A Visual Inspector for Boogie Programs

Márcio Coelho, Daniela da Cruz, Pedro R. Henriques, and Jorge S. Pinto
{marciocoelho,danieladacruz,prh,jsp}@di.uminho.pt

Departamento de Informática / CCTC
Universidade do Minho, 4710-057 Braga, Portugal

Abstract. *Design-by-Contract* is an approach that allows a programmer to specify the expected behavior of a component by means of preconditions, postconditions and invariants. These annotations (or logical assertions that complement the code) can be seen as a form of enriched software documentation and they can be used to verify that a program is correct with respect to its contracts.
Boogie is an intermediate verification language that is being used by more and more software verification tools as a target language. Actually, several annotation languages are nowadays translated to Boogie language.
Despite of its efficiency and popularity, Boogie, that is also a program verifier, does not contain visual information for the user. So, understanding how it works is a difficult task.
In this paper, we will discuss a visual tool that we developed to help in comprehending Boogie programs.

Keywords: Program Verification, Verification Condition Generators, Design-by-Contract, Boogie, Software Visualization

1 Introduction

Nowadays, more than ever, there is a strong concern to verify software systems. Hospitals, banks, governments and industries use software systems that can not have errors. Such errors can have drastic economic and human consequences. In the past (before the year 2000) the aerospace industry lost over a billion dollars due to severe bugs in software [10].

In this context of safe software, people were seeking for formal approaches (semantic oriented) to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behavior of the implementation matches that of the specification (this is usually called the *functional* behavior of the program), and moreover the program does not 'go wrong', for instance no errors occur during evaluation of expressions (the so-called *safety* behavior).

Following this thread, in the eighties born an approach called *Design by Contract* [8], introduced by Bertrand Meyer, which main idea is to include in a software system the mutual obligations and benefits between a "client" and a "supplier". Both agree on a contract that should be expressed in the source code. This is the metaphor upon which the Design by Contract approach relies.

Thus, by including a specification (using for instance JML [7] or Spec♯ [1]) in the source code it would be possible to establish a proof of correctness. A proof of correctness should mainly detect if a program is inconsistent with respect to its assertions. However, a proof itself can be erroneous. As a mathematician can err in formulating a proof, a program prover can make a similar mistake. The use of verification tools can help in reducing this kind of errors.

Boogie[5] is such kind of tool. Boogie compiler acts like a Verification Condition Generator (VCGen for short) and generates a set of Verification Conditions (VCs) from a program together with its specification. These verification conditions are usually generated in SMT-LIB language [2]. After this first step of generating the VCs, they are sent to a SMT Prover (Z3 [9] is the default one) to check if all they are correct. If all the verification conditions generated can be proved, then the program is guaranteed to be correct with respect to its specification. With Boogie, several provers can be used (e.g. Simplify [6]), but SMT provers are the most popular due to their efficiency.

In the other hand, Boogie is an intermediate verification language. Boogie language is a typed imperative language with procedures and arrays. It can be used to represent programs written in an imperative source language and is accepted as input to Boogie compiler.

A Boogie file includes a lot of information, such as axioms, procedures and their implementations. Due to the amount of information generated from a source program, it may be difficult to understand it. With this in mind we developed a tool that allows us to inspect a Boogie file and understand not only the source code but the process of translation and the process of generating the verification conditions.

In this paper we introduce GamaBoogie, a tool that contains a visualizer for Boogie programs. This visualizer displays the internal Identifier Table containing all the source identifiers and those introduced in the translation process. Moreover, it shows the control flow graph, that is an effective aid on understanding the translation/verification process. Some examples will be introduced, to illustrate the tool usefulness.

This tool makes part of a bigger effort of applying slicing algorithms to Boogie programs. However, before the implementation of such algorithms, a deep study of Boogie was required resulting in the construction of this visualizer.

Structure of the Paper Section 2 explains the main features of the visual inspector. The paper closes in Section 3 with some conclusions and future work.

2 The visual inspector

In this section we will present the different features offered by the visual inspector of GamaBoogie. These include: source code visualization, identifier table and flow graphs of a procedure.

Program example Listing 1.1 calculates the maximum number in an array and will be used as example to illustrate the features of GamaBoogie. Translating this source code into BoogiePL results in a file with a total of 1205 lines. Most of these lines corresponds to axioms and functions needed for the generation of the VCs (for instance, types disappear during the VCs generation; they are, if necessary, encoded as axioms).

```
1  public class maxarray
   {
3    int max;
     int[] vec = new int[100];
5
     private void maxarray1()
7    requires vec!=null;
     ensures 0 <= max && max <= vec.Length;
9    ensures forall{int a in (0:vec.Length) ; vec[a]<=vec[max]};
     {
11     int i = 0;
       max = 0;
13
       while (i < vec.Length)
15     invariant 0 <= i && i <= vec.Length;
       invariant 0 <= max && max <= i ;
17     invariant forall{ int a in (0:i); vec[a] <= vec[max]} ;
         {
19         if (vec[i] > vec[max]) { max = i; }
             i = i +1;
21       }
       }
23 }
```

Listing 1.1. Program example: Maximum of an Array

Source code In order to display the code to the user in a friendly manner, we are using the ScintillaNet[1] component for syntax highlighting. With this feature, the user can get a better visualization of the source code and explore it.

Figure 1 depicts this feature. As can be seen at right, the relevant information about the program is shown — the procedures, marked with a red P, and the implementations of that procedures, marked with a green I. This way, the user can inspect each one of these entities by clicking on it, and thus being redirected to the line that corresponds to such declaration. Besides that, the user can ask to hide the code automatically generated by Boogie with respect to types and axioms, displaying only the code for procedures and implementations. This allows the user to focus only on the code he is interested in.

Identifier Table After a Boogie program be loaded, an Identifier Table is shown. This table is built during the parsing phase of the program provided by the Boogie compiler. This table contains information about the identifiers declared in each implementation. The information displayed in this table includes: name of variable; class where it belongs; method where it is declared; type; and line where it first appears declared.

Figure 2 shows the Identifier Table for the program in Listing 1.1.

[1] http://scintillanet.codeplex.com/

Flow Graphs Boogie compiler already builds a graph structure for a given program. Thus, we took advantage of this fact to display it to the user.

There are two kinds of graphs shown to the user: the Control Flow Graph (CFG) and the Directed Acyclic Graph (DAG).

The user can filter, once again, the information he wants to see: he can start with a global perspective of the graph and then go deeper by inspecting the control flow of a block inside the implementation. Figure 3 shows the CFG of an implementation and Figure 4 shows the CFG of a block inside such implementation.

The visualizer allows to go to the previous graph at any time. When first running, the visualizer shows the available *classes* in the file. Choosing a class, the user can filter which CFG's method he is interested in. At this point we have two graphs for the same *block*: with and without dead blocks.

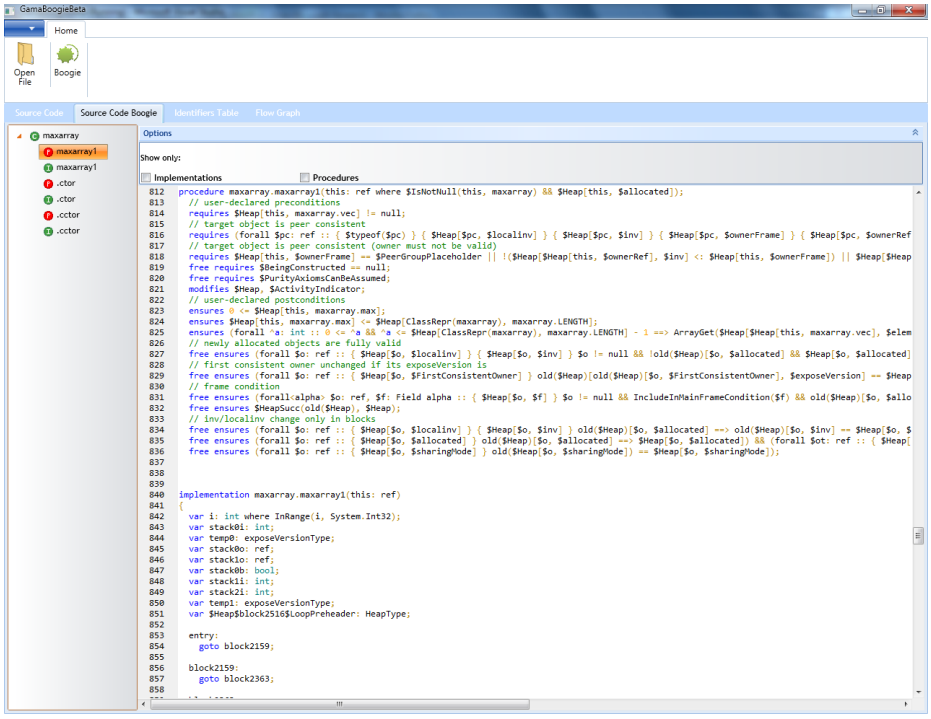

Fig. 1. Source Code visualization feature

3 Conclusions

This work sets up a new visualization tool for Boogie programs, in order to helping users to have a better understanding of the internals of Boogie programs

Identifier	Class	Method	Type	Line
stack1o	maxarray	maxarray1	ref	846
temp0	maxarray	.ctor	ref	1080
stack0o	maxarray	maxarray1	ref	845
stack0o	maxarray	.ctor	ref	1079
temp0	maxarray	maxarray1	exposeVersionType	844
stack0i	maxarray	maxarray1	int	843
stack0i	maxarray	.cctor	int	1184
i	maxarray	maxarray1	int	842

Fig. 2. Part of the Identifier Table

and build the gap existing between the translation of a source program into a Boogie program.

Currently, a promising feature is being implemented: slicing. In [3,4], the authors have shown that slicing a program using its specification produces more aggressive slices — in the sense that they are semantic-sensitive rather than syntactic, as is usual in tradicional slicing algorithms. Originally, this *assertion-based slicing* algorithm was implemented in GamaSlicer, and was applied to a subset of the Java programming language (with JML annotations). However, since Boogie system provides all the pieces necessary to cover a more broad range of programming languages, all these ideas are being carried to GamaBoogie.

References

1. Barnett, M., Leino, K., Schulte, W.: The Spec# programming system: An overview. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices pp. 49–69 (2005)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England) (2010)
3. Barros, J.B., da Cruz, D., Henriques, P.R., Pinto, J.S.: Assertion-based slicing and slice graphs. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 93–102. SEFM '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/SEFM.2010.18
4. da Cruz, D., Henriques, P.R., Pinto, J.S.: Contract-based slicing. In: Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I. pp. 106–120. ISoLA'10, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1939281.1939294
5. DeLine, R., Leino, K.R.M.: Boogiepl: A typed procedural language for checking object-oriented programs. Tech. rep. (May 2005)

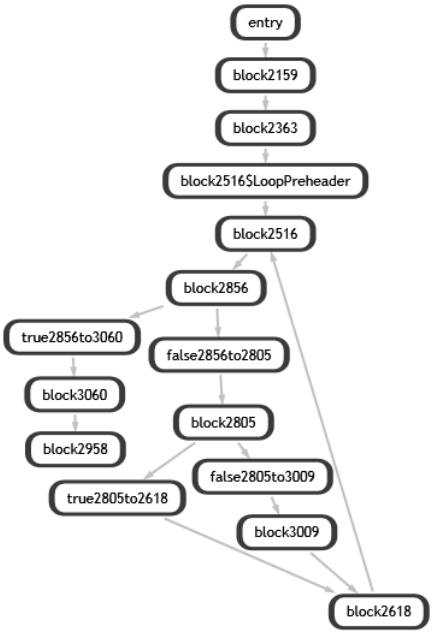

Fig. 3. CFG for an Implementation

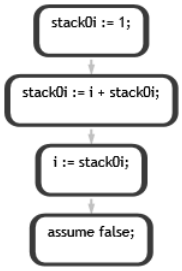

Fig. 4. CFG for a Block

6. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
7. Flanagan, C., Leino, K.: Houdini, an annotation assistant for ESC/Java. FME 2001: Formal Methods for Increasing Software Productivity pp. 500–517 (2001)
8. Meyer, B.: Design by contract. Technical report tr-ei-12/co, Interactive Software Engineering Inc. (1986)
9. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
10. Tassey, G.: The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project (2002)