Slicing Spec# Programs

Sebastian Grössl

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

March 2008

Supervised by:
Joseph N. Ruskiewicz
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

In this paper we present a program slicer for Spec#. We use this tool to slice verification condition failures found by Boogie, the static verifier built as part of the Spec# project. This is an effort to facilitate resolution of such failures and should ultimately help to make contract based programming more attractive.

Contents

Chapter 1

Introduction

Spec# is a new programming system for specification and verification of object-oriented software. The system consists of two components: the programming language Spec# with its compiler and the static verifier Boogie. Boogie is designed to be a language independent verifier and therefore operates on its own intermediate language BoogiePL. Thus in order to verify the program, the Spec# compiler first translates its programs to BoogiePL.

Boogie then attempts to establish all verification conditions and returns a trace leading to the error in the case where verification failed. This trace can be quite large and therefore it can be rather difficult for the programmer to find the cause of this failure. In order to make it easier for the programmer to resolve these verification errors we propose a method to make the trace smaller by using program slicing. Because the programmer is only presented relevant information, it should enable him to resolve those verification errors much more efficiently.

A program slicer for BoogiePL programs[5] has already been developed, but because Spec# programmers shouldn't necessarily be exposed to Boogie's intermediate language, we propose a method to make use of this existing BoogiePL slicer by transforming its results back to Spec# code. Although the same results could be achieved by slicing directly on the Spec# level, our method makes this technique available to all languages supporting verification using Boogie.

After some required background information, Chapter 3 will discuss how to achieve Spec# slices from BoogiePL slices on a conceptual level. In Chapter 4 we will then discuss some aspects important for an implementation.

Chapter 2

Background

2.1 Spec#

The Spec# language[2] is a superset of the programming language C# extending it by non-null types, method contracts, object invariants and an ownership type system. Our main focus in this paper will be on the aspect of contracts. Asserts, assumes, invariants, pre- and postconditions are the contracts present in the Spec# language. These programming constructs allow developers to express semantics of a program. This additional information can then be used by the compiler to ensure the program always retains a valid state.

Listing 2.1 presents a simple Spec# class A with a field i and a method f. In this simple method, it is easy to see that independent of the outcome of the condition in the if statement, the assert on line 11 fails if the method f is called with an argument smaller than 1.

```
1  public class A {
2    private int i;
3    public int f(int x)
4    {
5      int y;
6      if (x < i) {
7        y = 1;
8      } else {
9        y = 0;
10     }
11     assert x > 0; /* This might fail */
12     return y;
13   }
14 }
```

Listing 2.1: The Spec# class A.

2.2 Boogie

The second part of the Spec# system is its static verifier Boogie[1]. Although developed as part of the Spec# system, Boogie has been designed to be source language independent. This is the reason why it operates on its own programming language BoogiePL[3]. As a result any language that can be translated to BoogiePL can then also be statically checked using Boogie.

Listing 2.2 presents an example of a simple implementation declaration in BoogiePL. As seen in this example, BoogiePL employs strictly unstructured control flow. This means that all higher-level conditional statements are represented by blocks and *goto* statements connecting these blocks. Because we are looking at the implementation part of this Boogie program, the only

contracts we can find here are asserts and assumes. Pre- and postconditions are declared outside the implementation part and in a later phase inlined as assume and assert statements.

```
1  implementation A.f(this: ref, x: int) returns ($result: int)
2  {
3    var i: int, y: int;
4  entry:
5    assert this != null;
6    i := $Heap[this, A.i];
7    goto block1, block2;
8  block1:
9    assume x < i;
10   y := 1;
11   goto block3;
12 block2:
13   assume x >= i;
14   y := 0;
15   goto block3;
16 block3:
17   assert x > 0;
18   $result := y;
19   return;
20 }
```

Listing 2.2: The implementation of Boogie method $A.f()$.

Note also that the statement on line 6 represents an array access. The special variable $\$Heap$ represents the heap as a two-dimensional array with a tuple consisting of an object identifier and the qualified field name as its index.

2.3 Translation Function

We now assume we have a translation function that will produce Boogie code from Spec# code. We will represent this function as $Tr[\![S]\!]$ which takes a Spec# statement S and produces a semantically equivalent Boogie statement B. Additionally the function returns a mapping from S to B. This mapping gives us the ability to track which Spec# statement a Boogie statement was generated from.

We have already seen the results of such a translation when we presented the BoogiePL program in Listing 2.2 which corresponds to the Spec# method $A.f()$ presented in Listing 2.1. Note that in order to make our elaborations clearer, we simplify the translations currently in Spec# to what is needed for our example.

The most basic translations are straight forward. Assignments are handled by

$$Tr[\![a = b]\!] \quad = \quad Tr[\![a]\!] := Tr[\![b]\!]$$

where a is a variable and b an expression. Concatenation of statements looks like this:

$$Tr[\![s1; s2]\!] \quad = \quad Tr[\![s1]\!];\ Tr[\![s2]\!]$$

As we have seen already, the heap is handled like an ordinary array in Boogie. Field references are therefore translated into array accesses. For this case, our translation function looks as follows:

$$\begin{aligned} Tr[\![v = o.f]\!] \quad = \quad & assert\ Tr[\![o]\!] != null; \\ & Tr[\![v]\!] := \$Heap[Tr[\![o]\!], T.f] \end{aligned}$$

where o is a reference to an object of type T, f is one of its fields and v is the variable the value is assigned to. Note that this Spec# statement translates into two Boogie statements because we have to make sure the reference o is not null.

Conditionals in Spec# have to be translated into unstructured control flow. To achieve this, our translation function will create two new blocks containing assume statements that restrict the program flow to only one outcome of the condition each. The translation of if statements for example can be formalized as

$$
\begin{aligned}
Tr[\![if\ (q)\ b1\ else\ b2]\!] \quad = \quad & goto\ btrue,\ bfalse; \\
& btrue: \\
& \quad assume\ Tr[\![q]\!]; \\
& \quad Tr[\![b1]\!]; \\
& \quad goto\ bend; \\
& bfalse: \\
& \quad assume\ Tr[\![!q]\!]; \\
& \quad Tr[\![b2]\!]; \\
& \quad goto\ bend; \\
& end: \\
& \quad \ldots
\end{aligned}
$$

where $b1$ and $b2$ are blocks of statements and q is a Boolean expression. We can see the unstructuredness of Boogie very well in the *goto* statement that declares two targets to jump to. On occurrence of such a statement, the verifier will non-deterministically choose one of the targets. The *assumes* in the target block then make sure the right block is chosen. This is because if it reaches a block that contains an *assume* that evaluates to false, it will track back to the last multi-target *goto* and choose a different target.

Finally, *assert* and similarly *assume* statements translate to their equivalent Boogie version

$$
\begin{aligned}
Tr[\![assert\ q]\!] &= assert\ Tr[\![q]\!] \\
Tr[\![assume\ q]\!] &= assume\ Tr[\![q]\!]
\end{aligned}
$$

and *return* statements are translated as

$$
\begin{aligned}
Tr[\![return\ v]\!] \quad = \quad & \$result := Tr[\![v]\!]; \\
& return
\end{aligned}
$$

As mentioned before, we also assume that the translation function $Tr[\![]\!]$ gives us back the mapping $(map(S) \rightarrow B)$ from the source elements of Spec# to the Boogie elements. For simplicity we assume that each unique statement is labeled with a unique label. We will use the natural numbers denoting the lines in the code for these labels. In our example from above, the mapping would look as follows:

$$
\begin{aligned}
map(A.f) \quad = \quad & \{6 \rightarrow 5, 6 \rightarrow 6, 6 \rightarrow 7, 6 \rightarrow 8, 6 \rightarrow 9, 6 \rightarrow 11, 6 \rightarrow 12, 6 \rightarrow 13, \\
& 6 \rightarrow 15, 6 \rightarrow 16, 7 \rightarrow 10, 9 \rightarrow 14, 11 \rightarrow 17, 12 \rightarrow 18, 12 \rightarrow 19\}
\end{aligned}
$$

In conclusion, our translation function can be described by $Tr[\![S]\!] = \langle B, map(S) \rangle$, where S denotes a Spec# statement and B a collection of Boogie statements. In the following it is assumed that this translation is indeed sound. That is, if we verify the Boogie program translated from Spec#, we know that the Spec# program is also verified.

2.4 Slicing BoogiePL

Program slicing[6] is a technique to extract information out of a program that is relevant to a certain criterion. We use this method to make it easier for a developer to resolve issues that the verifier in Boogie detected by showing him a minimal set of statements that lead to a verification error. To this end we are using the slicer for BoogiePL programs developed in [5]. Further on, we will assume that the method described in the paper is indeed sound.

To incorporate this slicer, we have to provide it with a criterion to slice on. Because we want to get a slice for a failed verification condition, this should be our slicing criterion. Also to make the output even easier to understand, we do not want to slice on the whole program, but just one possible run through it. Fortunately, the Boogie verifier provides us with a counterexample in the form of a trace that leads to the failure. We will therefore slice on this trace and get a slice of a program trace that leads to the failed verification condition.

Looking at our BoogiePL program from above, Listing 2.3 shows what our example would look like after being sliced. The criterion used in this example is the assert on line 17 that we identified earlier.

```
5  assert this != null;
6  i := $Heap[this, A.i];
9  assume x < i;
17 assert x > 0;
```

Listing 2.3: The slice of Boogie method $A.f()$

Chapter 3

Conceptual

In this chapter we explain how to achieve Spec# slices from sliced Boogie programs. We show how this can be achieved by using slicing on the Boogie level. Also some problems and their solutions are presented. We finally show how this is considered sound.

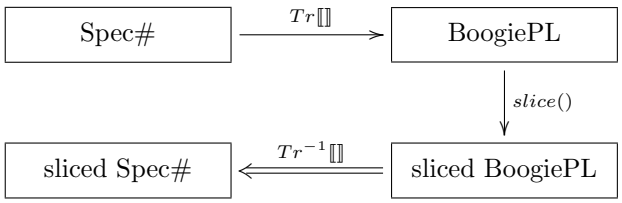


Figure 3.1: Transformations in slicing Spec# programs.

The figure above shows an overview of how the steps described in the background chapter and this one combined can achieve slicing Spec# programs. The Spec# program is translated into BoogiePL using the translation function defined in Section 2.3 and then sliced as described in Section 2.4. This chapter is devoted to the last step which maps back the slice to Spec# code.

3.1 Mapping back the translation function

In the background chapter, we presented the translation function $Tr[\![]\!]$ that given a Spec# program produces a Boogie program with a mapping $map(S)$ from the Spec# to the Boogie statements. Given this mapping, we will reverse the mapping such that for any Boogie statement we can determine the Spec# statement it was derived from. The reverse mapping must also consider that multiple statements in Boogie may map to the same statement in Spec#.

This reverse mapping we represent as the following equation $map^{-1}(B) = S$ which gives us a mapping from a Boogie statement B back to the original statement S in Spec#. We will use this mapping to produce a Spec# slice from a sliced BoogiePL program. It is straight forward to get this reverse mapping from a mapping obtained by our translation function.

For example, the reverse mapping of the method in Listing 2.2 would look as follows:

$$\begin{aligned} map^{-1}(A.f) \quad = \quad & \{5 \to 6, 6 \to 6, 7 \to 6, 8 \to 6, 9 \to 6, 11 \to 6, 12 \to 6, 13 \to 6, \\ & 15 \to 6, 16 \to 6, 10 \to 7, 14 \to 9, 17 \to 11, 18 \to 12, 19 \to 12\} \end{aligned}$$

Given this reverse mapping and the sliced Boogie program from Listing 2.3, we can produce the following Spec# program slice for our example:

```
6  if (x < i)
11 assert x > 0; /* This might fail */
```

Listing 3.1: The slice of Spec# method $A.f()$.

So our reverse translation function $Tr^{-1}[\![]\!]$ will just use the reverse mapping $map^{-1}()$ to translate back to Spec# code.

$$Tr^{-1}[\![B]\!] \quad = \quad map^{-1}(B)$$

In case a statement B that cannot be found in our mapping is given to this function, $Tr^{-1}[\![]\!]$ will not return a Spec# statement for it. Additionally, $Tr^{-1}[\![]\!]$ will also make sure every Spec# statement is only mapped back once.

However, we observe that the slice produced from our reverse mapping here contains an extra statement that does not provide any information as to why the program failed. This is due to the encoding of conditional statements in the form of assume statements.

3.2 Handling of structured statements

We observed in Listing 3.1 that we produce slices larger than necessary. This results from the fact that the Boogie slicer is unable to determine if an *assume* statement derives from a conditional or from an assumption in the methodology. To minimize this impact, we investigated three different options.

3.2.1 Post-processing

One way to solve this is to perform post-processing on the Spec# level. Finding and removing these conditionals would be straight forward, but it turns out that this is not really appropriate. The spirit of Boogie is to free the source languages from having to be concerned with the analysis that Boogie can perform. This method would clearly violate that idea.

But even more important is the fact that we might still get larger slices than are necessary. Assume that we have an additional statement $i := i + 1$ immediately before line 6 in our example in Listing 2.1. Because of the dependency on i in the conditional, this statement would also end up in the slice. Therefore before post-processing the slice would look like

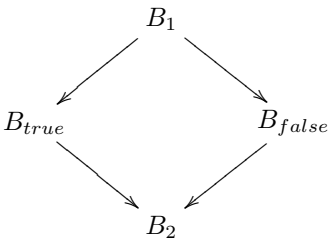
```
5  i := i + 1;
6  if (x < i)
11 assert x > 0; /* This might fail */
```

Listing 3.2: The slice of the extended Spec# method $A.f()$.

and post-processing would only remove the if statement leaving our newly added assignment statement on line 5 in the slice. To achieve the most compact slice possible, we would then have to perform slicing on the Spec# level again. However this would defeat the purpose of having Boogie perform slicing in the first place.

3.2.2 Structural analysis

Another option is to make use of the distinctive block structure produced by the translation of conditionals. As we observed in Section 2.3, *if* statements for example translate to the following block structure:



where B_1 ends in a *goto* with two targets and B_{true} and B_{false} contain complementary assume statements. Detecting these patterns would remove the necessity of post-processing on the Spec# level. However we observe that it is not always possible to determine that the blocks B_{true} and B_{false} just contain assume statements related to conditionals.

In the following example translated to BoogiePL, we wouldn't necessarily know if the *assume c* statement was explicitly introduced in Spec#, or if it is part of the structure of *if* statements

```
1 if (q) {
2   assume c;
3   ...
4 }
```

So this technique would only work under the condition that we knew exactly what type of translation is performed and how it is done. This method is therefore too sensitive and not really viable.

3.2.3 Annotating assume statements

We can remove the limitation of the translation by allowing *assume* statements to carry one more piece of information. We can either put more dependency on the comments before *assumes* or modify the syntax for *assume* statements. Our implementation makes use of the first option and as such checks the comments before *assumes* to find out if they came from a conditional. But since both variants have the same effect, we will focus on the discussion of the second, cleaner option here.

In that case we would change the syntax of *assume* statements to *assume q, cond* where *cond* is either *true*, meaning that this statement was translated from a conditional (i.e. from the program text) or *false* to denote that it originates from the Boogie methodology or an *assume* in the source language. We then have to adapt our translation function for *if* to

$$
\begin{aligned}
Tr[\![if\ (q)\ b1\ else\ b2]\!] \quad = \quad & goto\ btrue,\ bfalse; \\
& btrue:\ assume\ Tr[\![q]\!],\ true; \\
& \quad Tr[\![b1]\!]; \\
& \quad goto\ bend; \\
& false:\ assume\ Tr[\![!q]\!],\ true; \\
& \quad Tr[\![b2]\!]; \\
& \quad goto\ bend; \\
& end: \\
& \quad \ldots
\end{aligned}
$$

and similarly for the other conditionals. Any other *assume* statement would then have their *cond* parameter set to *false*. Like this conditionals can easily be distinguished and removed from the slice in case they do not contain any other statements, as is the case in our example from above.

Using this last solution we would translate our extended example from Section 3.2.1 to the following in BoogiePL

```
1  implementation A.f(this: ref, x: int) returns ($result: int)
2  {
3    var i: int, y: int;
4  entry:
5    assert this != null;
6    i := $Heap[this, A.i];
7    i := i + 1;
8    $Heap[this, A.i] := i;
9    goto block1, block2;
10 block1:
11   assume x < i, true;
12   y := 1;
13   goto block3;
14 block2:
15   assume x >= i, true;
16   y := 0;
17   goto block3;
18 block3:
19   assert x > 0;
20   $result := y;
21   return;
22 }
```

Listing 3.3: The implementation of the extended Boogie method $A.f()$.

which would then produce the following minimal slice:

```
19 assert x > 0;
```

Now mapping back to Spec# is straight forward and we can see that our slice is indeed much smaller and more compact then without our extensions for handling conditionals correctly.

Concluding this chapter, we have seen how to map back to the original Spec# code from its translated BoogiePL form. We have also annotated *assume* statements to be able to slice conditionals correctly. Assuming that the transformations described in the background section are sound, we can conclude that our Spec# slicer is sound too. This is indeed the case because it proves to be equivalent to a slicer written specially for the Spec# language.

Chapter 4

Implementation

In the previous chapter we have seen how we map back slices of BoogiePL programs. According to our Figure 3.1 this should enable us to build a Spec# slicer. But we have omitted a very important detail so far. There are additional transformations on the BoogiePL program that are needed to prepare it for verification condition generation. These transformations alter the Boogie program in ways that make it impossible to easily map back to Spec# using the method described above. Therefore in this chapter we will discuss some of these transformations and their impact on an implementation.

4.1 Loop invariant inference

One such transformation is the loop invariant inference. In this phase, Boogie uses abstract interpretation to generate loop invariants that are then inserted into the program. The special property of this transformation is that it only adds elements to the BoogiePL program and does not change it in any other way. As we have seen in Section 3.1 this poses no problem to our function $Tr^{-1}[\![]\!]$ because the additionally added program elements will just not map back to anything in the source Spec# code.

4.2 Passification

The most interesting transformation happens in the passification phase[4]. This is an essential optimization step that replaces all assignments with semantically equivalent *assume* statements. The verification conditions generated from these assignment-free programs can then be proven much more efficiently. Compared to the loop invariant inference step, this phase alters the program in various places by replacing assignments with *assumes* and by substituting variables. For example as we have already seen in Listing 3.3 the Boogie equivalent of the Spec# statement $i := i + 1$ is

```
1 assert this != null;
2 i := $Heap[this, A.i];
3 i := i + 1;
4 $Heap[this, A.i] := i;
```

Listing 4.1: A simple increment in BoogiePL.

After passification, all the assignments are replaced by *assume* statements and the variable i is substituted by versions that are not changed further. The assignment to the heap is even more complex, involving duplicating it and invoking an additional helper function that realises these changes. So our simple increment would then look as follows:

```
1 assert this != null;
2 assume i@0 == $Heap[this, A.i];
3 assume i@1 == i@0 + 1;
4 assume $Heap@0 == store2($Heap, this, $ev, tmp0);
5 assume $Heap@1 == store2($Heap@0, this, A.i, i@1);
6 assume IsHeap($Heap@1);
```

Listing 4.2: A simple increment in BoogiePL after the passification phase.

We observe that these changes to the program elements pose a serious problem to our mapping function, because they change the program quite dramatically. It is not possible anymore to easily map back the replaced program elements, because our mapping function can only map back the elements recorded by the transformation function.

One solution to this problem would be to create a copy of the whole program before it is altered. Then after the intrusive changes one can try to map back to the copy again. By exploiting block labels that stay the same and the structure of the statements in the blocks it is possible to achieve this. Although not optimal, this scheme provides for a relatively easy implementable way to close the last gap in our slicing efforts and that is the reason we use it in our implementation. Also, because of the dependence on comments introduced by our solution from Section 3.2.3, we must bring back these comments even though they are optimized away at some point. This option allows us to achieve that easily.

However, this is hardly the optimal solution. A more general method would allow mapping back to the original source language from any stage in the verification process. This is especially important for a verifier like Boogie that was designed to be source language independent. To be usable, the verification internals should be hidden from the user so that all the information he's faced with actually makes sense to him, i.e. the code is returned in the language it was originally written in. Apart from making this possible, this idea also takes into account that additional optimisation and verification passes might be added to Boogie at some point. An additional advantage is that it does not only make our slicing efforts easier, it also simplifies the current error reporting system.

Therefore, we propose introducing a source location indicator to every element that gives us the location of a program element in the original source code. This additional piece of information should always be retained when new elements are added to the program that substitute some other element. Using this scheme, it should always be possible to map back to the statement in the original language from any stage in the verification process. This extra information could be compared to debugging symbols generated by compilers, that allow mapping back to the exact position in the source code when debugging code.

Chapter 5

Conclusion

We have presented a slicer for Spec# programs to make resolving verification failures easier for developers. It works by applying slicing to the BoogiePL representation of the program, which we translate back to the original Spec# code again. In this back-translation, we have observed that conditionals pose a particular problem, because they translate to simple *assume* statements in Boogie. Because these *assume* statements are indistinguishable from *assumes* that were implicitly declared or added by the methodology, we have determined that annotating them with their origin resolves this issue.

We have also reasoned that a verifier which is designed to be language independent should have a reliable method to go back to the original source code from any point in the verification process. This would enable improved error reporting methods like program slicing and would also be very useful for existing error reporting facilities.

Bibliography

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, pages 364 – 387. Springer-Verlag, 2006.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49 – 69. Springer-Verlag, 2005.

[3] Robert DeLine and Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March, 2005.

[4] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193 – 205. ACM, January 2001.

[5] Karin Freiermuth. Using program slicing to improve error reporting in Boogie. Master's thesis, ETH Zurich, September 2007.

[6] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439 – 449. IEEE Press, 1981.