituted by actual parameters. Since we assume all variables are input/output variables, actual parameters are variables, let us say, $\overline{y}$. In this situation, function wlp is defined as follows:

$$wlp[a(\overline{y}), f] = pre|_{\overline{x}}^{\overline{y'}} \implies \text{all } \overline{n} \left(post|_{\overline{x'}}^{\overline{n}}|_{\overline{x}}^{\overline{y'}} \implies f|_{\overline{y'}}^{\overline{n}} \right) . \tag{2}$$

A few points need to be explained about (2). First, we assume that free variables in f are amongst $\overline{y'}$, $\overline{x_0}$. Variables in $\overline{x_0}$ are generated by the translation function $pcat$ given in (3). Second, $\overline{n}$ is an array of new variables, one for each variable modified by the action. Last, notice that the resulting formula has again its free variables amongst $\overline{y'}$, $\overline{x_0}$. This is also preserved in the remaining cases in the definition of function wlp.

For the remaining action constructs, the definition of function wlp is the following:

$$\begin{array}{lcl} wlp[g?, f] & = & g \implies f \\ wlp[p_1 + p_2, f] & = & wlp[p_1, f] \wedge wlp[p_2, f] \\ wlp[p_1 ; p_2, f] & = & wlp[p_1, wlp[p_2, f]] \\ wlp[p^*, f] & = & \bigwedge_{i=0}^{\infty} wlp[p^i, f] . \end{array}$$

Notice that wlp yields Alloy formulas in all these cases, except for the iteration construct, where the resulting formula may be infinitary. In order to obtain an Alloy formula, we can impose a bound on the depth of iterations. This is equivalent to fixing a maximum length for traces. A function $Bwlp$ (bounded weakest liberal

precondition) is then defined exactly as wlp, except for iteration, where it is defined by $Bwlp[p^*, f] = \bigwedge_{i=0}^{n} Bwlp[p^i, f]$, and n is the scope set for the depth of iterations.

We now define a function $pcat$ that translates partial correctness assertions to Alloy formulas. For a partial correctness assertion $\{\alpha(\overline{y})\}\ P(\overline{y})\ \{\beta(\overline{y}, \overline{y'})\}$

$$pcat\left(\{\alpha\}\ P\ \{\beta\}\right) = \forall \overline{y} \left(\alpha \implies \left(Bwlp\left[p, \beta|^{\overline{x_0}}_{\overline{y}}\right]\right)|^{\overline{y}}_{\overline{y'}}|^{\overline{y}}_{\overline{x_0}}\right) . \qquad (3)$$

Of course this analysis method where iteration is restricted to a fixed depth is not complete, but clearly it is not meant to be; from the very beginning we placed restrictions on the size of domains involved in the specification to be able to turn first-order formulas into propositional formulas. This is just another step in the same direction.

4 Translating Java Programs to DynAlloy

It will be made clear in this section that once DynAlloy is available, translating Java becomes immediate. It is also clear that other programming languages, or description languages, can be easily translated to DynAlloy without requiring complicated ad-hoc translations. In Fig. 4 we present the grammar for the subset of Java we will translate in this article. We have also dealt with dynamic dispatch, but is not treated in this paper due to space limitations.

```
program ::= classdecl* procdecl*
classdecl ::= class class {class field;}
procdecl ::= class static proc (class var,){stmt}
stmt ::= var = new class()
   | var = expr
   | expr.field = expr
   | while pred { stmt }
   | if ( pred ) stmt else stmt
   | stmt ; stmt
expr ::= null | var | expr.field
   | expr.proc(expr,. . .,expr)
pred ::= expr (boolean)
   | expr == expr | !expr | expr && expr
```

Fig. 4. The syntax of a subset of Java

In order to handle aliased objects appropriately, we adopt the object model of JAlloy [11]. The JAlloy model of the List signature requires just a basic signature for lists without fields

sig List { } ,

and fields are considered as binary relations

val : List $\rightarrow$ lone Val,
next : List $\rightarrow$ lone List .

These binary relations can be modified by the DynAlloy actions. We will in general distinguish between simple data that will be handled as values, and structured objects.

Action SetNext is now specified as follows:

act SetNext(l1, l2 : List, next : List $\rightarrow$ lone List)
  pre = { l1 != Empty }
  post = { next' = next ++ (l1 $\rightarrow$ l2) }

We introduce now in DynAlloy atomic actions that create objects, and atomic actions that modify an object's field. We denote by $\mathrm{Objects}_C$ the unary relation (set) that contains the set of objects from class C alive at a given point in time. This set can be modified by the effect of an action. In order to handle creation of an object of class C in DynAlloy, we introduce an atomic action called NewC, specified as follows:

act NewC(o : C)
  pre = { true }
  post = $\{o'$!in $\mathrm{Objects}_C$ and o' in $\mathrm{Objects}_C{}'\}$

Notice that $\mathrm{Objects}_C$ should have been passed as a parameter. In order to maintain notation simple, we keep this kind of variables global. An atomic action that sets the value of field f of object o, is described in DynAlloy as follows:

act Setf(o : C, v : C', f : C $\rightarrow$ C')
  pre = { o in $\mathrm{Objects}_C$ }
  post = { f' = f ++ (o $\rightarrow$ v) }

From the class extension hierarchy in Java, a signature extension hierarchy is defined in DynAlloy. A class declaration

class C {
  C_1 field_1;
  ⋮
  C_k field_k;}

produces a DynAlloy model that includes definitions for a signature C and the necessary actions for creating objects and modifying their fields:

sig C { }

NewC(o : C)

$\mathrm{Setfield}_1$(o : C, v : C_1, field_1 : C $\rightarrow$ C_1)
⋮
$\mathrm{Setfield}_k$(o : C, v : C_k, field_k : C $\rightarrow$ C_k)

We proceed now to the translation of simple statements.

$$\text{v = new C} \quad \mapsto \quad \text{NewC(v)} .$$

In order to translate assignment of an expression to a variable, we introduce action VarAssign as follows:

act VarAssign(v1, v2 : C)
pre = { true }
post = { v1' = v2 }

The translation then becomes:

$$\mathrm{v = e} \quad \mapsto \quad \mathrm{VarAssign(v,e)}\ .$$

In order to translate the assignment of an expression e to the f-field of an object o, we use action Set$f$. The translation of the statement then becomes:

$$o.f = e \quad \mapsto \quad \mathrm{Set}f(o,e,f)\ .$$

For more complex program constructs, the translation is defined as follows,

while pred {stmt}	$\mapsto$	$(\mathrm{pred?};\mathbf{stmt})^*;(!\mathrm{pred})?$,
if (pred) stmt1 else stmt2	$\mapsto$	$(\mathrm{pred?};\mathbf{stmt1}) + ((!\mathrm{pred})?;\mathbf{stmt2})$,
stmt1 ; stmt2	$\mapsto$	$\mathbf{stmt1};\mathbf{stmt2}$,

where the boldface **stmt**, **stmt1** and **stmt2** stand for the recursive application of the translation to the statements stmt, stmt1 and stmt2, respectively.

5 Test-Data Generation with JAT

JAT is a tool that generates test input data for Java methods according to different white-box testing criteria. The current prototype of JAT does statement coverage, branch coverage and path coverage. In order to obtain a finite Alloy formula, we finitize the code. This is done by performing up to a predetermined (user defined) number of loop unrolling or recursive call unfolding. Also, for those methods called from the analyzed method that are provided with a JML contract, the user can choose whether she/he prefers to use the contract in the test generation process, or rather to inline the code in the caller method.

In Section 5.1 we describe the architecture of JAT. In Section 5.2 we present a case study. Finally, in Section 5.3 we analyze related work.

5.1 The Architecture of JAT

Figure 5 provides the architecture of JAT. Boxed entries in the figure correspond to processes, while non boxed entries correspond to data. Arrows show the flow of data between processes. JAT takes, as a mandatory input, compilable source Java code, together with an indication of the method to be tested. Optional inputs to the tool are a partial JUnit [12] test suite, and JML [14] annotations for data invariants and pre conditions of methods.

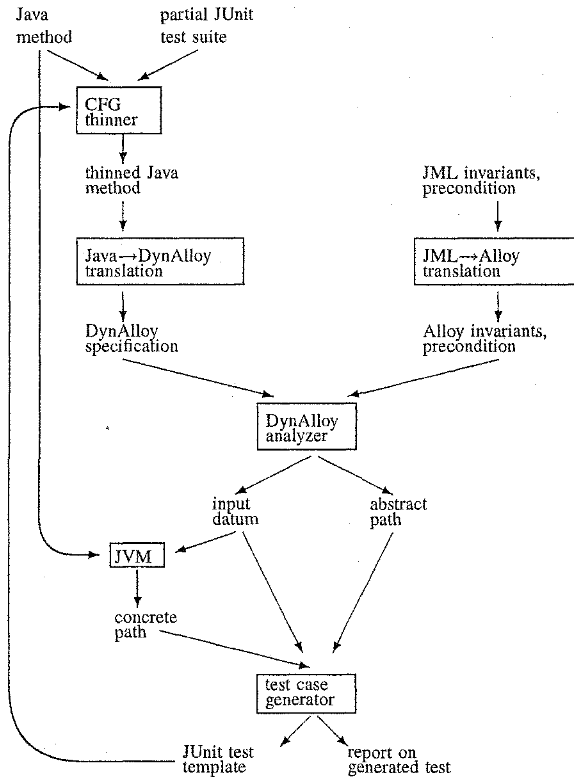

Fig. 5. Architecture of JAT

The Control Flow Graph Thinner The thinner starts by constructing the method's control flow graph (CFG). Given the source code for the method under analysis, and the partial test suite (if any tests are available), the CFG thinner analyzes the coverage produced by the provided test suite. It runs first the method under analysis in all input data available in the provided test suite, and marks the traversed statements and conditions in the CFG. In this way we have a partial coverage of the CFG. Notice that a good starting test suite will greatly improve the test input data generation process.

The CFG thinner then produces appropriate subgraphs of the CFG to be translated to DynAlloy in order to look for new test input data. From the supplied subgraph of the CFG, JAT produces one input datum. The source method is then executed on this input, and the coverage marking in the original CFG is updated by adding the marking of the newly covered statements and conditions. The thinning process then starts again from the newly marked CFG.

Notice that retrieving proper subgraphs of the CFG allows us to get better analysis times by reducing the size of the problem to be solved.

The JML→Alloy Translator JML [14] preconditions and data invariants allow us to generate better test input data, i.e., generation of data that does not satisfy invariants or is not expected as input from the method is prevented.

The DynAlloy Analyzer The DynAlloy Analyzer is used in order to find a model of the specification produced by the Java→DynAlloy translation. In case the DynAlloy Analyzer succeeds in finding an appropriate model, from this model we can retrieve a path on the DynAlloy code (and therefore on the original Java code), as well as an input datum i.

The Test Case Generator We will call the path in the Java code inferred from the DynAlloy Analyzer *abstract*, as opposed to the *concrete* path in the Java code obtained by executing the Java source code with input i on the Java virtual machine. Although it seems like both paths ought to be the same, the use of incorrect JML specifications may produce wrong paths. The test case generator then generates the concrete path and compares the abstract and concrete paths. If they are different, it generates a report for the user, and a JUnit test showing the difference between the value obtained by the concrete execution and the value at the end of the abstract path is generated. This helps the user in finding bugs in the JML specifications. In case the abstract and the concrete path agree, a JUnit test template is generated containing a description of the found input datum. This template is then fed to the CFG thinner.

5.2 Case Study: Red-Black Trees

In order to evaluate the usability of JAT, we looked for branch coverage in an implementation of sets using red-black trees. The implementation was obtained from the class TreeMap in the java.util package. We analyzed the method "add" that inserts an element in a tree and restores the red-black tree invariant.

In order to handle trees whose height is less than or equal to 6, we performed up to 6 loop unrolls in method treeInsert. The total number of lines of code checked, considering the inlined methods and the loop unrolls, is of approximately 230 lines. There are 16 branches to be covered. Following the technique we described in Section 5.1, the CFG thinner produced 9 subgraphs of the CFG. Looking for test inputs from these 9 CFGs allowed us to cover 15 out of the 16 branches, within a scope of 4. Actually, the remaining branch corresponds to an if statement where the if branch is always taken.

In Table 1 we present the analysis times of JAT using DynAlloy. SAT-solvers are usually very sensitive to increases in scope. Fortunately, test input data generation most of the times requires small structures to achieve a high coverage, and therefore SAT-solving becomes a viable technique. This hypothesis on the factibility of using small scopes is known as the *small scope hypothesis*. Different columns show the analysis time for different scopes. Running times were computed in a computer with a 64-bit AMD Athlon 3200 with 2 GB of RAM running on a dual channel architecture. Time is expressed in seconds.

Scope	3	4	5	6	7	8	9	10	11
	52	56	63	80	101	101	102	120	150

Tab. 1. Running times for the generation of test data.

5.3 Related Work

A vast amount of research on test input data generation has been done in the last few years. Some research, as is the case in the SLAM project [1] or in the case of the DART tool [5], assumes absence of aliasing. On the other hand, we aim at the analysis of programs that make extensive use of complex structures. Other research points toward specification testing. For instance, TestEra [13] uses the Alloy Analyzer for specification based testing. Tools such as CUTE [16], Symtra [18], or the work of Visser et al. [17] using Java Pathfinder, base their research on symbolic execution. We solely depend on SAT-solving for analysis purposes. An approach close to ours is the one followed in the INKA tool [7]. The tool handles complex data structures in C, but cannot handle dynamic allocation.

6 Conclusions

We have presented a novel formal method for the analysis of Java programs based on a translation of Java programs to DynAlloy, and the use of SAT-solvers. The experiments we have conducted show that JAT can be effectively used in the analysis of non trivial Java methods that create objects and handle complex data.

References

1. Ball. T, *A Theory of Predicate–Complete Test Coverage and Generation.* Technical Report MSR-TR-2004-28, Microsoft Research, Redmond, WA, April 20004.
2. Dijkstra E. W. and Scholten C. S., *Predicate calculus and program semantics*, Springer-Verlag, 1990.
3. Frias, M. F., Galeotti, J. P., Lopez Pombo, C. G., and Aguirre, N. M. *DynAlloy: Upgrading Alloy with actions.* In *Proceedings of the 27th. International Conference on Software Engineering,* G-C. Roman, Ed. Association for the Computer Machinery and IEEE Computer Society, ACM Press, St. Louis, Missouri, USA, 2005, 442–450.
4. Frias M.F., Lopez Pombo C.G., Baum G.A., Aguirre N.M. and Maibaum T.S.E., *Reasoning About Static and Dynamic Properties in Alloy: A Purely Relational Approach,* in ACM-Transactions on Software Engineering and Methodology (TOSEM), 14(4), 478 – 526, 2005.
5. Godefroid P., Klarlund N. and Sen K., *DART: Directed Automated Random Testing,* in Proceedings of the ACM SIGPLAN 2005 Conference on Programming Languages Design and Implementation (PLDI), 2005.
6. Goldberg, E. and Novikov, Y. *BerkMin: A fast and robust sat-solver.* In *Proceedings of the conference on Design, automation and test in Europe*, C. D. Kloos and J. da Franca, Eds. IEEE Computer Society, Paris, France, 142–149, 2000.
7. Gotlieb A., Denmat T. and Botella B., *Constraint-Based Test Data Generation in the Presence of Stack-Directed Pointers*, in Proceedings of ASE'05, Long Beach, CA, USA, ACM Press.
8. Harel D., Kozen D. and Tiuryn J., *Dynamic Logic*, MIT Press, October 2000.
9. Jackson, D. *Alloy: a lightweight object modeling notation.* ACM Transactions on Software Engineering and Methodology 11, 2, 2002, 256–290.
10. Jackson D., Schechter I. and Shlyakhter I., *Alcoa: the Alloy Constraint Analyzer*, Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 2000.

11. Jackson D. and Vaziri, M., *Finding Bugs with a Constraint Solver*, in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), August 21-24, 2000, Portland, OR, USA. ACM, 2000, pp. 14–25.
12. JUnit: http://www.junit.org.
13. Khurshid S. and Marinov D., *TestEra: Specification-Based Testing of Java Programs Using SAT.*, Automated Software Engineering 11(4): 403–434 (2004)
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby, *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*. TR 98-06-rev27, Iowa State University, Department of Computer Science, April 2005.
15. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. *Chaff: engineering an efficient SAT solver.* In *Proceedings of the 38th conference on Design automation*, J. Rabaey, Ed. ACM Press, Las Vegas, Nevada, United States, 2001, 530–535.
16. Sen K., Marinov D. and Agha G., *CUTE: A Concolic Unit Testing Engine for C*, in Proceedings of the ACM SIGSOFT Conference on Foundations of Software Engineering, Lisbon, Portugal, 2005.
17. Visser W., Pasareanu C., Khurshid S., *Test Input Generation with Java PathFinder*, in Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004. ACM 2004, pp. 97–107.
18. Xie T., Marinov D., Schulte D. and Notkin D., *Symtra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution*, in Proceedings of TACAS 2005.