The Boogie Verification Debugger (Tool Paper)

Claire Le Goues[0], K. Rustan M. Leino[1], and Michał Moskal[1]

[0] University of Virginia, Charlottesville, VA, USA
legoues@cs.virginia.edu
[1] Microsoft Research, Redmond, WA, USA
{leino,micmo}@microsoft.com

Abstract. The Boogie Verification Debugger (BVD) is a tool that lets users explore the potential program errors reported by a deductive program verifier. The user interface is like that of a dynamic debugger, but the debugging happens statically without executing the program. BVD integrates with the program-verification engine Boogie. Just as Boogie supports multiple language front-ends, BVD can work with those front-ends through a plug-in architecture. BVD plug-ins have been implemented for two state-of-the-art verifiers, VCC and Dafny.

0 Introduction

Deductive software verification technology is sufficiently mature that tools can formally verify non-trivial programs written in semantically intricate modern languages. However, these tools remain difficult to use. They require considerable expertise, patience, and trial-and-error, especially to decipher error conditions and verification failures. In sum: our tools can understand our programs, but we cannot understand our tools.

In this paper, we present a *verification debugger*, called BVD (Boogie Verification Debugger), to help users understand the output of a program verifier. Our tool advances the state-of-the-art in program verification by increasing the communication bandwidth between the verifier and the user. Much as a dynamic debugger allows a programmer to explore a failing run-time state, the verification debugger allows her to explore—and, by extension, understand—a failing verification state. For example, a user can inspect the assumed values of local and heap-allocated variables. Constructing such a debugger is challenging because of the disconnect between the theorem prover counterexample model and the programmer's mental model of program execution.

Verification tools vary in their automation levels. At one extreme lies fully automatic verifiers. This class of verifier includes many abstract interpreters or model checkers; to obtain full automation, such tools typically limit their reasoning to certain domains. At the other extreme lies verifiers that accept user input at all proof steps. This class of verifier describes many mechanical proof assistants; to obtain this flexibility, these tools typically expose the user to the internal representation of proof obligations. In this paper, we consider a family of verifiers between the automatic and interactive extremes, which we refer to as *auto-active verification*. An auto-active verifier has two major characteristics: the user can define assertions for the verifier to prove, and all user interaction is done at the level of the program source language. For example, a user may help the verifier along by declaring a precondition or loop invariant in the program, but

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 407–414, 2011.

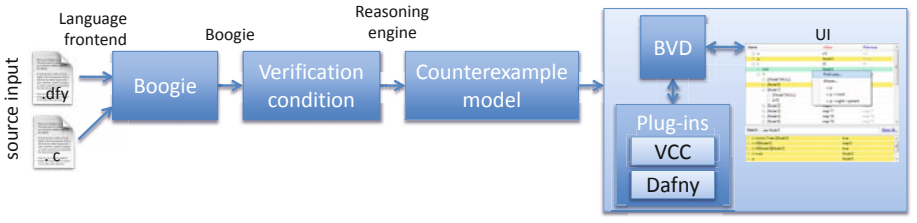

Fig. 0. BVD in the context of a failed verification in the Boogie system. Input source is compiled to Boogie, which Boogie translates into a verification condition (VC). When the reasoning engine cannot discharge the VC, it produces a counterexample model. BVD interacts with a language-specific plug-in to interpret and display the counterexample model to the user.

never interacts directly with the underlying reasoning engine. Examples of auto-active verifiers include extended static checkers [4] and program verifiers like Dafny [8] and VCC [2].

A prevalent implementation technique for auto-active verifiers is to translate the source program and its user annotations into an *intermediate verification language*, like Boogie [1] or Why [5]. This intermediate program is used to generate a verification condition; this condition is passed to a reasoning engine, for example a Satisfiability Modulo Theories (SMT) solver like Z3 [3]. Intermediate verification languages like Boogie confer similar benefits to verification as intermediate program representations do to compilation. In particular, they allow different source-language front-ends to share a verifier back-end. BVD, presented in this paper, works with Boogie and provides a plug-in architecture to support different front-ends. We have built BVD plug-ins for VCC [2] and Dafny [8].[0]

The rest of this short paper is organized as follows. Section 1 situates BVD in the context of the architecture of the Boogie pipeline. Section 2 describes the features, implementation, and use of BVD on indicative examples, and provides screenshots. Section 3 presents related work, while Sect. 4 discusses future work and concludes.

1 High-Level architecture

Fig. 0 provides a high-level overview of the architecture of BVD as it fits in the pipeline of a failed verification in Boogie. The Boogie verification system supports multiple language front-ends. Input source code is transformed by a language-specific front-end into the Boogie intermediate verification language. The Boogie tool then transforms the intermediate program to generate a verification condition (VC) to pass to a reasoning engine. When verification fails, the underlying reasoning engine produces a counterexample model under which the postcondition does not hold. Unfortunately, this counterexample model, consisting of a list of elements and interpretations of functions used in the language encoding, does not map transparently back to the source code. BVD

[0] Boogie, BVD, and Dafny are available as open-source projects at `http://boogie.codeplex.com/`. VCC is available with source at `http://vcc.codeplex.com/`.

interacts with a language plug-in to interpret the counterexample model and display it in a way meaningful to the end-user, and not only the verifier author.

This paper is primarily concerned with the boxed region of the diagram, which corresponds to the BVD tool. BVD support requires minor modification to the front-end compiler, described below; such modifications are typically negligible in practice.

2 BVD

BVD is geared towards the debugging of failed verifications of imperative languages, characterized by sequential execution states. Typically, regular debuggers display one execution state at a time, through which one may step forward, and in some cases backward (e.g., the OCaml debugger or IntelliTrace in VS2010). Similarly, an SMT counterexample model consists of *states* leading to a failed assertion. Each state ultimately corresponds to a location in the source code, and encodes relevant memory contents at those source locations. BVD translates the SMT state sequences into "execution" states and memory values that the user can understand, and displays them in an informative way. This section is concerned with the details of this translation process.

2.0 Insert example

To make the discussion more concrete, consider the following Dafny program implementing a linked list with insertion. The programmer has annotated the method to provide information to the verifier (line 2) and to tell the verifier what it should prove (line 4, which asserts that **n** has been successfully inserted):

```
0  class ListNode { var data: int; var next: ListNode; }
1  static method Insert(hd: ListNode, n: ListNode)
2    requires hd != null ∧ n != null;
3    modifies hd, n;
4    ensures n.next == old(hd.next);
5  { n.next := hd.next; hd.next := n; }
```

This method fails to verify because it is possible for **hd** and **n** to be aliases on entrance to **Insert**, leading both **hd->next** and **n->next** to point to **n** after line 5. This violates the postcondition. This can be addressed by adding a precondition asserting **hd != n** on method entrance.

However, the verifier provides very little information to help the programmer understand this failure. Its error message states that the postcondition is violated, and that line 5 is implicated. The underlying reasoning engine's counterexample model is similarly unhelpful. A Z3 error model, for example, represents program states as named model elements, such as ***0 -> true, *37 null** and functions interpretations on elements, such as **MapType1Select -> *40 *41 *17 -> *56; else -> #unspecified**

We will use this example to clarify the discussion of our tool, and will demonstrate the tool on it and other examples.

2.1 Computing Memory Contents

The model produced by the reasoning engine contains values for a number of memory locations in a number of states. Regular debuggers allow the user to inspect values of variables as well as the values of fields of objects pointed to by the variables. BVD follows this model by translating the SMT's model of memory contents into more familiar object trees that the user can explore.

BVD names memory locations using *access paths*—usually source-level expressions that access a given memory location. Example access paths in the `Insert` example include `hd`, `n.next`, and `hd.next.data`. After the execution of `Insert()`, `n.data` and `hd.next.data` name the same memory location. In addition to local variables, BVD can use Skolem constants as roots, allowing the user to explore verification-specific states as necessary.

Values of memory locations are expanded recursively via communication between BVD and the language plug-in. BVD supplies the program points corresponding to counterexample states to the language-specific BVD plug-in. The latter returns a list of root paths—typically local and global variables—available at the static program points. BVD then supplies values of these root paths according to the model, which the plug-in "expands" to yield additional interesting access paths. The expansion may contain source-level fields (when the value is a pointer to an object), indexes into arrays, maps, sets, or applications of functions. The process repeats recursively, in breadth-first order, stopping whenever an already visited node is reached.

2.2 Displaying States and Access-Path Values

Canonical names. Some values, like integers, Booleans, or the special `null` pointer, are easy to display. In contrast, regular pointers in the SMT model do not have user-meaningful values; they are assigned place-holder values. BVD asks the language plug-in to provide canonical (*i.e.*, state-independent) names for them. The VCC and Dafny plug-ins use the type of the object and a sequential number (*e.g.*, `ListNode'0`).

Stable tree view. BVD displays values for both the currently and previously selected execution states, in blue and red respectively, allowing side-by-side comparison. Some paths may be available only in some states, *e.g.*, `hd.next.next` is unavailable in state 1 because `hd.next` is `null`, but is available in state 2. To avoid abrupt changes to the access-path tree while browsing the states, the left pane shows the union of path trees for all states. A path's name is greyed out if its value is unavailable in the selected state.

Fig. 1 shows BVD on the counterexample produced by Dafny on the `Insert` program. Paths `hd` and `n` are roots; `hd.next` was generated by the Dafny BVD plug-in as the expansion of `hd`. The screenshot displays `hd` and `n` in state 1 (Value) as compared to state 2 (Previous). The right-hand pane displays the value of `hd.next` in all states. The use of canonical names illustrates that `hd` and `n` are aliased: they have the same value (`ListNode'0`). The current state shows the value of `old(hd.next)`: (`null`). If we advance to the final state (corresponding to the execution of the last source code line) we will see that `n.next` points to `ListNode'0`, not `null`.

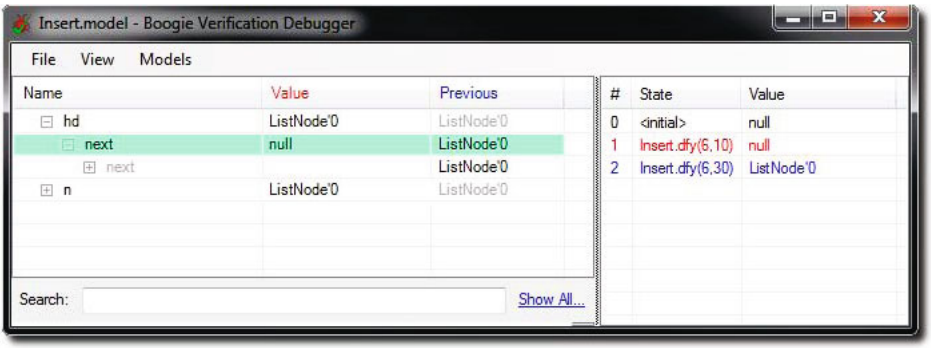

Fig. 1. BVD on the Dafny **`Insert`** example. The right pane provides navigation through execution states, and shows the value of the currently selected access path throughout execution. For example, **`hd.next`** is **`null`** in the first two states, and **`ListNode'0`** in the third. The left pane allows browsing the access path tree and inspecting values in the current (in red) and previously selected (in blue) state, allowing for comparison. Canonical names are provided by the language plug-in for model values that are not otherwise user-meaningful (such as **`ListNode'0`**).

2.3 Complex Data Types and Search

Skolem constants. The screen shots in Figs. 2 and 3 are of a failed VCC verification of a recursive red-black tree implementation. Fig. 2 illustrates the use of canonical names to display complex data type values. The **`m@Red..(79,15)`** is a Skolem constant (**`m`**) for which a quantified invariant at line 79 column 15 fails to hold.

Sparse data structures. Fig. 2 also shows a ghost field **`t->owns<Tree>`**, introduced by the VCC verification methodology and containing the set of objects owned by **`t`**. The set is displayed using canonical names: it contains **`Node'0`**, does not contain **`Node'9`**, and so on. The set representation in the SMT model is *sparse*—it does not say anything about possible **`Node'42`**, because it does not need to: the value of **`Node'42`** is irrelevant to the verification failure. We use a similar display format for maps, arrays, and specification functions (either methodology-supplied like **`$inv2(...)`**, or user-defined).

Search. Canonical names are useful for spotting aliases and understanding changes of variables across states. The user may search for aliases or uses of a given access path value. In a given state, a value is used by an access path if either the location pointed to by the access path contains the value, or the access path itself mentions the value. Fig. 3 shows screenshots of some of BVD's search capabilities. To determine correspondence between canonical names and variables in the current state, the user right-clicks on an access path (left screen shot) or uses the search facility (right screen shot).

Both these features are desirable and possible due to the size of the models: the models are in the range of hundreds (up to a few thousands) of elements—too large for the human being to look at as a whole, but much smaller than the gigabytes of heap that a dynamic debugger may need to consider to provide such features.

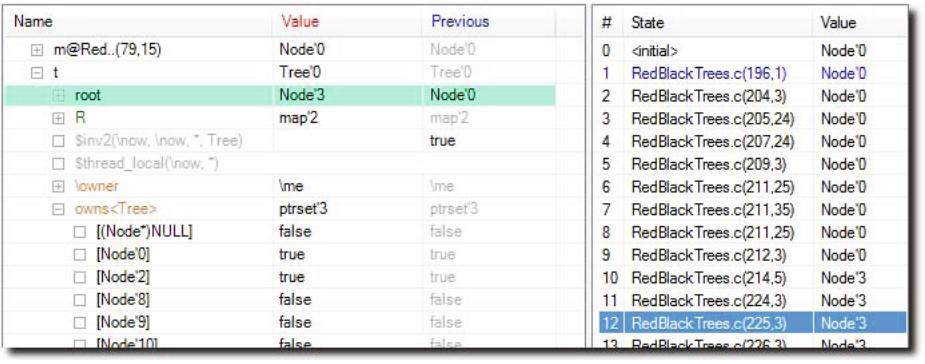

Fig. 2. BVD on a failed VCC verification of a recursive red-black tree implementation. Skolem constants receive canonical names (such as **`m@Red..(79,15)`**) as with regular variables. This example demonstrates BVD's treatment of sparse data structures: only values relevant to the counterexample—those with values in the model—are displayed. **`Node'42`**, for example, is not displayed in the ghost field **`t->owns<Tree>`**, because it is not relevant to the failure.

2.4 Plug-in Programmer Interface

The translator from source language to Boogie needs to insert special assumption statements at program points that capture source code locations and variable values, which are included in the counter-example model and can then be used to decode states displayed in BVD. The source-language BVD plug-in is responsible for enumerating access paths. BVD provides the plug-in with a mapping from Boogie variables to model elements in each marked program point, as well as complete enumeration of model elements and associated function interpretations.

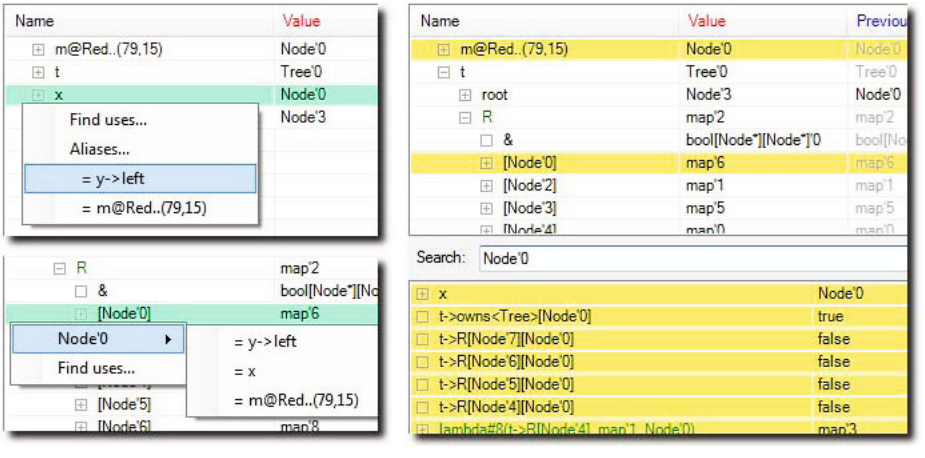

Fig. 3. Search on the red-black tree example. Right-clicking a node produces commands to jump to other access paths containing the current value (aliases). The pop-up menu at the bottom shows three locations containing **`Node'0`** in the current state; the menu above it shows two other locations besides **`x`** (currently selected). The menu also allows populating the full-text search box.

Writing language plug-ins is not difficult. The VCC plug-in is about 1000 lines of C# code, while Dafny is only about 400 (mostly due to Dafny being a leaner language). The modifications needed in the C and Dafny to Boogie translators are negligible.

3 Related Work

The idea of adding instrumentation to verification conditions for the purpose of generating usable error messages is old. For example, ESC/Modula-3 labeled sub-formulas in verification conditions and used the labels returned by the SMT solver to determine the source location of the potential program error [4]. ESC/Java extended the mechanism to allow the reconstruction of an execution trace leading to the error [9]. The Spec# verifier extended Boogie with a mechanism to retrieve the values of certain pre-determined expressions in the error state; for example, this lets the verifier report the value used as an index in an array bounds error.[1]

The forerunner to our work was the VCC Model Viewer[2], which provides a debugger-like, interactive user interface to explore the verification state [2]. BVD integrates into Boogie rather than building on top of it, permitting a simpler encoding. In addition, BVD's architecture supports plug-ins for multiple languages, and through its use of canonical names, permits more advanced features like stable tree views and search.

Müller and Ruskiewicz have implemented a Visual Studio dynamic debugger plug-in for Spec#, with the same purpose [10] as our tool. The debugged program is a variation of the original program that uses values from the SMT model to construct concrete data structures; these are used according to the verification semantics of the program. For example, instead of iterating a loop, the verification semantics immediately jumps to the final loop iteration, where the values of variables are constrained only by the loop invariant, which is how the program verifier views the execution. Their tool can identify some spurious error reports. Our approach is simpler, in that it avoids the many problems associated with constructing concrete data structures from a mathematical model. Furthermore, our approach makes explicit the partial information considered by the SMT solver, which lets us sparsely represent arrays and maps and show functions.

There has also been work to improve the user experience with software model checkers. The typical output of a model checker is a full execution trace leading to an error. There has been work to prune these enormous traces by comparing successful executions with failing executions, seeking to determine the cause of the error [0,6].

The auto-active verifier VeriFast is based on symbolic execution and works with both C and Java programs [7]. Its IDE displays, at each program point, both the heap structure and the constraints on the values of variables and heap locations. It does not currently display concrete values for variables, though it could in principle.

4 Conclusions and Future Work

We have presented BVD, a multi-language verification debugger that helps programmers decipher and diagnose program verifier output. We have built BVD plug-ins for

[1] This `-enhancedErrorMessages` mode was implemented by Ralf Sasse.

[2] Developed by Markus Dahlweid and Lieven Desmet.

VCC and Dafny and found the verification debugger to be useful in practice. For example, it has elucidated why, in the **`SiftUp`** and **`SiftDown`** methods of a priority-queue heap data structure, it is necessary to include a loop invariant that establishes a connection between the parent and children of the node being updated.[3] We believe that tools like BVD are necessary to move verification into the hands of non-experts.

As future work, we wish to conduct user experiments with verification non-experts, perhaps in a teaching setting. We wish to add features that further help narrow the cause of an error (not just debug the symptoms) or identify spurious error reports (see Sect. 3). Adding to the textual tree views provided in BVD, we would like to see complementary visualization (*e.g.*, nodes and arrows) of the data structures in error states. Finally, we would like to see an even tighter integration of aids like a verification debugger into IDEs, so that it can become standard practice to have verification and diagnostic information available to the programmer immediately as code is being designed.

References

0. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. In: POPL 2003, pp. 97–105. ACM, New York (2003)
1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
3. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center (December 1998)
5. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
6. Groce, A., Kröning, D., Lerda, F.: Understanding Counterexamples with `explain`. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004)
7. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
8. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
9. Leino, K.R.M., Millstein, T.D., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Sci. Comput. Program. 55(1-3), 209–226 (2005)
10. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 73–87. Springer, Heidelberg (2011)

[3] See `Test/dafny1/PriorityQueue.dfy` at `boogie.codeplex.com`.