DynAlloy Analyzer: A Tool for the Specification and Analysis of Alloy Models with Dynamic Behaviour

Germán Regis
Dept. of Computer Science,
University of Rio Cuarto, Argentina

César Cornejo
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Simón Gutiérrez Brida
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Mariano Politano
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Fernando Raverta
Digital Communications Lab,
University of Cordoba, Argentina

Pablo Ponzio
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Nazareno Aguirre
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Juan Pablo Galeotti
Dept. of Computer Science,
University of Buenos Aires, Argentina

Marcelo Frias
Dept. of Software Engineering,
Buenos Aires Institute of Technology,
Argentina

ABSTRACT

We describe DynAlloy Analyzer, a tool that extends Alloy Analyzer with support for *dynamic* elements in Alloy models. The tool builds upon Alloy Analyzer in a way that makes it fully compatible with Alloy models, and extends their syntax with a particular idiom, inspired in dynamic logic, for the description of dynamic behaviours, understood as sequences of states over standard Alloy models, in terms of *programs*. The syntax is broad enough to accommodate abstract dynamic behaviours, e.g., using nondeterministic choice and finite unbounded iteration, as well as more concrete ones, using standard sequential programming constructions. The analysis of DynAlloy models resorts to the analysis of Alloy models, through an optimized translation that often makes the analysis more efficient than that of typical ad-hoc constructions to capture dynamism in Alloy.

Tool screencast, binaries and further details available in:
`http://dc.exa.unrc.edu.ar/tools/dynalloy`

1 INTRODUCTION

Software models are an important part of most software development approaches. They come in many forms and concern different stages of software development, from requirements, where problem domain concepts relevant to the system as well as its goals need to be explicitly described [33], to design and implementation, where design concepts and implementation details, conveying decisions made during software construction, need to be (abstractly) captured [17]. Since models often concentrate on a particular aspect of the problem or system being described, they are easier for developers to grasp and more useful to communicate ideas, as well as for anticipating properties or concerns that arise, for instance, from design decisions or problem domain facts.

Formal models, i.e., models in a language with a formal syntax and precise semantics, are better suited for rigorous analysis than their informal counterparts, thanks to the fact that they have a precise meaning from which one can logically obtain conclusions [30]. Moreover, if a formal language is appropriately designed, its specifications can also be automatically analyzed, thus relieving their users from having to manually perform logical reasoning from specifications [34].

Alloy [21] is a popular formal specification language, that has been carefully designed to support automated analysis. Alloy features a simple syntax, with a few constructs with intuitive meaning, and a simple formal semantics, based on relations. Both the syntax and semantics are based on concepts that many developers are familiar with. This simplicity plays an important role in making the language's specifications automatically analyzable. Indeed, Alloy is supported by the Alloy Analyzer, a powerful analysis tool that allows one to search for instances of specifications as well as to check intended properties of models by resorting to SAT solving.

Alloy is a very expressive language, suitable for specifying a wide variety of *static* properties of systems, through formulas in *relational logic*, the logical formalism underlying Alloy, used to capture the intention of operations, assumed and intended properties of systems [20]. Alloy's expressiveness makes the analysis of specifications based on SAT solving necessarily incomplete: one may find counterexamples of intended properties and instances of specified models in *bounded* scenarios, but the absence of such counterexamples or instances does not imply their nonexistence [20]. While Alloy is expressive enough so that one may encode *dynamic* properties of systems, i.e., properties that predicate over system executions or successive state changes of some sort, models involving such kind of properties often become intricate by the inclusion of ad-hoc constructions to capture dynamism (cf. [21, 22]). This kind of ad-hoc characterizations of state change and dynamic behaviour is not standardized, so models requiring such kind of construction may significantly differ from one another, reducing model understandability. Moreover, characterizations of dynamic behaviour over Alloy specifications have a significant impact in analyzability, in many cases causing serious scalability issues that call for sophisticated optimizations that reduce readability even further [10–12].

The DynAlloy language, originally introduced in [11] and further developed in [10, 12, 13], deals precisely with the above described issue. DynAlloy extends Alloy's syntax with a particular idiom, inspired in dynamic logic [18], for the description of dynamic behaviours, understood as sequences of states over standard Alloy models, in terms of *programs*. The DynAlloy syntax is broad enough to accommodate abstract dynamic behaviours, e.g., using nondeterministic choice and finite unbounded iteration, as well as more concrete ones, using standard sequential programming constructions. The analysis of DynAlloy models resorts to an analysis of Alloy models, through a optimized translation that often makes the analysis more efficient than that of typical ad-hoc constructions to capture dynamism in Alloy [10, 12].

In this tool demonstration paper we describe DynAlloy Analyzer, a tool that extends the highly regarded Alloy Analyzer with support for *dynamic* elements in Alloy models. The tool builds upon Alloy Analyzer in a way that makes it fully compatible with Alloy models, and extends the model's syntax with elements for describing *actions* (atomic state changes), abstract programs over these actions, and correctness assertions. Programs can be run and partial correctness assertions involving programs can be checked in DynAlloy Analyzer, in the same way that predicates can be run and assertions can be checked in Alloy Analyzer. DynAlloy Analyzer is a complete redevelopment of the original DynAlloy tool; while the original tool was implemented as a separate compiler that translated DynAlloy specifications into Alloy ones, and did not include commands for analysis in the notation, the new tool is integrated into Alloy Analyzer, enabling a fully transparent usage for Alloy users, and providing analysis commands that better reflect the style found in Alloy specifications.

2 ALLOY MODELS WITH STATE CHANGE

Alloy is a *model-oriented specification language*, similar to other formal languages such as Z [31], VDM [23] and B [2]. As for these other languages, specifications (or models, as these are more often called) are written by defining data domains, properties and operations between these domains. Data domains are defined in Alloy via *signatures*. Signatures represent sets of *atoms*, and can be extended, with signature extension representing set containment. A signature is called *abstract* if it does not have proper elements, but all its elements are those of its extending signatures. Alloy does not allow one to define *atoms* in its models; these can be captured instead through singleton signatures. Consider the following signature definitions taken from the *Farmer* model (a model of the well-known puzzle that asks whether it is possible for a farmer to cross a river with a chicken, a sack of grain and a fox, with a boat that can hold the farmer and at most one element, and avoiding that the fox eats the chicken or the chicken the grain, if they become unsupervised):

```
abstract sig Object {
    eats: set Object
}

one sig Farmer, Fox, Chicken, Grain extends Object { }

fact { eats = Fox->Chicken + Chicken->Grain }
```

Data domain `Object` is composed of exactly four "elements" (singletons), `Farmer`, `Fox`, `Chicken` and `Grain` (signatures that extend a same signature are disjoint). As the definition of `Object` shows, signatures can have *fields*, that represent relations; in the case of field `eats`, it denotes a relation between objects, indicating what objects each object eats.

We would like to refer to a state changing situation, that maintains two sets of objects, those on the *near* and *far* sides of the river, respectively. The usual Alloy way of capturing this situation is via an additional signature for states, and using total orders (from a library specification) of states to capture executions, as follows:

```
open util/ordering[State]

sig State {
    near, far: set Object
}
```

We then need to impose restrictions, via Alloy *facts* (assumed properties of the model), to capture that in the initial state all objects are in the near side of the river, and how transitions are governed (unsupervised objects eat other objects, farmer can only take one other item with him, etc.).

```
fact { first.near = Object && no first.far }

pred crossRiver[from, from', to, to': set Object] {
    one x: from | {
        from' = from - x - Farmer - from'.eats
        to' = to + x + Farmer
    }
}

fact {
    all s: State, s': s.next | {
    Farmer in s.near =>
        crossRiver[s.near, s'.near, s.far, s'.far]
        else crossRiver[s.far, s'.far, s.near, s'.near] }
}
```

Finally, the puzzle is attempted to be solved by trying to get a trace in which, in its last state, all objects reached the far side of the river:

```
run { last.far = Object } for exactly 8 State
```

3 STATE CHANGE: THE DYNALLOY APPROACH

Let us describe how the *Farmer* model would be modeled in DynAlloy. DynAlloy is an extension of Alloy for better describing state change; the part of the model that does not deal with state change is done exactly as in the case of Alloy: we will maintain the definition of `Object` and composing signatures, and the facts constraining them. But, we will not use signature `State`, nor linear orderings. Instead, we will use *actions* to describe state change. We will first use *atomic actions*, to indicate basic state change steps, and then *programs* to get the same traces got in the Alloy approach.

Atomic Actions. Atomic actions are the basic building block for describing state change. An atomic action is specified by indicating what it applies to (the *parameters*, what the action changes), the required conditions to execute the action (the *enabling condition*, that we call *precondition*), and the effect of the action (the *postcondition*). As a first example of an atomic action, consider the following:

```
act crossRiver[from, to: set Object] {
  pre { Farmer in from }
  post { one x: from |
          from' = from - (x + Farmer) - from'.eats &&
          to' = to + (x + Farmer)
       }
}
```

In a DynAlloy action, the parameters define the state the action changes, in this case, the formal parameters `from` and `to`. There is no need to define a special signature to represent the state, since the *state* is implicitly defined as the parameters of the action. Moreover, it is indicated explicitly what is expected for the action to be applicable, namely that the farmer must be in `from`; it is important to remark that action preconditions are *enabling conditions*: when they are not satisfied, the action cannot be executed. Finally, the primed versions of the parameters of the action are *not* parameters themselves: they refer to the state of these parameters after the action has been executed.

Composite Actions (programs). While in Alloy characterizations of state change, executions are described via facts that explicitly indicate how state changes, in DynAlloy these are more conveniently defined by the construction of actions. Atomic actions are the base case for describing more complex behaviours, in *composite actions* or *programs*. The DynAlloy version of what is captured via orderings in the Alloy approach, is shown below. It features various program constructs: *assumptions*, *test actions*, *nondeterministic choice*, *sequential composition*, and *iteration*:

```
program solvePuzzle[near, far: set Object] {
    assume (Object in near && no far);
    (crossRiver[near, far] + crossRiver[far, near])*;
    [Object in far]?
}
```

Again, it is important to remark that this program's state is given by its formal parameters, the sets of object `near` and `far` (no need for a special signature representing the state). The program is composed of the sequential composition of three parts. The first is an assumption of what is expected at the beginning (i.e., an enabling condition given as part of the code). It indicates that it is assumed that initially all objects are on the near side of the river. The second is a finite iteration of a nondeterministic choice of two actions; these actions correspond to the two possibilities: either the river is crossed from `near` to `far` or is crossed from `far` to `near`. And these are iterated (in each iteration, any of the enabled actions might be taken, although for this case we know that exactly one of them will be enabled) a finite number of times. Finally, the execution is allowed to continue only if, after the iteration, all the objects got to the far side of the river.

DynAlloy features other mechanisms for building composite actions, although these can all be reduced to atomic actions, sequential composition, tests/assumptions and iteration. They are however very useful for more conveniently capturing some state changing behaviours. For example, atomic action `crossRiver` can be redefined in terms of a more concrete program, as follows:

```
act choose[x: univ, s: set univ] {
    pre { some s }
    post{ x' in s }
}

prog refCrossRiver[from, to: set Object] var [x: Object] {
    assume (Farmer in from);
    choose[x, from];
    from := from - (Farmer + x);
    from := from - from.eats;
    to := to + (Farmer + x)
}
```

This program has the same effect as the atomic action `crossRiver`, although not in a single, atomic step. It performs the river crossing by nondeterministically choosing an object in the `from` side, and then updating the state of `from` and `to` via *atomic assignments*. Program `solvePuzzle` can simply call program `refCrossRiver` instead of the atomic action `crossRiver`.

3.1 Analysis of DynAlloy Models

Alloy provides two ways of analyzing a specification: via *runs*, which search for satisfying models of a given predicate, and via *asserts*, which search for counterexamples of intended properties of a specification. DynAlloy provides similar features, but targeting the analysis of dynamic properties. These are *program runs*, and *partial correctness assertions*. A program run is very much like a run of a predicate. A run of a program will search for satisfying models of the program, i.e., executions of the program. A partial correctness assertion is, as in the context of program verification [8, 19], an expression of the form: { *pre* } *prog* { *post* }, where *prog* is a program, and *pre* and *post* are formulas over the state of the program (the precondition and postcondition, respectively). Such an assertion is true if and only if, for every execution of the program *prog*, if the execution starts in a state satisfying *pre*, then if the execution terminates it must do so in a state satisfying *post*. As examples of partial correctness assertions, consider the following. The first states that objects cannot be on both sides of the river at the same time (`noQuantumObjects`, an assertion also present in the original Alloy model); the second is an assertion that states that, once an object is lost in an execution, it cannot be resurrected.

```
assert NoQuantumObjects [near, far: set Object] {
 pre { no (near & far) }
 prog { (crossRiver[near, far] + crossRiver[far, near])* }
 post { no (near' & far') }
}

assert noResurrection[near, far: set Object, x: Object] {
 pre { Object in near && no far }
 prog { (crossRiver[near, far] + crossRiver[far, near])*;
        [x !in (near+far)]? ;
        (crossRiver[near, far] + crossRiver[far, near])*
      }
 post { x !in (near'+far') }
}
```

The analysis mechanism behind `Alloy Analyzer` is SAT based bounded verification (or SAT based bounded model finding). Alloy Analyzer employs user provided bounds, the scope, in order to exhaustively search for counterexamples of intended properties, within the provided bounds. DynAlloy's main analysis mechanism resorts to Alloy's analysis: DynAlloy Analyzer will use Alloy Analyzer "behind the scenes", in order to check a partial correctness assertion, or run a program. But DynAlloy models have an additional source of potential unboundedness: program iteration. So, the user will have to provide an extra bound, to indicate the maximum number of iterations to be considered in programs (also called loop unrolls). This is done as part of the *commands* for program runs and partial correctness assertions, as in the following examples (`lurs` stands for loop unrolls):

```
run solvePuzzle for 4 lurs 7
check noResurrection for 4 lurs 8
```

To automatically analyze DynAlloy specifications using Alloy Analyzer, we translate annotated DynAlloy programs into Alloy specifications. This is realized through a *bounded* version of *weakest liberal precondition* [8], as described in [10]. This predicate transformer allows us, given a bound n in the number of loop unrolls, to transform a program into an Alloy predicate. This translation often leads to better performance compared to the traditional Alloy approach to state change, as shown in [10] and with further examples in the DynAlloy website. For atomic actions, it is straightforward; more complex programs are translated as follows:

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

4 RELATED WORK

The need to capture dynamic behavioural properties over Alloy specifications is present in a wide number of applications. Various tools capture program semantics in Alloy, notably Forge [6, 7] and TACO [14, 15]. The latter uses DynAlloy as an intermediate language for reducing program verification to SAT solving, as well as some related tools do [1, 28, 29]. These tools also significantly exploit *relational bounds* as introduced through KodKod [32], and have seen a dramatical increase in performance thanks to KodKod being current Alloy's model finding engine. There have also been proposals to extend Alloy with dynamic behaviour constructs for more abstract modeling. Besides DynAlloy [10], other extensions such as Imperative Alloy [27] and Electrum [26], have been used to complement Alloy with rich logical languages for dynamic behaviour, e.g., linear temporal logic as realized in TLA+ [25], and use these to specify and check properties in a bounded manner. Applications of Alloy models with dynamic behaviour appear in the context of dynamic software architecture [4, 5], dynamic access control policies [9], and the analysis of specifications originating in informal languages such as the UML. Many direct uses of Alloy also require dynamic behaviours, as is shown, e.g., in examples and case studies in [3, 7, 21, 22].

5 CONCLUSIONS

The importance of powerful and efficient automated analysis to accompany formal specification languages is widely acknowledged, and the success of Alloy has been in great part due to its emphasis in automated analysis. The significant advances in SAT solving, as well as in improved encodings of analysis problems from different contexts into SAT (indirectly) through Alloy and similar languages, maintains the language being relevant in various areas of software engineering (cf., e.g., [3, 7, 15, 16, 24]). The need for *dynamic* elements in Alloy models arises naturally in many of these contexts, which is evidenced by the various approaches that have emerged, to conveniently capture dynamic behaviours. This tool demonstration paper presented a full redevelopment of DynAlloy Analyzer, a tool that incorporates one of these approaches into Alloy Analyzer. This implementation is faithful to the style of the Alloy tool, e.g., in the way commands are issued, and how programs and partial correctness assertions are introduced. The tool is fully compatible with standard Alloy, produces detailed compile-time error reports, and features a mature encoding of dynamic behaviour into Alloy [13] that has proved to be more efficient than many similar ad-hoc Alloy constructions for capturing dynamism.

REFERENCES

[1] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. Improving test generation under rich contracts by tight bounds and incremental SAT solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 21–30. IEEE Computer Society, 2013.

[2] J.-R. Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[3] Hamid Bagheri and Sam Malek. Titanium: efficient analysis of evolving alloy specifications. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 27–38. ACM, 2016.

[4] Roberto Bruni, Antonio Bucchiarone, Stefania Gnesi, Dan Hirsch, and Alberto Lluch-Lafuente. Graph-based design and analysis of dynamic software architectures. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2008.

[5] Antonio Bucchiarone and Juan P. Galeotti. Dynamic software architectures verification using dynalloy. *ECEASST*, 10, 2008.

[6] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with sat. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 109–120, New York, NY, USA, 2006. ACM.

[7] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

[8] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[9] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[10] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. Dynalloy: Upgrading alloy with actions. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 442–451, New York, NY, USA, 2005. ACM.

[11] Marcelo F. Frias, Carlos López Pombo, Gabriel Baum, Nazareno Aguirre, and T. S. E. Maibaum. Taking *Alloy* to the movies. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 678–697. Springer, 2003.

[12] Marcelo F. Frias, Carlos López Pombo, Gabriel A. Baum, Nazareno Aguirre, and T. S. E. Maibaum. Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Trans. Softw. Eng. Methodol.*, 14(4):478–526, 2005.

[13] Marcelo F. Frias, Carlos López Pombo, Juan P. Galeotti, and Nazareno Aguirre. Efficient analysis of dynalloy specifications. *ACM Trans. Softw. Eng. Methodol.*, 17(1):4:1–4:34, 2007.

[14] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.

[15] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 25–36. ACM, 2010.

[16] Jaco Geldenhuys, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. Bounded lazy initialization. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2013.

[17] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[18] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic.* MIT Press, Cambridge, MA, USA, 2000.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[20] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[21] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[22] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In A. Min Tjoa and Volker Gruhn, editors, *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 62–73. ACM, 2001.

[23] Cliff B. Jones. *Systematic Software Development Using VDM (2Nd Ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[24] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Testera: A tool for testing java programs using alloy specifications. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 608–611. IEEE Computer Society, 2011.

[25] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[26] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 373–383, New York, NY, USA, 2016. ACM.

[27] Joseph P. Near and Daniel Jackson. An imperative extension to alloy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*, pages 118–131. Springer, 2010.

[28] Bruno Cuervo Parrino, Juan Pablo Galeotti, Diego Garbervetsky, and Marcelo F. Frias. Tacoflow: optimizing SAT program verification using dataflow analysis. *Software and System Modeling*, 14(1):45–63, 2015.

[29] Nicolás Rosner, Jaco Geldenhuys, Nazareno Aguirre, Willem Visser, and Marcelo F. Frias. BLISS: improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Trans. Software Eng.*, 41(7):639–660, 2015.

[30] Hossein Saiedian. An invitation to formal methods. *Computer*, 29(4):16–17, April 1996.

[31] J. Michael Spivey. *Z Notation - a reference manual (2. ed.).* Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[32] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[33] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley Publishing, 1st edition, 2009.

[34] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.