The DynAlloy Visualizer

Pablo Bendersky
Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
pbendersky@gmail.com

Juan Pablo Galeotti
Saarland University
Saarbrücken, Germany
galeotti@cs.uni-saarland.de

Diego Garbervetsky
Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
diegog@dc.uba.ar

We present an extension to the DynAlloy tool to navigate DynAlloy counterexamples: the DynAlloy Visualizer. The user interface mimics the functionality of a programming language debugger. Without this tool, a DynAlloy user is forced to deal with the internals of the Alloy intermediate representation in order to debug a flaw in her model.

1 Introduction

Alloy [6] is a formal specification language, which belongs to the class of the so-called model-oriented formal methods. Among the key features that make Alloy an appealing modeling language to a wide community of users is its simple syntax, object oriented semantics and more importantly, its analyzability. Alloy models can be automatically analyzed by simply declaring a bound on the number of elements for each domain. We refer to this bound as the *scope* of the analysis. Then, the Alloy model is translated into a propositional formula. In turn, an off-the-shelf SAT-Solver is invoked to decide the validity of the given SAT-problem. If the formula is satisfiable, then the semantic preserving translation ensures the original Alloy model has a counterexample. Analogously, if the formula is unsatisfiable, the translation ensures the Alloy model has no counterexamples, but *within* the scope of the analysis previously selected by the user. Both the Alloy tool and source code are publicly available for download[1].

DynAlloy [3] is an efficient extension of the Alloy language with procedural actions. A DynAlloy user is able to declare atomic actions in terms of pre and postconditions using standard Alloy predicates. Composite actions (namely programs) can be declared by combining these atomic actions and other simpler composite actions even further. The syntax for composite actions is given in Figure 1. A user can assert that if a given precondition holds, and a certain program execution ends, then the program execution led to a particular postcondition. In other words, since there is no guarantee of program termination, the semantics of DynAlloy assertions is that of partial correctness assertions.

DynAlloy was devised with the explicit goal of allowing users to automatically analyze the written models. In order to do so, the DynAlloy tool performs a semantic preserving transformation from a DynAlloy model into an Alloy model. A high level description of DynAlloy's architecture is given in Figure 2. It is worth noticing that the *iteration* composite action might lead to an infinite Alloy model for the general case. To avoid this, DynAlloy restricts the analysis to traces of a fixed-length only. As with the object domains, the user has to select this maximum fixed-length. This is done by limiting the number of repetitions the iteration body might be executed.

[1] http://alloy.mit.edu/alloy/download.html

Nazareno Aguirre, Leila Ribeiro (Eds.):
First Latin American Workshop on Formal Methods (LAFM 2013)
EPTCS 139, 2014, pp. 59–64, doi:10.4204/EPTCS.139.6

$$
\begin{array}{rcll}
program & ::= & \langle formula, formula \rangle (\bar{x}) & \text{“atomic action”} \\
& | & formula? & \text{“test”} \\
& | & program + program & \text{“non-deterministic choice”} \\
& | & program ; program & \text{“sequential composition”} \\
& | & program^* & \text{“iteration”} \\
& | & \langle program \rangle (\bar{x}) & \text{“invoke program”}
\end{array}
$$

Figure 1: Grammar for composite actions in DynAlloy

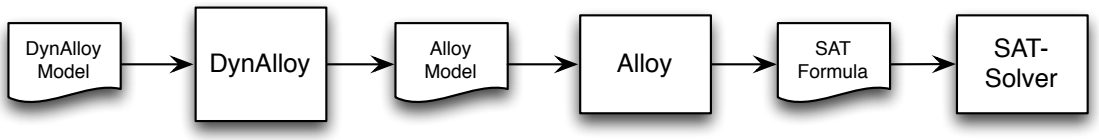

Figure 2: A high-level view of the DynAlloy architecture

If we restrict ourselves to the architecture depicted in Figure 2, a user obtains a rather binary result after analyzing a DynAlloy model. Such answer might be expected when the given assertion is valid (always within the scope of analysis and the length of the traces under scrutiny). Nevertheless, if the assertion is invalid, the user is unable to understand in more depth how the intended model differs from the written one. In other words, the user is unable to *debug* the DynAlloy model.

In contrast, the Alloy tool includes a graphical user interface for displaying the counterexample found in terms of the relational semantics of the Alloy language. This means that Alloy maps back the solution (namely the satisfying assignment) found by the SAT-solver from the propositional level, to the Alloy level. Although exploring the Alloy counterexample appears as a more suitable solution than simply obtaining an error message, the user has to fully understand the internals of the translation of the model from DynAlloy to Alloy.

In this work we present an extension to the DynAlloy tool that allows users to navigate a counterexample in terms of the DynAlloy semantics rather than the Alloy semantics. The rest of the article is organized as follows. In Section 2 we present a motivational example. In Section 3 we describe our tool for visualizing DynAlloy counterexamples, as well as some implementation details. Finally, in Section 4 we conclude and discuss related work.

2 A Motivational Example

In order to illustrate the reader, let us a consider a simple example for specifying operations in singly linked lists. As we already stated, DynAlloy is an extension of the Alloy language. So, the user specifies different domains (namely, disjoint set of atoms) by declaring Alloy *signatures*. In the example we declare a singleton domain *null* to represent an empty pointer. Next, we declare two signatures, one for lists and the other for nodes contained in the lists:

```
one sig null {}
sig List {}
```

```
sig Node {}
```

Each List element points to a Node element (or the null value) by means of field *header*. Similarly, a Node element points to the next Node element (or the null value) through field *next*. The count of elements within the list is stored using field *size*. In order to allow updating the value of fields, *head*, *size* and *next* fields are declared as functional variables instead of regular Alloy fields (whose semantics is static). In the program body, the $a \to d$ denotes the ordered pair $\langle a, d \rangle$, and $++$ denotes the relational override between a functional relation and an ordered pair. The effect of such operator is to replace the previous value stored for a with d in the new value of the functional relation.

```
program removeLast[thiz: List, header: List -> one(Node+null),
                   next: Node -> one(Node+null), size: List -> one Int]
var [curr: Node+null, prev: Node+null, newSize: Int]  {
  prev := null;
  curr := thiz.header;
  while isNotNull[curr] do {
    prev := curr; curr := curr.next
  };
  if isNotNull[prev] {
    header := header ++ (thiz->null);
    newSize := sub[thiz.size,1];
    size := size ++ (thiz->newSize)
  } else { skip }
}
```

As the reader may have noticed, `if` and `while` constructs are not valid productions in the grammar introduced in Figure 1. It is worth noticing that these more complex programming structures can be described using these basic logical constructs. More specifically, `if B {P} else {Q}` can be rewritten as the following DynAlloy program $B?;P + (\neg B)?;Q$. Similarly, `while B do {P}` can be expressed as $(B?;P)*;(:B)?$.

Finally, the user declares a property of interest and the corresponding DynAlloy assertion. In this case, she wants to state that the value stored in the *size* field of the list agrees with the actual number of Node elements that are reachable from the *header* field traversing the *next* field. Such property can be compactly expressed by means of the Alloy reflexive transitive closure operator:

```
thiz.size=#(thiz.header.*next-null)
```

For checking the correctness of the DynAlloy assertion, let us assume the user chooses the default scope of analysis (each signature can contain at most 3 atoms and iterations can be exercised at most 3 times). Not very surprisingly, the analysis reports the assertion does not hold. In this scenario the only source for understanding the problem (apart from the DynAlloy model itself) is the Alloy counterexample reported by the Alloy tool (Figure 3). As the reader may notice, it is rather cumbersome to understand exactly what part of the DynAlloy model is flawed. This is due to the fact that the counterexample is shown in terms of the *internal* representation of the DynAlloy model into an Alloy model that the DynAlloy tool applies.

3 Debugging a DynAlloy model

In this work we present an extension to the DynAlloy tool that allows users to navigate a DynAlloy counterexample, the DynAlloy Visualizer. The visualizer extension performs the following tasks:

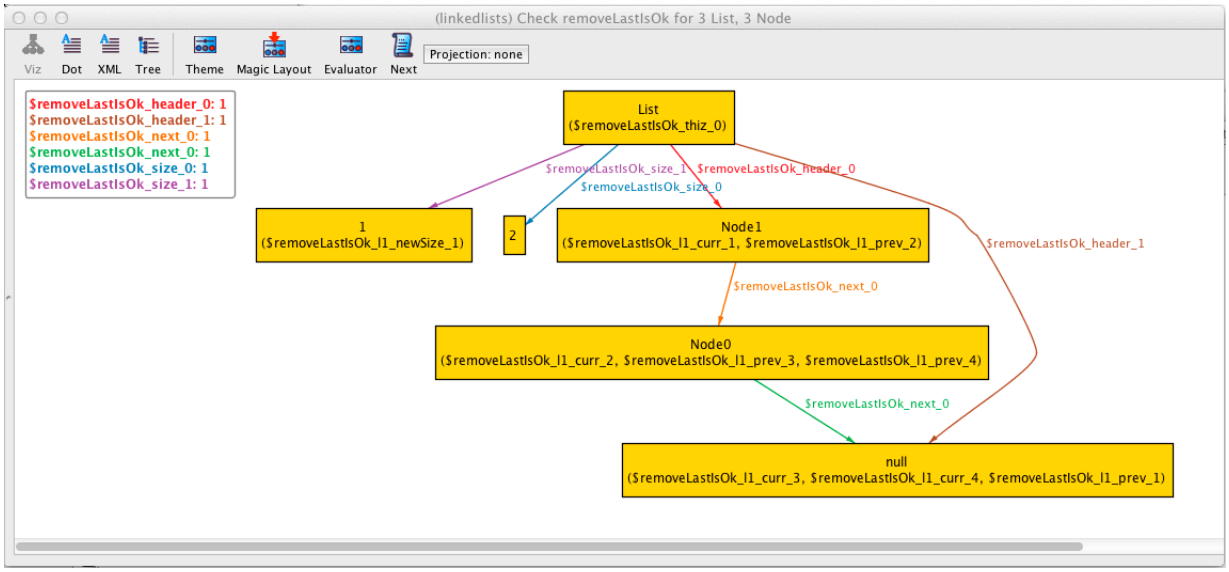

Figure 3: The Alloy counterexample displayed using the standard visualization

- DynAlloy models are translated to Alloy models as presented in [3], but preserving a mapping from DynAlloy to Alloy.
- If a counterexample is found, the counterexample is lifted from the Alloy domain back into the DynAlloy realm using the collected mapping.

Figure 4 shows the graphical-user interface of the DynAlloy Visualizer. This interface allows the user to navigate the trace and inspect user-defined expressions and field values. Users can edit the DynAlloy model using the left pane. Once editing is finished, she can launch a new analysis by hitting the *Execute* menu option. In case a counterexample is found, the trace is presented on the lower right pane (Figure 5). The counterexample trace is presented in a tree form where each node represents an action taken (an atomic action, an assignment, an *if* decision, etc.) and parent nodes represent DynAlloy program calls.

On the upper right pane the values of selected variables, fields and user-defined Alloy expressions are displayed. If the user navigates the counterexample trace (by going back and forth in the steps of the trace) these values are automatically updated to display their values at the current step.

In the example, the user can notice that although the size of the list correctly decreases from 3 to 2, the resulting list has no elements since the *header* field points to null. In other words, instead of setting the *header* field to null, what the program has to do at this point is set the value of `prev.next` to null. Once this change to the model is done, the new analysis returns no counterexample.

Implementation Details

As already mentioned, while translating a DynAlloy model to the Alloy language, our tool builds a mapping from the source DynAlloy programs into the target Alloy formulas. By using a bidirectional mapping we are also able to efficiently find the DynAlloy program for a given Alloy formula. Luckily, the Alloy tool provides a programmatic interface for inspecting a given counterexample. Each counterexample found leads to a fresh `A4Solution` instance with all the necessary information. Following the translation presented in [3], a disjunction in the target Alloy formula is introduced only in the case

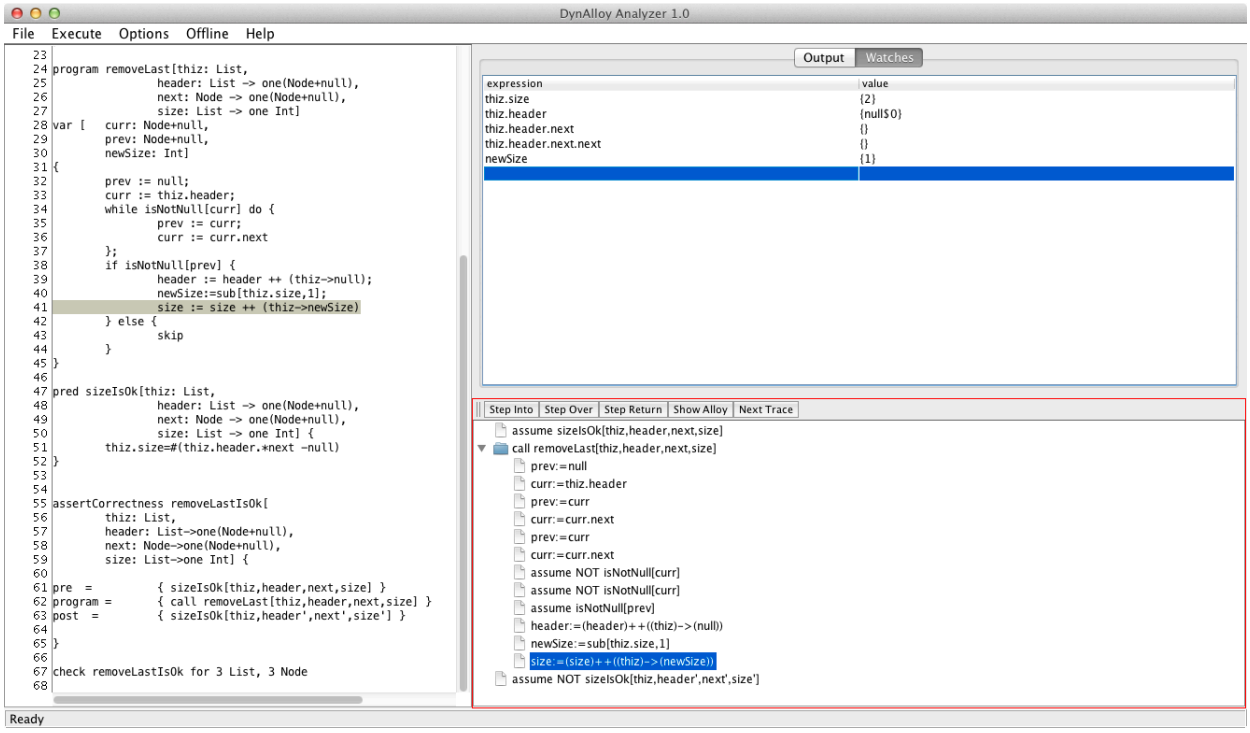

Figure 4: The graphical-user interface of the DynAlloy Visualizer. The left pane allows the user to edit changes on the DynAlloy model. The right panes display the counterexample (if found). The lower right pane shows the trace while the upper right pane displays the values of the selected expressions at the current step of the trace.

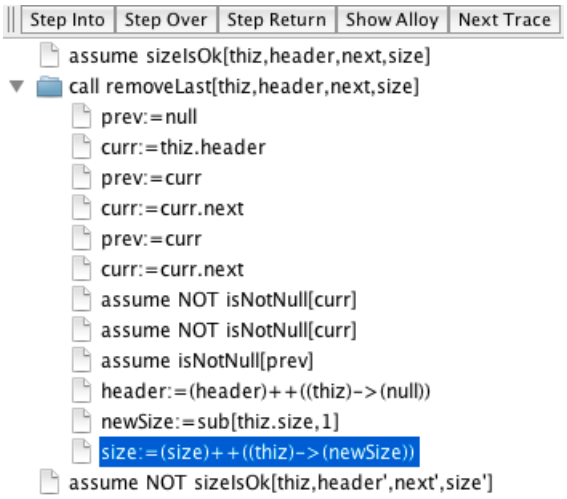

Figure 5: The DynAlloy trace in detail. In the tree form view each node represents an action taken and parent nodes represent DynAlloy program calls. The DynAlloy user is able to go backward and forward in the trace by simply clicking the new step.

of the non-deterministic choice $+$. Given the abstract syntax tree of the resulting Alloy formula, our tool evaluates each sub formula. If the sub formula evaluates to true, then the original DynAlloy program is obtained, and the procedure is recursively applied on each component of the sub formula. We implemented this algorithm generically by instantiating the *Visitor* design pattern.

A useful feature of Alloy Analyzer is the unsat core highlighting. This feature allows a user to see a (possibly minimal) subset of the model constraints from which the assertion follows. Although the current version of our prototype does not offer such highlighting, the current DynAlloy architecture is suitable for supporting this feature in a future release.

4 Conclusions and Related Work

In this work we have presented an extension to the DynAlloy tool for visualizing DynAlloy counterexamples. We have also shown through a motivational example how this extension might help a user to debug her model without dealing with any intermediate representation.

Among many other reasons, the success of Alloy as a lightweight formal method was based in helping its users to understand the counterexample provided by its back-end (in this case, an off-the-shelf SAT-Solver). Many other tools follow this line. The Boogie Verification Debugger [5] provides plugins for visualizing counterexamples for VCC and Dafny users. JForge [2] is a tool for Java bounded verification based on the Alloy language. The JForge plug-in allows the user to visualize the offending program trace, but described in an intermediate representation language. TacoPlug [1] is an eclipse plugin for the bounded verifier TACO [4]. TacoPlug provides many debugging features such as heap memory graph visualization and Java trace navigation. Since TACO uses DynAlloy as a backend, TacoPlug is a client of the Visualizer's API.

Our tool is implemented as an extension of the DynAlloy tool, which has been released as open source to the community. For more information on DynAlloy, please visit the website: `http://www.dc.uba.ar/dynalloy`

Acknowledgments

This work has been partially funded by CONICET, UBACyT-20020110200075/20020100100813, MinCyT PICT-2010-235/2011-1774/2012-0724, CONICET-PIP 11220110100596CO, MINCYT-BMWF AU/10/19, INRIA Associated Team ANCOME, and LIA INFINIS and MEALS 295261.

References

[1] M. Chicote & J.P. Galeotti (2012): *TacoPlug: An Eclipse plug-in for TACO*. In: *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pp. 37–42, doi:10.1109/TOPI.2012.6229808.

[2] G. Dennis (2009): *A Relational Framework for Bounded Program Verification*. Ph.D. thesis, MIT.

[3] M. F. Frias, J. P. Galeotti, C. López Pombo & N. Aguirre (2005): *DynAlloy: upgrading alloy with actions*. In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, ACM, pp. 442–451, doi:10.1145/1062455.1062535.

[4] J. P. Galeotti, N. Rosner, C. López Pombo & M. F. Frias (2010): *Analysis of invariants for efficient bounded verification*. In: *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, ACM, pp. 25–36, doi:10.1145/1831708.1831712.

[5] C. Le Goues, K. R. M. Leino & M. Moskal (2011): *The Boogie Verification Debugger (Tool Paper)*. In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, *LNCS* 7041, pp. 407–414, doi:10.1007/978-3-642-24690-6_28.

[6] D. Jackson (2006): *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.