

Using Program Slicing to Improve Error Reporting in Boogie

Karin Freiermuth

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

September 2007

Supervised by:

Joseph Ruskiewicz
Prof. Dr. Peter Müller

Abstract

This paper presents a new program slicing approach, which provides, on one hand, slicing programs containing contracts and, on the other hand, the handling of arrays for the purpose of obtaining efficient slices.

Program slicing is useful in debugging complex programs. BoogiePL is an intermediate language used for program verification. In order to apply program slicing on BoogiePL programs, we have to handle contracts. The proposed slicing techniques so far, do not involve contracts. This work presents program slicing in the presence of contracts which makes it possible to slice BoogiePL programs.

Most slicing techniques suffer from generating slices approaching the size of the original program. To overcome this, the thesis proposes an approach how to take advantage of Boogie's theorem prover to handle arrays in order to achieve more compact program slices.

Contents

1	Introduction	7
1.1	Overview	7
1.2	Structure	7
2	Background	9
2.1	Control Flow	9
2.2	PDG - Program Dependence Graph	10
2.3	Program Slicing	12
2.4	BoogiePL Language	15
2.5	Boogie - A Static Verifier	17
3	Program Slicer	21
3.1	Handling of goto's	21
3.2	Dependencies of Program Statements	22
3.3	Dependencies of Contracts	24
3.4	Extended Statements	29
4	Implementation	31
4.1	Control Flow Graph Generation	31
4.2	Program Dependence Graph Generation	32
4.3	Using the Tool	34
5	Using the Theorem Prover in the Presence of Arrays	37
5.1	Handling of Arrays using Static Flow Analysis	37
5.2	Using the Theorem Prover	37
6	Conclusion	41

Chapter 1

Introduction

1.1 Overview

BoogiePL is an intermediate language providing a basis for verification condition generation of object-oriented programs including specifications. Boogie is a static verifier [1] for BoogiePL programs [2] and responsible for verification condition generation and theorem proving. Boogie guarantees finding the error whenever the program is not correct. However, the reported error does often not indicate the real cause of the error.

The goal of this master thesis is to reduce the complexity of the original BoogiePL program in order to help the programmer debugging and understanding the program. The complexity should be reduced by extracting only those parts of the original program that are relevant with respect to a certain point of interest. Reducing the complexity of a program with respect to failed condition helps the programmer understanding and fixing the error.

The applied technique in order to reach reduced complexity, is program slicing. Program slicing takes a program and a point of interest, the slicing criterion, as input and generates a subset of statements of the input program, called slice. A slice consists of at least all the parts that might have influence on the input criterion.

Program slicing was introduced by Weiser in 1979 [3]. During the past decades several program slicing techniques have been proposed. An overview is available in [4]. In theory most of these slicing techniques work well. The proposed techniques, however, firstly do not involve contracts neither have they proposed an efficient way of program slicing in the presence of arrays.

This thesis presents a program slicing approach which improves traditional slicing in that way that it involves programs which contain contract statements, and slices programs which contain arrays efficiently.

So far program slicing has been executed on programs containing only program statements. Working with BoogiePL programs, which do not only contain program statements but also contracts, program slicing needs to be extended to handle contracts. An implementation of extended program slicing in the presence of contracts is given in this thesis.

In the presence of arrays, the size of the resulting slice based on static analysis does generally less differ from the original program, which makes program slicing less attractive. BoogiePL programs in practice mainly consist of arrays. Therefore a way has to be found how to treat arrays in terms of program slicing to generate efficient slices. This thesis extends the technique of static program slicing by using the theorem prover. The new slicing technique takes advantage of the theorem prover in order to improve program slicing in the presence of arrays.

1.2 Structure

The thesis is organized as follows: Chapter 2 gives a short overview of the basic elements that are used in this thesis. Sections 2.1 and 2.2 explain two internal program representations, that

have been chosen as base for the presented implementation of program slicing. Program slicing itself is described in detail in Section 2.3. Finally, the idea of program slicing based on a program dependence graph as internal program representation is given. The subsequent sections go into Boogie: Section 2.4 gives an overview of the relevant language specifications of the BoogiePL. Section 2.5 introduces Boogie and its functions.

Chapter 3 introduces program slicing based on a *program dependence graph (PDG)*. PDG generation uses static flow analysis and is based on the *control flow graph (CFG)* (Section 2.1). A PDG is built up by finding data and control dependencies. Data dependencies of program statements have already been introduced in earlier approaches. Section 3.2 explains data dependence for the basic program statements using static flow analysis. Contracts in BoogiePL are assume and assert statements. Section 3.3 presents an approach how to deal with contracts in terms of dependencies. The described approach turns traditional program slicing into an extended slicing technique, which is able to slice programs containing contracts.

The concrete application of the approach in Chapter 3 is described in Chapter 4. This chapter illustrates the concrete implementation of the extended program slicing approach, which is able to slice BoogiePL programs.

Chapter 5 extends the approach given in Chapter 3. This chapter introduces a new slicing approach to generate efficient slices in the presence of arrays. After illustrating the problem of redundant array dependencies in traditional static slicing, it gives an idea of how to include Boogie's static verifier to help eliminating redundant array dependencies.

Chapter 2

Background

2.1 Control Flow

CFG - Control Flow Graph

Flow analysis is used to determine invariant facts about a certain point of interest in a program. These facts are independent of the execution path of the program. Flow analysis is used to get information about a program at compile time and is often applied in compiler optimization [?]. A control flow graph is an internal program representation using graph notation and provides a basis for flow analysis.

Definition 1 (Control Flow Graph). A *control flow graph (CFG)* is a directed graph consisting of all possible flows of control in a program. In a CFG, the nodes represent blocks (Definition 2) and the edges indicate the possible flow of control from block to block. CFGs contain special nodes, namely the entry and exit node, corresponding to the beginning and the end of the program, respectively.

Block

Definition 2 (Block). A block is a linear sequence of program statements. It has one entry point and one exit point. A block may have many predecessors and many successors. An entry block has no predecessors and exit blocks do not have any successors.

Example 1 (Original BoogiePL program).

```
1 int u where u == 1;
2 int w where w == 1;
3 int x where x == 0;
4 int y where y == 0;
5 int z where z == 0;
6
7
8 entry:
9   assume x == 0;
10  assume y == 0;
11  assume z == 0;
12  x := w;
13  y := w;
14  assume x == w;
15  z := w;
16  goto a, b, c;
```

```

17
18 a:
19   x := w;
20   goto exit;
21
22 b:
23   x := 1;
24   goto exit;
25
26 c:
27   call z := Max(u, w);
28   w := 2;
29   x := 2;
30   z := 2;
31   assert x == z;
32   y := x;
33   y := 1;
34   goto b;
35
36 exit:
37   havoc z;
38   w := z;
39   z := x + y;
40   assert z == 3;
41   return;

```

Figure 2.1 shows the control flow graph of the BoogiePL program in Example 1.

2.2 PDG - Program Dependence Graph

A *program dependence graph (PDG)* is an internal program representation. It is a directed graph with nodes corresponding to statements and control predicates, and edges corresponding to data dependencies (Section 2.2) and control dependencies (Section 2.2). Data and control dependencies are based on the CFG (Definition 1) of a program.

Data Dependency

A data dependency between two statements exists, if one statement directly defines the referenced value of the other statement. *Directly* means in this case that no other statements that redefine the referenced value are in between.

Definition 3 (Data dependency). There is a data dependency from one to another statement if there is some variable z such that

- the first statement may be assigned to z ,
- the second statement may use the value in z , and
- there exists an execution path in the CFG between the two statements.

The data dependencies of a statement can be obtained by finding all transitive data dependencies of its referenced variables. Figure 2.2 shows the data dependencies for statement $z := x + y$ in Example 1.

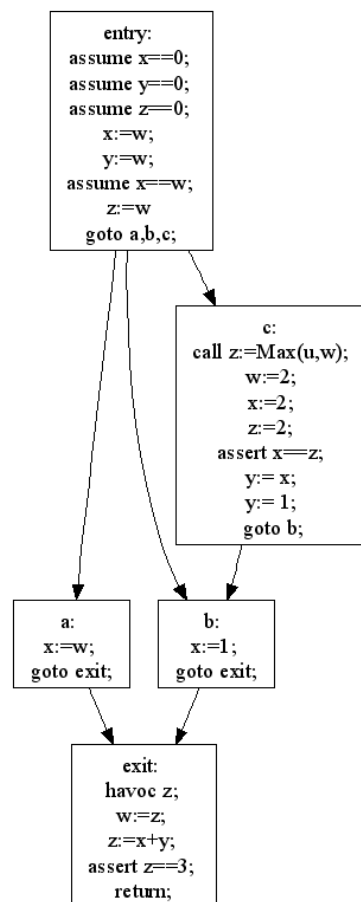


Figure 2.1: Control flow graph of Example 1

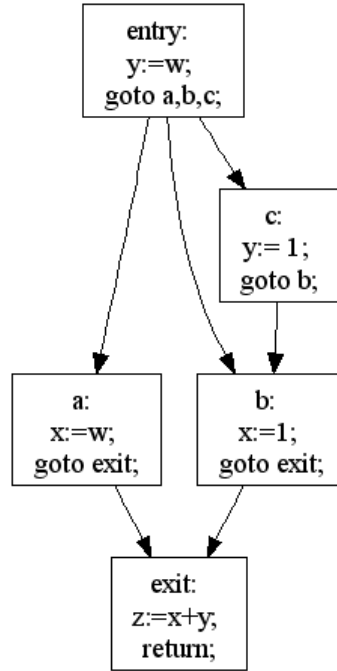


Figure 2.2: Data dependencies of statement $z := x + y$ in Example 1.

Control Dependency

A control dependency between two statements exists, if there is a trace in the CFG and if the execution of one statement depends on the other statement.

Definition 4 (Control dependency). There is a control dependency from a *control* statement to a program statement if the boolean outcome of the control statement decides whether the program statement is executed.

```

1 if (z == 0) x := 0;
2 else x := 1;
3 return;
  
```

Figure 2.3 shows the control dependencies of the program above.

2.3 Program Slicing

Program slicing is a technique to extract the parts of a program which are relevant to a given criterion. The criterion usually is given by a variable at a certain program statement. Program slicing was first introduced by Weiser in 1979 [3]. During the past years, many program slicing algorithms have been introduced [4]. The original technique proposed by Weiser is based on solving data flow equations. Program slicing generally reduces a certain program to a subset of statements of the original program. Slicing means, to extract all those statements of the original program which have influence on a certain variable at a statement of interest and get rid of the redundant statements in terms of the specified criterion. The resulting slice is supposed to be of reduced complexity compared to the original program. By this means, slicing helps the programmer understanding and debugging a program.

Definition 5 (Slicing criterion). A slicing criterion is a variable at a certain statement of interest.

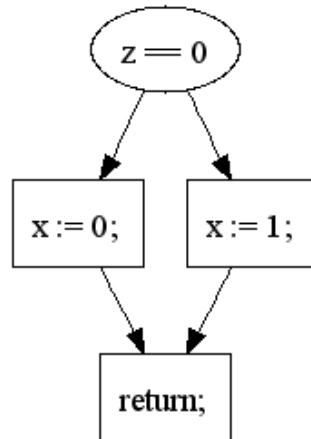


Figure 2.3: Control dependence graph

Example 2 (Slicing criterion).

One possible slicing criterion of the program in Example 1 is variable `z` in statement `assert z == 3`.

```

1 ...
2 exit:
3   z := x + y;
4   assert z == 3; //Slicing criterion
5   return;

```

Definition 6 (Program Slicing). Program Slicing is the process of generating a slice of a given program. A slice is a subset of statements of the original program which, whenever the original program terminates, behaves the same way as the original program, with respect to a certain slicing criterion (Definition 5).

The following program represents a slice of the program in Example 1 with respect to the slicing criterion in Example 2:

Example 3 (Slice).

```

1 int w where w == 1;
2 int x;
3 int y;
4 int z;
5
6 entry:
7   assume x == 0;
8   assume y == 0;
9   assume z == 0;
10  y := w;
11  assume x == w;
12  goto a, b, c;
13
14 a:
15  x := w;
16  goto exit;

```

```

17
18 b:
19   x := 1;
20   goto exit;
21
22 c:
23   assert x == z;
24   y := 1;
25   goto b;
26
27 exit:
28   z := x + y;
29   assert z == 3; //Slicing criterion
30   return;

```

Slices

Program slicing is a decomposition technique. Weiser's original definition of program slicing is based on an iterative solution of dataflow equations. Weiser defines a slice as an executable program that is obtained from the original program by deleting zero or more statements. In order to be a slice with respect to a slicing criterion (Definition 5), a subset of statements of the original program has to satisfy the properties in Definition 6.

For a slicing criterion exists at least one slice, namely the program itself. A slice is minimal if no other slice of the same slicing criterion contains fewer statements. According to Weiser, minimal slices are not necessarily unique and the problem of finding the minimal slice is undecidable.

Program Slicing based on Solving Data Flow Equations from the CFG

Weiser described conventional slicing as obtaining a slice by iteratively solving data flow equations. The data flow equations are based on the CFG of a program and on a slicing criterion. Weiser's iterative algorithm uses two distinct layers of iteration:

1. Tracing transitive data dependencies,
2. tracing control dependencies, which includes control predicates in the slice. For each such predicate, step 1 has to be repeated to include the statements it depends on.

It is not difficult to see that in using Weiser's iterative algorithm we have to start over new for each slicing criterion. This means for every new slice that has to be generated the complete set of data flow equations has to be established. Considering the efficiency of Weiser's iterative algorithm, it seems obvious that the process of generating different slices of a program is time consuming. Generating a slice from a given criterion means solving a set of data flow equations, which takes $O(n^2)$ time (where n is the number of statements). This process needs to be repeated for every new criterion. The resulting cost in order to generate m slices finally results in $O(m*n^2)$.

Program Slicing based on a PDG

Reachability Problem

Since solving sets of data flow equations based on the CFG (control flow graph, Section 2.1) is time consuming (Section 2.3) we want to take advantage of the PDG (program dependence graph, Section 2.2). Using a PDG instead of a CFG as internal program representation prevents from solving different sets of data flow equations for each point of interest. Using a PDG, program slicing can be redefined as reaching problem. Slices are obtained by traversing the PDG. The main advantage of the PDG is that slices can be extracted in linear time. The idea of defining

program slicing as reaching problem based on a PDG was introduced by Ottenstein and Ottenstein and is described in detail in [5]. The following section explains the main idea.

Using a PDG

Approaches considering program slicing as reachability problem are based on a PDG. Slice generation basically includes two steps: In a first step, a PDG is established from the original program. The PDG consists of nodes representing a program statement. The nodes are connected according to their data and control dependencies. A slicing criterion is represented as a node. In a second step, from a given slicing criterion the slice can be obtained by following the edges backwards, starting at the criterion node. The resulting slice consists of all those nodes collected on all possible backward walks starting at the criterion node and ending at the node representing the first statement in the program.

A PDG is an ideal program representation for constructing program slices because it consists of exactly those dependencies, namely data and control dependencies which are used to generate a slice. Compared to the CFG, a PDG omits redundant information, such as sequential dependencies, which do not give any information in terms of slice generation. Working with a PDG a slice can be extracted in linear time by graph traversing.

The advantage of the PDG approach is: Once a PDG has been established, slices can be obtained with respect to any desired criterion without much effort. The process of generating slices takes only linear time for each slice and is executed by graph traversing. However, establishing a PDG from a program depends on the number of statements n , and takes $O(n^2)$ time. Due to the fact that slice generation of every additional slice takes $O(n)$ time, the resulting time to generate m slices of a program consisting of n statements takes $O(n^2) + O(m * n)$ time. The slicing approach based on a CFG, where for each slicing criterion a set of data flow equations has to be solved, takes time $O(n^2)$ for every new slice, resulting in $O(m * n^2)$ for m slices. When focusing on generating multiple slices according to varying criterions it is preferable to choose a PDG as internal program representation.

2.4 BoogiePL Language

BoogiePL is an intermediate language for program analysis and program verification. The language is a simple coarsely typed imperative language with procedures and arrays. Additionally, functions can be introduced and properties of these functions can be declared. Boogie can generally be used to represent programs in imperative source languages, such as Spec#. From BoogiePL one can generate verification conditions or perform other program analysis afterwards. BoogiePL is accepted as input to Boogie, a static program verifier.

BoogiePL Types

BoogiePL has a simple type system containing the basic types `bool`, `int`, `ref`, `name`, `any`, `type name` and `array type`. Type `bool` represents boolean values, Type `int` represents mathematical integers, `ref` represents references such as objects, pointers and addresses. The type `name` represents various kinds of defined names like type and field names. Type `any` is a super type of all other types. BoogiePL allows one dimensional and two dimensional arrays. Even if the type information is erased during the verification process simple types are introduced in BoogieBL. The reason for having types is to improve readability.

This section summarizes those parts of the BoogiePL language which are later used in this thesis.

Type := `bool` | `int` | `ref` | `name` | `any` | `[Type]Type` | `[Type,Type]Type`

BoogiePL Programs

A BoogiePL program consists of a set of declarations. Among others, variables and procedures can be declared as follows:

Variables

In BoogiePL variables are global variables, local variables, parameter variables, such as input and output variables and expression-bound variables. A global variable is accessible to all procedures.

Variable := `var` `IdTypeList`;
 IdList := `Id` [, `IdList`]*
 Id := `String`

Procedures

A procedure consists of an implementation body and might additionally contain specifications such as pre- and postconditions and declarations of input and output variables. Input and output variable declarations are described in detail later.

The procedure specification consists of `requires`, `modifies` and `ensures` clauses. The `requires` and `ensures` expressions are of type `bool`. The input parameters are in scope of the `requires` clause and input and output parameters are in scope of the `ensures` clause.

The implementation body consists of an optional set of local variables and a number of blocks.

The execution of a procedure consists of setting the output parameters and local variables to arbitrary values of their types, and afterwards executing a sequence of blocks, beginning with the first listed block and continuing to other blocks according to the listed labels in the transfer commands. If the transfer command is a `goto` command the indicated blocks are executed, if the transfer command is a `return` command then the execution of the procedure ends. Local variables are accessible within the respective procedure and are declared as described in the previous section.

Each block is identified by a block label, that must be distinct from all other block labels in the same procedure. A block is a sequence of commands. The last command of each block is a transfer command, which is either a `goto` or a `return` command. Each label listed after a `goto` must refer to a block in the same procedure.

A command is an assignment, a `havoc` statement, a passive command or a procedure call. A `havoc` removes any information about the variable by assigning an arbitrary value. The type on the right of an assignment must be assignable to the type on the left, which means, that either the variable on the left is of type `any` or the types are identical.

Procedure := `procedure` `String`(`Variable`*) [`returns`(`Variable`)]?
 Specification? `ImplementationBody`?

Specification	:=	SpecificationClause*
SpecificationClause	:=	RequiresClause EnsuresClause ModifiesClause
RequiresClause	:=	requires Expression;
EnsuresClause	:=	ensures Expression*;
ModifiesClause	:=	modifies Expression;
ImplementationBody	:=	VariableDecl* Block ⁺
Block	:=	Id Command* TransferCmd
Command	:=	String := Expression; Id Index := Expression; assert Expression; assume Expression; havoc IdList; call [IdList :=] Id ([ExpressionList]);
Index	:=	[Expression] [Expression, Expression];
ExpressionList	:=	Expression[, ExpressionList]*
TransferCmd	:=	goto IdList; return ;
Id	:=	String
IdList	:=	Id[, IdList]*

Assertions in BoogiePL

In addition to program statements the BoogiePL language contains assertions. An assertion is a contract. It is either an **assume** or an **assert** statement. With the use of assertions the programmer can express the intended behavior of the program as follows:

- The programmer only cares about the program execution if the **assume** expression holds. In case of an **assume** statement which does not hold, the program can do anything.
- The programmer claims the **assert** expression to hold. If it does not the program is not correct.

In Boogie the two passive commands behave as follows:

- If the expression of an **assume** command holds then the command terminates. If the expression does not hold then the program execution stalls.
- If the expression of an **assert** command holds then the command terminates. If the expression does not hold then the program execution is aborted to prevent the program from a wrong execution.

2.5 Boogie - A Static Verifier

This section summarizes some important aspects of Boogie. Details are given in [1].

Program Verification

Boogie is a static verifier used to verify BoogiePL programs (Section 2.4). To verify a program, Boogie generates *verification conditions (VCs)* in a first step. *Verification condition generation (VC Generation)* involves the statements and the declarations in the program. Additionally other properties of the source language have to be guaranteed. The resulting VCs are passive commands, namely **assume** and **assert** statements represented in first-order formulas. The resulting passive commands can be verified by Boogie's theorem prover. The theorem prover verifies the VCs using weakest precondition calculus, which is explained in detail in the next section. If the program is not correct, Boogie guarantees to find the error. In that case an error message containing the failed condition is returned.

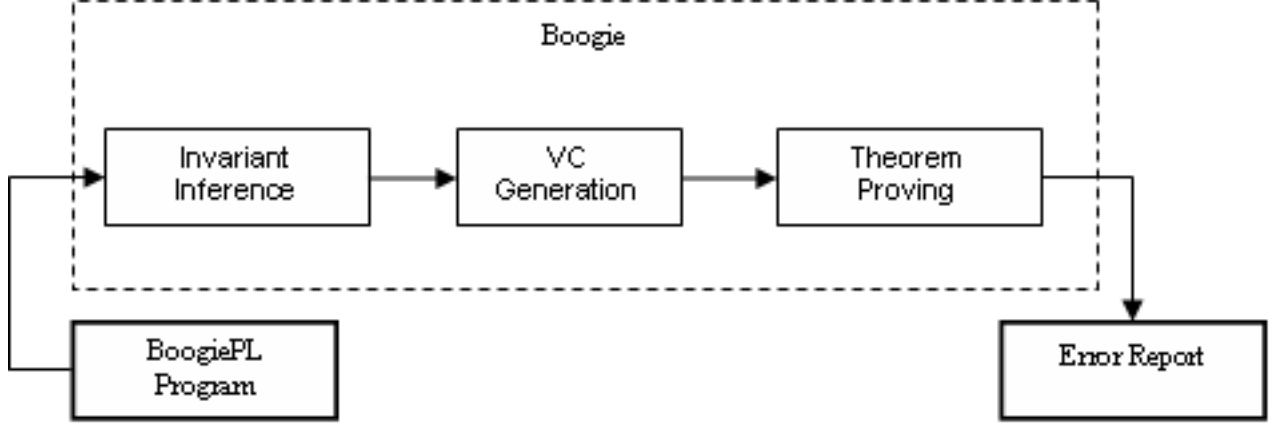


Figure 2.4: The Boogie pipeline: Boogie takes as input a BoogiePL program and returns a failed condition in case of an error.

Verification Condition Generation and Verification in General

BoogiePL programs are verified by performing a series of transformations. Before the verifying process Boogie transforms all procedure statements into passive commands. The final verification is done using *weakest precondition (WP)* calculus: Applying *WP* calculus, Boogie generates a VC for each block. The idea behind weakest precondition is to check for every program state if it is true or not. As soon as a program is in a false state the verification fails.

The *WP* process checks the program states using backwards analysis: As long as the *prestate (P)* is at least as strong as both, the *weakest precondition (WP)* of a statement and the *poststate (Q)*, the program verification will not fail. For a prestate to be true, it has therefore to imply the weakest precondition. This is expressed by the following implication:

$$P \Rightarrow WP(\text{statement}, Q).$$

At this point of verification the program consists of passive commands only, which means that the *statement* is a passive command. Given a passive command with an expression E and a poststate Q , *WP* for passive commands is calculated as follows:

$$WP(\text{assume } E, Q) \equiv E \Rightarrow Q.$$

$$WP(\text{assert } E, Q) \equiv E \wedge Q$$

With respect to the evaluation of the **assume** expression E , the two following cases can arise:

- The assume expression E evaluates to *true* implies: If Q is *true* then the weakest precondition is *true*. P is *true* because the weakest precondition is implied by P . Therefore the program state P is *true*.
- The assume expression E evaluates to *false* implies: The weakest precondition of E and Q is *true* anyway. P is *true* as well because P implies the weakest precondition. Therefore the program state P is *true*.

With respect to the evaluation of the `assert` expression E , the two following cases can arise:

- The assert expression E evaluates to *true* implies: If Q is *true* then the weakest precondition is *true*. P is also true because the weakest precondition is implied by P . Therefore for the program state P is *true*.
- The assert expression E evaluates to *false* implies: The weakest precondition of E and Q is *false* anyway. Therefore P is *false* because P implies the weakest precondition. Therefore the program state is *false*.

As soon as a *false* program state is encountered (see last point), the verification fails.

Verification in Boogie

Using weakest precondition calculus explained above, Boogie tries to verify all execution paths in the control flow graph. It does so by introducing a boolean block variable for each block. A block variable corresponding to $Block_i$ is defined to be *true* if every execution path starting from $Block_i$ is correct.

The control flow graph of Example 1 consists of the following execution paths:

- $Block_{entry} \rightarrow Block_a \rightarrow Block_{exit}$,
- $Block_{entry} \rightarrow Block_b \rightarrow Block_{exit}$,
- $Block_{entry} \rightarrow Block_c \rightarrow Block_b \rightarrow Block_{exit}$.

After introducing block variables $block_i$ the *BlockEquations* can be generated for each block.

The boolean value of block variable $block_{entry}$ is implied by

$$WP(\text{PassiveCommands of } Block_{entry}, block_a \wedge block_b \wedge block_c).$$

This implication leads to the following BlockEquation for $block_{entry}$:

$$WP(\text{PassiveCommands of } Block_{entry}, block_a \wedge block_b \wedge block_c) \Rightarrow block_{entry}$$

The WP function computes the weakest precondition of the *PassiveCommands* of $Block_{entry}$ with respect to the postcondition $block_a \wedge block_b \wedge block_c$.


From the facts above the VC of the whole program can be concluded:

$$Axioms \wedge BlockEquations \Rightarrow block_{EntryBlock},$$

where $Axioms$ is the conjunction of all axioms in the program and $BlockEquations$ is the conjunction of all *BlockEquations* and $block_{EntryBlock}$ is the unique entry block of the implementation (the first executed block).

The program verifier states a program as correct if the final VC of the program is *true*. For a program where no failed condition could be encountered, Boogie guarantees the program to be correct (Section 2.5).

Verification conditions are encoded in such a way that, in case of an error, the trace leading to the error can be reconstructed from the failed verification proof. During the process of translating the BoogiePL program into a verification condition, explained above, a mapping table is built up. The mapping table maps labeled subformulas to the original BoogiePL program elements and provides information to map back labeled output from the theorem prover into an error message in terms of the original BoogiePL program.

A terminal window titled "/SpecSharp/Boogie/BoogieExamples" with standard window controls. The terminal text is as follows:

```
karin@se-pc32 ~/SpecSharp/Boogie/BoogieExamples
$ Boogie Example.bpl
Spec# Program Verifier Version 0.86, Copyright (c) 2003-2007, Microsoft.
Error on line 792
Example.bpl(792,5): Error BP5001: This assertion might not hold.
Spec# Program Verifier finished with 0 verified, 1 error
```

Figure 2.5: Boogie’s error report only contains the line number corresponding to the failed condition.

A Sound and Incomplete System

Boogie guarantees to be a **sound** system. This means that if the input program contains one or more failed assertions Boogie guarantees to find them.

However, Boogie is **incomplete**. Due to incompleteness Boogie might, even in a correct program, state failed assertions. This comes from underspecification. Boogie reports a program error and returns the corresponding failed condition whenever it has not been able to verify the condition.

Error Reporting in Boogie

Boogie’s error report consists of a line that references the failed condition. One problem, the programmer has to deal with, is Boogie’s incompleteness: Boogie’s error reports do not necessarily belong to a program error, but they can be spurious errors. Thus, in case of a failed condition returned by Boogie, the programmer can not imply a false condition.

Another problem, which arises especially in complex programs, is that the programmer might not be able to understand the source of the error by only getting the failed condition. If one does not understand the program, it may be difficult to detect and to fix the error.

Chapter 3

Program Slicer

The program slicer presented here is able to generate slices from a BoogiePL program in linear time. The way the slicer obtains the resulting slices is by including statements into the slice that might affect a slicing criterion. The rest is omitted. The included statements are obtained by graph traversing: All those statements are included into the slice that might affect the slicing criterion directly or indirectly.

Graph traversing is based on a *program dependence graph (PDG)*. A PDG represents all program dependencies of a program (Section 2.2). A program dependency is either a data dependency (Section 2.2) or a control dependency (Section 2.2).

Program dependencies are based on the *control flow graph (CFG)* (Section 2.1) because the CFG represents all possible execution paths, which are used to apply static analysis for dependency generation.

The CFG is obtained according the transfer commands, namely `goto` and `return` statements, occurring in each block.

This chapter gives a deep insight into the slicing process explained above, starting with establishing a CFG according to `goto` labels, generating a PDG based on the CFG, and ending with extracting the slice from the PDG. The chapter is organized as follows: Section 2.1 explains how to establish a CFG of a BoogiePL program according the transfer commands in the blocks.

So far program dependencies have only dealt with program statements. Establishing a PDG out of a BoogiePL program the question arises how to deal with assertions, such as `assume` and `assert` statements. Section 3.3 explains how assertions are handled in terms of PDG generation. Beside the assertions, Section 3.2 explains the remaining BoogiePL statements in terms of their data dependencies.

Sections 3.2 and 3.3 illustrate the dependencies of the basic statements, namely assignments (to local variables and to arrays), `havoc` statements, local variable declarations, `assume` and `assert` statements. The remaining statements, namely pre- and postconditions, procedure calls and statements including quantifiers, are derived from the basic statements. They are explained in Section 3.4.

3.1 Handling of `goto`'s

The CFG of a BoogiePL program is constructed according the transfer commands. The CFG represents the program blocks as nodes and the control flow as directed edges. Each block contains exactly one transfer command, which is either a `goto` or a `return` statement.

A `goto` command transfers control to one or more other blocks, depending on the block labels listed after the `goto`. A block containing a `goto` command has as successor blocks in the control flow graph all those blocks that are labeled after the `goto`. A `return` command indicates the end of a control flow path. A block containing a `return` command has no successors and represents the end of a control flow path.

To establish a CFG, the first executed block in the program is set as root. According its goto labels a CFG can now recursively be established by finding the successor blocks as long as goto commands can be obtained. The CFG is complete, when for each control flow path the return block is found.

3.2 Dependencies of Program Statements

A statement consists of two sets of variables, one set consisting of variables which are *defined* at the statement, the other set consisting of the variables which are *referenced* at the statement. In the following these two types of variables are called *defined variables* and *referenced variables*, respectively.

Generally can be said that every statement is data dependent on all those statements that have or, since the control flow is statically unknown, *might* have defined its referenced variable.

Assign Statements in General

To generate the relevant data dependencies of an assign statement, that might influence the concerning assignment, it has to be examined how referenced variables are influenced.

Generally it can be distinguished between assign statements assigning a local variable, and assign statements assigning an array element.

In case of assigning local variables, it is obvious which statements are responsible for the value of the referenced variable in the current statement. The defining statements can be found by considering all control flow paths leading to the current statement. For each of these paths, the statement that has most recently defined the value of the current statement has to be found. However, in case of assignments assigning array elements, where the value of the referenced array element might only be known at runtime, it cannot be examined statically which statements define the value of the referenced array elements.

The following two sections explain the data dependence generation of the two different assignment types.

Assignments of Local Variables

The question is raised, how can the data dependencies in program in Example 1 for the assignment $z := x + y$ in block *exit* be obtained? First we look at the possible execution paths:

- $entry \rightarrow a \rightarrow exit$,
- $entry \rightarrow b \rightarrow exit$,
- $entry \rightarrow c \rightarrow b \rightarrow exit$.

To generate the data dependencies of $z := x + y$ in *exit* we examine all control flow paths. For each path those statements that have most recently defined the referenced variables x and y have to be found.

- For the first control flow path consisting of the blocks $entry \rightarrow a \rightarrow exit$ the influencing statement for x is $x := w$ in block a , and the influencing statement for y is $y := w$ in *entry*.
- For the second control flow path consisting of the blocks $entry \rightarrow b \rightarrow exit$ the influencing statement for x is $x := 1$ in block b and the influencing statement for y is $y := w$ in *entry*.
- For the third control flow path consisting of the blocks $entry \rightarrow c \rightarrow b \rightarrow exit$ the influencing statement for x is $x := 1$ in block b and the influencing statement for y is $y := 1$ in block c .

Assignments of Array Elements

Considering assigned array elements and defining the dependencies similar to the previous case, the following problem will arise:

Example 4 (Array dependencies).

```

1 a[i] := x;
2 a[j] := y;
3 z := a[k];

```

Since the dynamic value of the array element assigned to `z` is only known at runtime the program slicer can not determine statically which variables are responsible for the referenced value `a[k]` of the concerning statement `z := a[k]`. Dependant on the values of the array indices, the following situations can occur:

- Variable `a[j]` would influence the concerning statement if $j \equiv k$.
- Variable `a[i]` would influence the statement if $i \equiv k$ and $j \neq k$.
- Variable `a[k]` would neither be influenced by `a[i]` nor by `a[j]` if $k \neq i$ and $k \neq j$.

The fact that there is often more than one possible statement on one control flow which *might* have an influence on a specific array element, leads to establishing multiple data dependencies on the same path. In addition, for all assign statements a dependency that defines the index variable is established.

Local Variables

Local variable declarations define the type of a variable. Additionally, the value of a local variable might be assumed using a `where` clause. The following program shows a possible slice generated from the program in Example 1 with respect to statement `assert z == 3`:

```

1 int w where w == 1;
2 int x;
3 int y;
4 int z;
5
6 entry:
7   assume x == 0;
8   assume y == 0;
9   assume z == 0;
10  y := w;
11  assume x == w;
12  goto a, b, c;
13
14 a:
15  x := w;
16  goto exit;
17
18 b:
19  x := 1;
20  goto exit;
21
22 c:
23  assert x == z;
24  y := 1;

```

```

25  goto b;
26
27  exit:
28  z := x + y;
29  assert z == 3; //Slicing criterion
30  return;

```

The variable declarations of the original program (**before** the slicing process) are:

```

1  int u where u == 1;
2  int w where w == 1;
3  int x where x == 0;
4  int y where y == 0;
5  int z where z == 0;

```

In order to slice the variable declarations, we keep only the declarations which declare variables that appear in the slice. Additionally, only the **where** clauses of variables whose value is relevant in the slice, are part of the slice.

According to that, the original variable declarations are sliced as follows: Variable **u** is not contained in the slice, which means that its declaration will not be part of the slice. Further, the **where** clauses assuming the initial values of **x** and **z** have no influence on the final value of **z** in the slicing criterion: **x** is redefined in all control flow paths leading to the slicing criterion and the initial value of **z** has no influence on the final value of **z** either. Thus, the **where** clauses of the local variables **x** and **z** are not part of the slice. Variable **y** is already redefined in the first statement. Since the first statement is contained in every control flow path its **where** clause can be omitted as well. In contrary, the slicing criterion depends on the value of **w** assigned by the **where** clause (used in the entry block in statement **y := w**). The **where** clause assigning the value 1 to **w** must therefore be included into the slice.

The sliced variable declarations are the followings:

```

1  int w where w == 1;
2  int x;
3  int y;
4  int z;

```

Havoc Statements

A **havoc** statement basically destroys the value of a variable by assigning a arbitrary value. A **havoc z** does not depend on any statement since the value of **z** is destroyed anyway, no matter what it was before. Therefore a **havoc** is neither control nor data dependent on other statements. The statement defines one or more variables but does not reference any variables. The data dependencies are generated analog to assign statements (see 3.2).

We look at block *exit* in Example 1:

```

1  havoc z;
2  w := z;

```

The value of **z** is set to an arbitrary value in statement 1. Therefore statement 2 is data dependent on statement 1. Statement 1 does not depend on any other statements.

3.3 Dependencies of Contracts

Contract dependencies turn up as soon as assertions, namely assume and assert statements are involved. This section shows how contracts depend on other statements, how they influence statements and how they depend on each other.

Assertions as Control Statements

As explained in Section 2.4 assertions can be considered as contracts to give a programmer the possibility to express the behavior of a program. An assertion control statements because it decides the further program execution: If an `assume` statement does not hold the program is no longer controlled and if an `assert` statement does not hold the program execution fails. For this reason, both, assume and assert statements, could be considered as *control statements* if we were interested if the program terminates or not. Since we are only interested in the values of variables in a certain statement if the program execution really gets to that point, we assume the contracts to hold do not treat them as control statements but as defining statements. This means, that contracts determine the value of subsequent statements in the control flow.

Assertions and Data Dependency

There is a data dependency between an assertion and the (target) statement pointing to by the assertion if

- the referenced variables in the target statement are contained as free variables in the assertion, and
- the two statements are in the same state space.

Dependencies can also arise among assertions in the same state space, which is explained later in this section.

State Space of Assertions

Assertions are in the same state space in program if none of the values of any variable has been changed. The following two program extractions both show two assume statements that are obviously in the same state space:

```

1  assume x == y ;
2  assume y == 0 ;

1  assume y == 0 ;
2  assume x == y ;

```

The extractions show the same assume statements in different order. Obviously, the assume statements within the same program influence each other: From both program extractions the assertion `assume x == 0` can be derived. It can generally be said that

- assume and assert statements can influence each other and
- the statements within a state space can be in any sequential order without changing the semantics of the program.

Data Dependencies

Assertions (assume and assert statements) depend on statements appearing in the control flow paths leading to the respective control statement. The variables of an assertion are considered as referenced variables. An assertion is data dependent on those statements that have, for each path in the control flow, most recently defined the value of its referenced values.

Example 5.

```

1 ...
2 x := 0;
3 y := 0;
4 w := 0;
5 assume x == w;
6 z := w;
7 ...

```

In Example 5 the assume statement in line 5 is data dependent on the assign statements on line 2 and 4 since the referenced variables `x` and `w` are defined in line 2 and 4 and there are no redefining statements for the variables `x` and `w` in between.

On one hand, assertions *depend* on statements as explained before, on the other hand, they *influence* other statements. In Example 5, the statement on line 6 is data dependent on the assume statement on line 5 because the referenced value of `w` in line 6 is determined by the expression in the assume statement `x == w` referencing `w` (and `x`).

In case of only one assertion (Example 5) the data dependencies are quite obvious. Dependencies get much more interesting if there is more than one assertion: Assertions appearing in a sequence.

Example 6.

```

1 ...
2 x := 0;
3 y := 0;
4 w := 0;
5 assume x == y;
6 assume w == x;
7 assume w == 0;
8 z := y;
9 ...

```

On which statements do the assume statements depend? Obviously, the statements can be treated similar to the previous example. By finding the statements which have most recently defined the referenced values of the control expressions, the following data dependencies can be obtained:

- `assume x == y` is data dependent on `x := 0` and `y := 0`,
- `assume w == x` is data dependent on `w := 0` and `y := 0`,
- `assume w == 0` is data dependent on `w := 0`.

Which statements are determined by the assume statements? At the first look, statement `z := y` on line 8 does not depend on the assume statement `assume w == 0`, neither does it depend on `assume w == x` because the referenced variable `y` is not referenced by any of the variables occurring in these two assume statements. It seems as if the only defining statement for `y` in the assignment `z := y` is the assume statement on line 5, `assume x == y`.

Statement `assume x == y` is data dependent on the assume statement on line 6, `assume w == x` because it gives more information about the value of `x`. Further, `assume w == x` is data dependent on the assume `w == 0`, which finally defines the value of `y` as 0. By generating the mentioned transitive dependencies the value of `y` has finally been defined.

This example shows how assertions in the same state space influence each other. Thus, for a referenced variable it has to be iterated through the set of assertions in the current state space in order to decide whether or not an assertion influences the according variable.

Iteration Process

In this section the iteration process of finding all assertions within a state space, which has been introduced in the previous section, is explained in detail. The goal of the iteration process is to extract those assertions of the original set that directly or indirectly influence a specific variable. In a first step, all assertions in the same state are merged into one set. In a second step, the iteration process for each set can be executed. In the following example the goal of the iteration process is to find all those assume statements that directly or indirectly determine the referenced variable (here y). The iteration process for Example 6 is executed as follows:

- Remaining set: $\{\text{assume } w == x, \text{assume } x == y, \text{assume } w == 0\}$,
- referenced variables: $\{y\}$,
- current set: $\{\}$.

Slicing according to variable y means traversing iteratively through the set of assume statements. Since the assume statements with a state space can be in any sequential order the slicer has to go iteratively through the set finding all directly and indirectly influencing dependencies until a fix point is reached. Starting at the first element in the set, $\text{assume } w == x$, no dependency on variable y can be detected. In contrary, the second set element, $\text{assume } x == y$, declares y and will therefore be part of the resulting set of assumes. Additionally a new referenced variable, namely x , is included in the set of referenced variables:

- Remaining set: $\{\text{assume } w == x, \text{assume } w == 0\}$,
- referenced variables: $\{y, x\}$,
- current set: $\{\text{assume } x == y\}$.

The last statement of the remaining set, $\text{assume } w == 0$, only defines the variable w , which is not in the set of referenced variables and therefore does not give any information about the slicing variable y yet. It is therefore not included in the current set at this moment. The next step in the iteration process goes back to the first element in the set of the remaining statements. For $\text{assume } w == x$ can easily be obtained that this statement will now give more information about y because it defines x (which has currently been included in the set of referenced variables). Statement $\text{assume } w == x$ is therefore included into the slice, as well. Additionally, w is included into the set of referenced variables:

- Remaining set: $\{\text{assume } w == 0\}$,
- referenced variables: $\{y, x, w\}$,
- current set: $\{\text{assume } x == y, \text{assume } w == x\}$.

Now it is known that both, $\text{assume } x == y$ and $\text{assume } w == x$, control the referenced variable y . From that can be concluded that the remaining statement, $\text{assume } w == 0$, defines the value of y indirectly as 0. Therefore the remaining statement has to be included, as well.

- Remaining set: $\{\}$,
- referenced variables: $\{y, x, w\}$,
- current set: $\{\text{assume } x == y, \text{assume } w == x, \text{assume } w == 0\}$.

The iteration process is aborted as soon as the the remaining set is empty or a fix point has been reached. The fix point is reached, if no variable occurring in the set of referenced variable can be obtained as referenced variable in the remaining set of assumes.

Dependencies of multiple Variables

The previous example has illustrated the iterating process to generate a set of assume statements determine a specified variable. When generating a complete program dependence graph, the iteration process has to be executed for every referenced variable that might be dependent on the set of assume statements. After determining the set of statements for every variable, the resulting set of statements is obtained by unifying the single sets.

```

1 ...
2 assume x == y;
3 assume w == x;
4 assume w == 0;
5 assume a == b;
6 assume c == 0;
7 z := y;
8 z := a;
9 ...

```

- Remaining set: {**assume** w == x, **assume** x == y, **assume** w == 0, **assume** a == b, **assume** c == 0},
- referenced variables: {a, y},
- current set: {}.

Iteration process:

1. Generate determining set for variable a: {**assume** a == b}
2. Generate determining set for variable y: {**assume** w == x, **assume** x == y, **assume** w == 0}
3. Merge sets: {**assume** a == b, w == x, **assume** x == y, **assume** w == 0}

Handling of Assert Statements

The examples in the previous sections have illustrated the handling of assume statements in terms of data dependencies. In order to examine the dependencies of assert statements we take a look at the control flow path containing the blocks

entry → *c* → *b* → *exit*

in the program in Example 1.

```

1 ...
2 b:
3   x := 1;
4   goto exit;
5
6 c:
7   call z := Max(u, w);
8   w := 2;
9   x := 2;
10  z := 2;
11  assert x == z;
12  y := x;
13  y := 1;
14  goto b;
15 ...

```

How does the assert statement `assert x == z` in block *c* influence the variable *x* in `y := x`? We can assume that, if the program at some point reaches the statement `y := x`, the assert statement must have held. Therefore the assert statement must be true in that case and the expression in the assert statement can be assumed to be true. Due to the fact that `x==z` can be assumed, the variable *x* in `y := x` is influenced by the assert statement and therefore it exists a data dependency.

Generally can be said that all assert statements on the execution path leading to a certain statement are assumed to hold. Otherwise the execution would not have reached the statement. Therefore all assert statements on the execution path can be assumed to be true.

As known easily can be obtained, assert statements behave similar to assume statements in terms of data dependency generation. Assert statements can therefore be handled like assume statements.

3.4 Extended Statements

The program dependencies of the following statements can be derived from the basic statements explained in the previous sections.

Calls

A call is basically data dependent on those statements, that define the variables given as input to the called procedure. If the value(s) returned by the called procedure define(s) one or more variables then the call might additionally have influence on other statements in terms of data dependency. Dealing with calls we further have to take into account that the pre- and postconditions of the called procedure, which are desugared assume and assert statements, cause additional assume and assert statements in the current procedure.

Example 7 (Procedure call). The original program in Example 1 calls procedure `Max` in block *c*. The corresponding procedure `Max` is defined here:

```

1 c:
2   call z := Max(u, w);
3   ...
4   goto b;
5
6
7 procedure Max(int: i, int: g j)
8 requires i != j;
9 requires 0 < i;
10 ensures 0 < result;
11 ensures result == i || result == j;
```

In order to guarantee the proof obligations of the called procedure `Max` the `requires` clauses (the `ensures` respectively) are desugared into assert statements (assume statements respectively).

The following sequence of statements shows block *c* after translating the pre- and postconditions of procedure `Max` (Example 7) into assume and assert statements:

```

1 c:
2   assert u != w;
3   assert 0 < u;
4   call z := Max(u, w);
5   assume 0 < z;
6   assume z == u || z == w;
7   ...
8   goto b;
```

Considering the input variables of as *referenced* variables of the call and the assigned variables as **defined variables** the data dependencies of a call statement can generally constructed analog to assign statements explained in Section 3.2.

Pre- and Postconditions

Preconditions and postconditions in a BoogiePL program can basically be treated as assume and assert statements. All preconditions have first to be translated into assume statements. Afterward they are inserted at the beginning of the program, which is at the beginning of the unique entry block. Postconditions have to be translated into assert statements and inserted at the very end of the program, which is at the end of a return block.

Quantifiers

Assertions containing quantifiers can be considered as usual assume and assert statements. Example 6 shows a sequence of assume statements all of which containing an equality as expression. The following example slightly differs from Example 6: The expression on line 7 has been replaced by a quantifier expression.

```
1 ...
2 x := 0;
3 y := 0;
4 w := 0;
5 assume x == y;
6 assume w == x;
7 assume (forall i :: w == i); \\ Quantifier expression
8 z := y;
9 ...
```

To generate the data and control dependencies of a quantifier expression first the referenced variables have to be determined. The referenced variables of a quantifier statement are the free variables in the expression. After extracting the referenced variables quantifier expressions can be handled similar to assume and assert statements.

Chapter 4

Implementation

This chapter provides some implementation details of the program slicer, whose main functions are described in Chapter 3. The program slicer has been implemented as part of Boogie. Section 4.1 describes the overall idea of the implementation of the slicer. The subsequent sections describe the important parts of the implemented program slicer, which are the block nodes, statement nodes and the process of slice generation.

4.1 Control Flow Graph Generation

Block Nodes

As described in detail in Section 2.1 a *control flow graph (CFG)* consists of block nodes, each node representing a block. The implemented control flow graph consists of one entry block of type `EntryNode` and one or more exit blocks of type `ReturnNode`. Both, `EntryNode` and `ReturnNode`, extend the type of the basic block, the `CFNode` (control flow node).

CFG Generation

The CFG of blocks of type `CFNode` is established according to the labels in the specific transfer statements, which are part of each block (Section 3.1). A block containing a `return` statement as transfer command is of type `ReturnBlock` and has no successors. The remaining blocks contain a `goto` command, which specifies one or more successor blocks listed as labels after the `goto`. A `CFNode` represents a block and contains a list of all successor nodes representing the successor block of the corresponding block. To generate the control flow graph of the program, starting with the entry node the Slicer looks recursively for all successors of the current block according to the labels, generates a `CFNode` and adds its successor blocks as nodes to its internal list of successor nodes. It stops when each control flow path has reached a block containing a `return` command as transfer statement.

Algorithm 1 (Control flow generation).

```
1 CreateSuccessors(currentBlock){
2   foreach (successorBlock of the currentBlock){
3     Add successorBlock to successorList of currentBlock;
4     if (successorBlock has return command){
5       //break
6       return;
7     }
8     else {
9       //recursive call
```

```

10     CreateSuccessors (successorBlock);
11   }
12 }
13 }

```

4.2 Program Dependence Graph Generation

The construction of the *program dependence graph (PDG)* consists of program dependence generation between nodes.

Program Dependence Nodes

All program statements, such as assignments, calls, havoc, assume and assert statements, are implemented as PDNodes (program dependence nodes). The construction of data dependencies is established according to the data dependencies described in Sections 3.2 and 3.3. The process of finding all data dependencies in a program is done by finding the influencing statements for each node. All PDNodes that might directly influence another PDNode are finally pointing to its respective node.

Assignment Nodes

The dependencies between nodes representing an assign statements are described in Section 3.2 in detail. Assign statements referencing a local variable are treated in a different way than assign statements referencing an array elements.

The data dependencies of assign statements are found recursively. For each node, the potentially influencing assignments are found.

To establish all data dependencies of a PDNode representing an assign statement, Algorithm 2 is executed for every assign statement in the program.

Algorithm 2 (Data dependence generation).

```

1 CreateSuccessors (inputStatement)
2
3 if (inputStatement references a local variable){
4   Lookup in the current block for the most recently defining statement
5   ;
6   if (defining statement found in the current block){
7     Create dependency;
8     return;
9   }
10  else {
11    // No statement found in the current block
12    foreach trace in the CFG leading to inputStatement {
13      Lookup for the most recently defining statement in a previous
14      block;
15      Create dependency;
16    }
17    return;
18  }
19 }
20
21 else if (inputStatement references an array element){
22   Lookup in the current block for all defining statements;

```

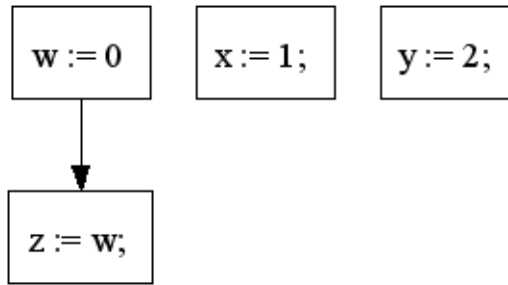



Figure 4.1: The value of z only depends on statement $w := 0$.

```

21  if (defining statement found in the current block){
22      create dependency;
23      // No return because lookup goes on anyway
24  }
25  //No else clause because the CFG is traversed anyway
26  foreach trace in the CFG leading to inputStatement {
27      Lookup for the all defining statements in all previous blocks;
28      Create dependency for all defining statements;
29  }
30  return;
31 }

```

Example 8 (Local variable dependency).

```

1 w := 0;
2 x := 1;
3 y := 2;
4 z := w;

```

The local variable z depends on the statement which has most recently defined w (Figure 4.1).

Example 9 (Array dependency).

```

1 a[w] := 0;
2 a[x] := 1;
3 a[y] := 2;
4 a[z] := a[w];

```

The array element $a[z]$ depends on all statements that have ever defined an array element of the array a (Figure 4.2).

Assertion Nodes

The goal is to find those assertions of a program, which might have influence on the slicing variable. Assertions following directly one after the other are in the same state. The order of assertions does not matter. Each set of assume resp. assert statements in the same state represents one node, namely an assertion node. An assertion node is either an `AssumeNode` or an `AssertNode`. Each assertion node consists of a set of assertions which is either a set of assume statements or a set of assert statements. A slice of such a set with respect to a given slicing variable is found by determining all the assertions within the set, which might affect the given variable. To find all dependencies, transitive dependencies within the set have to be considered as well. A sliced set of assertions can basically be found by iterating and finding all relating assertions recursively until a fix point is reached (as described in Section 3.3).

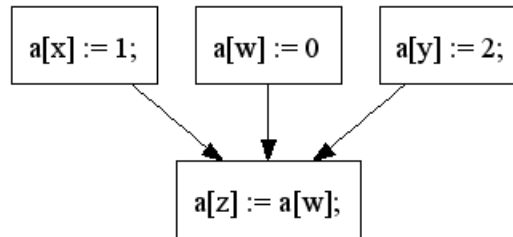


Figure 4.2: The value of $a[z]$ depends on all statements on the control flow trace to the current statement defining a .

4.3 Using the Tool

The program slicer takes as input a program statement and a program and slices the program with respect to the input statement. It can be used for debugging a failed condition by giving as input the failed condition returned by Boogie and the failed program. The program slicer slices the failed program with respect to the failed assertion, which helps the programmer to understand and fix the error.

New command line options are introduced:

- `/showSlice`
- `/slicingLevel`
- `/slicingLine`
- `/slicingVar`

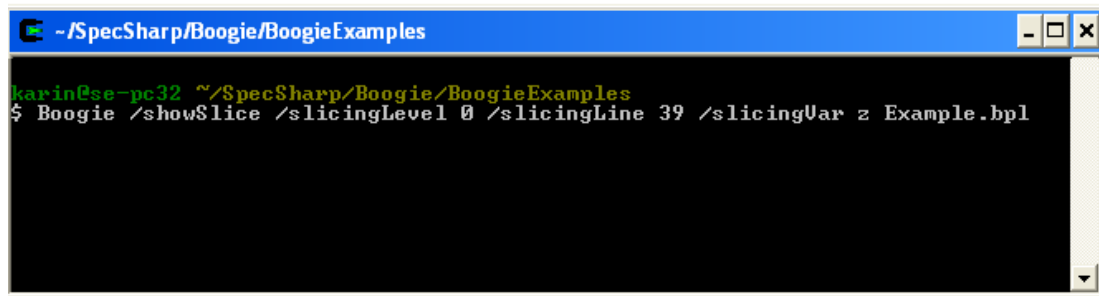
showSlice takes a BoogiePL program as input and generates a slice with respect to the input statement.

slicingLine takes an integer parameter as input specifying the slicing statement.

slicingLevel can be set to 0 or 1. It specifies the slicing approach:

- Slicing level 0 generates a slice using static flow analysis
- Slicing level 1 slices with the help of the static verifier in a dynamic way.

Figure 4.3 shows an example of the slicing command.

A terminal window with a blue title bar containing the text `~/SpecSharp/Boogie/BoogieExamples`. The terminal content shows a shell prompt `karin@se-pc32` followed by the command `~/SpecSharp/Boogie/BoogieExamples` and the Boogie command `$ Boogie /showSlice /slicingLevel 0 /slicingLine 39 /slicingVar z Example.bpl`.

```
~/SpecSharp/Boogie/BoogieExamples
karin@se-pc32 ~/SpecSharp/Boogie/BoogieExamples
$ Boogie /showSlice /slicingLevel 0 /slicingLine 39 /slicingVar z Example.bpl
```

Figure 4.3: Command to generate a slice of the program Example. Using static analysis (slicing level 0) with respect to variable `z` in the statement on line 39.

Chapter 5

Using the Theorem Prover in the Presence of Arrays

5.1 Handling of Arrays using Static Flow Analysis

Section 3.2 describes the static dependencies of array elements. Using static flow analysis to determine data dependencies of array elements, an array update generates an update on every array element in the corresponding array. This solution is obviously not very efficient since it might generate far more dependencies than actually needed. Example 9 in the previous chapter has dealt with array dependencies:

```
1 a[w] := 0;  
2 a[x] := 1;  
3 a[y] := 2;  
4 a[z] := a[w];
```

The critical point here is that it can not be decided whether the two array elements $a[w]$ and $a[y]$ on line 3 and 4 are equal or not, when using static flow analysis. Thus, the PDG generator is forced to establish a potential dependency. This obviously leads to redundancy in the PDG.

The variable values of w and y influence the array elements as follows:

- $w \equiv y \Rightarrow a[w] \equiv a[y]$ and
- $w \neq y \Rightarrow a[w] \neq a[y]$.

Equality of w and y guarantees that the referenced variable $a[w]$ in line 4 is equal to the defined variable $a[y]$ in line 3. This means that statement 4 depends definitely on statement 3. Knowing that two array elements are equal would obviously help to avoid redundant data dependencies. However, whether $a[w]$ and $a[y]$ on line 4 and 3, respectively, in Example 9 are equal or not can not be determined using static flow analysis. Therefore an array update can potentially influence all referenced array elements occurring in subsequent statements on the control flow path. On the other hand, a referenced array element can potentially depend on all array updates of the same array on the control flow path: If we look at Example 9 then the referenced array element $a[w]$ on line 4 can potentially be defined in line 1 or in line 2 or in line 3. Therefore a data dependency from each array update on the control flow leading to $a[w]$ on line 4 has to be defined. This, of course, leads to redundant data dependencies, which, considering the slicing process, further results in complex slices approaching the size of the original program.

5.2 Using the Theorem Prover

Boogie's theorem prover is responsible for proving passive commands (Section 2.5). As we have seen in the previous section knowing the fact that two array elements are equal would help to

avoid redundant data dependencies.

The idea is to take advantage of Boogie’s theorem prover and make it verify if two array elements are equal. When giving an assertion containing an equality of two array elements to the theorem prover it will return a failed condition if it has not been able to verify the assertion. In the other case, when a failed condition is returned, Boogie guarantees that the assertion is true, which implies equality of the two array elements.

To optimize the PDG in the presence of arrays, in particular to get rid of unnecessary array dependencies, an assertion containing an equality of the indices of two array elements is given to the theorem prover as input. A data dependency between the two concerning statements is established or not according to the boolean output of the theorem prover: If the assertion does not fail we can assume that the array elements are equal. Equality of the two array indices implies a data dependency and previous assignments to array elements of the same array can be omitted. However, a failed assertion does not imply that the two indices are not equal. The failed assertion could also be a spurious error (Section 2.5). Thus, the potential dependency must not be omitted. In order to establish all potential dependencies further dependencies have to be found.

In order to take advantage of the theorem prover provided by Boogie, Algorithm 2 in Section 4.2 is extended to Algorithm 3.

Algorithm 3 (Data dependence generation with the help of the theorem prover).

```

1 CreateSuccessors(inputStatement){
2   if (inputStatement references a local variable){
3     ...
4   }
5   else if (inputStatement references an array element){
6     Lookup in the current block for the most recently defining
7       statement;
8     if (defining statement found in the current block){
9       //Use help of the theorem prover to check array elements on
10        equality
11       Create assertion;
12       Make the theorem prover verify the assertion;
13       if (assertion has not failed){
14         //array elements are equal
15         Create dependency; //Definitive dependency
16         return;
17       }
18       else{
19         Create dependency; //Potential dependency
20         //No return because lookup goes on
21       }
22     }
23     foreach trace in the CFG leading to inputStatement {
24       while (No definitive dependency is found){
25         Lookup for the definitive and potential defining statements in
26           the previous blocks;
27         Create dependency for all definitive and potential defining
28           statements;
29       }
30     }
31   }
32 }

```

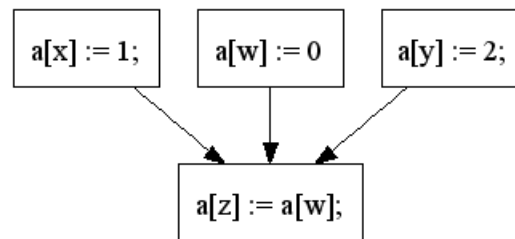


Figure 5.1: Array dependencies using static flow analysis contain redundant dependencies.

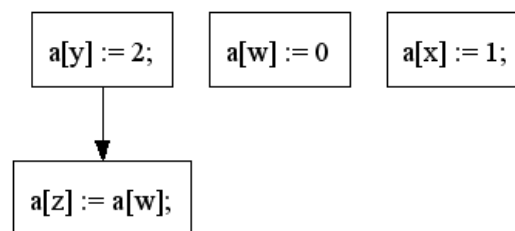


Figure 5.2: Dependencies in case of equality of $a[w]$ and $a[y]$: With the help of the theorem prover redundant dependencies as they occur in Figure 5.1 can be omitted.

Chapter 6

Conclusion

This thesis has presented a program slicer for BoogiePL programs. In addition to traditional program slicing applications our slicer is able to slice programs including contracts. The slicer generates efficient slices for programs consisting of scalar variables. However, as in many slicing techniques, slices in the presence of arrays tend to reach the size of the original program. Since BoogiePL programs in practice mainly consist of heap statements a method had to be found, to generate compact slices in the presence of arrays, in order to turn the slicer into a useful tool in practice. With Boogie's theorem prover we have found a powerful resource to avoid redundant dependencies. The technique of program slicing in combination with the theorem prover has turned program slicing into an attractive technique to generate efficient slices even in the presence of arrays.

As future work we plan an extension of the static program slicer to include the theorem prover. Furthermore, the slicer is planned to be extended in that way, that it generates slices automatically from the failed condition.

Bibliography

- [1] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
- [2] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
- [3] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.
- [4] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.
- [5] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.