University of Minho
Informatics Department

Master Course in Informatics Engineering

GamaBoogie

A Contract-based Slicer for Boogie Programs

Márcio João Carvalho Coelho
Informatics Department - University of Minho

Supervised by:

Professor Pedro Henriques Rangel, University of Minho
Professor Daniela Carneiro da Cruz, University of Minho

Braga, 28th December 2011

Acknowledgements

Sometimes there is the need to have something in your to help you to identify who are the real friends you have that are always by your side. Would be impossible for me to finish this work, with these results, if I did not have special friends like I have. I am grateful for that and I want to express my deepest feeling of gratitude to my friends and family, one by one.

To Professor Pedro Rangel Henriques, I am very grateful for all the support and help given through these two years of work, and for the great contribution for my academic results. Muito Obrigado por tudo Professor.

I would like to thank Professor Daniela da Cruz for her enormous help in the development of the tool. Without her support I would not be able to finish the tool. It is impossible for me to quantify all the help she gave me in this work. Muito Obrigada Professora Daniela.

To Professor Jorge Sousa Pinto, for all the insights. Muito Obrigado Professor Jorge.

To Nuno Oliveira, for his presence and support in carrying out my tasks. For his advice, suggestions and concerns. He has always a kind word to say, and I will never forget the funny moments we had in that lab. Obrigado Nuno.

To Tiago Veloso, for his work with this latex template. Obrigado Tiago.

I would also like to thank my team mates from the 2004/2005 class. For all the great experiences that we had together, and for the unforgettable moments we had during this long walk. I will never forget you guys. Obrigado pessoal.

To my best friend Sofia Fernandes, who always believed in me. Without you it would not be the same person I am today. I will not thank you for existing but for being by my side since the beginning. Muito Obrigada Sofia.

Last, and not least, to my parents and brother for the unconditional support, patience and motivation that helped me dealing with the stress. They have never gave up of supporting me along these seven tough years.Pai, Mãe e Rui, Obrigado por tudo.

Abstract

In the context of the Informatics Engineering MSc. degree (MEI), second year, this document describes and discusses a master thesis project in the area of source code analysis using slicing and program verification techniques.

Design-by-Contract *is an approach that allows a programmer to specify the expected behaviour of a component by the means of preconditions, postconditions and invariants. These annotations (or contracts) can be seen as a form of enriched software documentation and they are used to verify that the program is correct with respect to these contracts. On one hand, slicing the program to extract the code that is relevant for the contracts reduce the size of the program and improves its verification. On the other hand, using the contract to slice a program in a finer, more sensible semantic way, enables to optimise the code reducing it to the minimum necessary to keep the postcondition true.*

Following the last research direction, Daniela da Cruz introduced in her Ph.D. thesis the concepts of Assertion-based Slicing and Contract-based Slicing to explore (at an intra-procedural or inter-procedural level) the optimisation of code according to the semantics expressed by contracts. Her approach is based on the source code analyses and her slicing uses annotations present in the procedures.

This thesis focuses on the study of BoogiePL language and its use as an intermediate representation for annotated programs in order to build slices of Boogie programs (instead of slicing the source code). Boogie compiler is used to generate the verification conditions in SMT just for the computed slice; these conditions can be then passed to the Z3 Prover to ensure that all contracts are preserved when invoking annotated procedures. The final objective is to make possible the comparison between the SMT code produced by this approach and the code currently generated by GamaSlicer, expecting to obtain a more efficient solution.

To implement that idea, a tool called GamaBoogie was developed. That tool, at the end, offers more than its main functionality in slicing and verifying. Actually it allows to inspect and visualise boogie programs; this functionality seems to be very useful for Boogie program comprehension.

Resumo

No contexto do Mestrado em Engenharia Informática (MEI), segundo ano, este documento é uma dissertação que descreve e discute um projeto de tese de mestrado na área de análise de código fonte utilizando técnicas de slicing *e de verificação de programas.*

Design-by-Contract *é uma abordagem que permite ao programador especificar o comportamento esperado de um componente por meio de pré-condições, pós-condições e invariantes. Estas anotações (ou contratos) podem ser vistos como uma forma de enriquecer a documentação do software e são usados para verificar se o programa está correto com relação a esses contratos. Por um lado, fazer o* slicing *do programa para extrair o código que é relevante para os contratos, pode reduzir o tamanho do programa e melhorar a sua verificação. Por outro lado, usando o contrato para fazer o* slicing *de um programa de forma mais sensível à semântica, permite otimizar o código reduzindo-o ao mínimo necessário para manter o contrato válido.*

Seguindo a direção da última pesquisa, Daniela da Cruz introduziu na sua tese de doutoramento os conceitos de Assertion-based Slicing *e* Contract-based Slicing *para explorar (a nível intra-procedimento ou inter-procedimento) a otimização de código de acordo com a semântica expressa por contratos. A sua abordagem é baseada na análise de código fonte e o* slicing *aplicado a tais programas utiliza anotações presentes nos procedimentos.*

Esta tese centra-se no estudo da linguagem BoogiePL e no seu uso como uma representação intermédia para programas anotados a fim de construir slices *de programas Boogie (em vez de fazer o* slicing *ao nível do código fonte). O compilador Boogie é usado para gerar as condições de verificação em SMT apenas para o* slice *calculado; essas condições podem ser passadas para o Prover Z3 para garantir que todos os contratos são preservados quando invocados os procedimentos anotados. O objetivo final é tornar possível a comparação entre o código final produzido por esta abordagem e o código atualmente gerado pelo GamaSlicer, com a expetativa de obter uma solução mais eficiente.*

Para implementar essa ideia, uma ferramenta chamada GamaBoogie foi desenvolvida. Essa ferramenta, no final, oferece mais do que sua funcionalidade principal de slicing *e verificação. Na verdade, permite inspecionar e visualizar programas Boogie; esta funcionalidade demonstrou ser útil para a compreensão de programas Boogie.*

Contents

List of Figures

List of Listings

List of Acronyms

BoogiePL Boogie Program Language. 4, 8, 15, 61

CFG Control Flow Graph. v, 41–43, 49

CIL Common Intermediate Language. 8

CS Conditioned slicing. 27

DAG Directed Acyclic Graph. v, 41, 42, 49

DS Dynamic slicing. 26

DbC Design-by-Contract. 6, 29

JML Java Modeling Language. 3, 6, 7, 9

LCFG Labeled control flow graph. 33, 34

PV Program Verification. 3, 5, 6, 61

SMT Satisfiability Modulo Theories. 8, 11, 14

SS Static slicing. 26

VCC Verifier for Concurrent C. 6, 8

VCGen Verification Condition Generator. 3, 7, 11, 33

VC Verification condition. 11, 20, 53

WLP Weakest liberal precondition. 20, 50, 52

WP Weakest precondition. 11, 20, 52

Chapter 1

Introduction

Nowadays, more than ever, there is a strong concern to verify software program because it is vital for society. Hospitals, banks, governments and industries use software systems that can not have errors. Such errors can have drastic economic and human consequences. In the past (before the year 2000) the aerospace industry lost over a billion dollars due to severe bugs in the software [61].

There is a way to ensure that a software satisfies its functional specification by constructing a mathematical proof. A proof of correctness should mainly detect if a program is inconsistent with respect to its assertions (this is, if it will not perform the intended task). However, a proof itself can be erroneous. As a mathematician can err in formulating a proof, a program prover can make a similar mistake. The use of verification tools can help in reducing this kind of errors, such as Java Modeling Language (JML) [34] and Spec♯ [6].

Of course, the use of software verification tools cannot guarantee the absence of errors but can avoid certain kind of errors; sometimes these tools can go further by allowing the programmer to derive parts of a program automatically from logical specifications. These specifications provide a representation of the program's intended meaning or goal; they say what the program is supposed to do rather than how to do it. Thus they are easier to understand than the details of the program itself, and can be directly accepted or rejected depending on whether they match or fail to match the user requirements of the program.

The dream of ideal software development (without errors) recalls for the use of Program Verification (PV) techniques [51]. Due to the need expressed by those companies that use safety critical systems, some software houses decided to specialise in PV instead of following with software development. The idea of of applying deduction to programs goes back to the late 1960s [5]. With PV it is possible to say if a program is correct for a given specification.

A VCGen is a program that reads a piece of code together with a specification and produces a set of proof obligations whose validity implies the partial correctness of the code with respect to its specification. This set will be passed to a prover and the program is correct if the proof obligations are valid.

Slicing was first proposed in 1979, by Weiser in his PhD thesis [66] and was then used to ease program debugging, software testing, software metrics, software maintenance, program comprehension and so on. Slicing can be use to extract the statement relevant to a particular computation.

The Slicing executed in GamaSlicer is done in the source program. Several program verifiers of .NET use an intermediate verification language. So if the Slicer is executed in this intermediate level, between the source program and the Verification Condition Generator (VCGen), we must be able to reduce the verification time.

GamaSlicer is a tool designed by Daniela da Cruz [24], that have a VCGen and a Slicer that allows to slice programs annotated with pre and postconditions in Java+JML.

Boogie is an intermediate language for program analysis and program verification. The lan-

guage is a typed imperative language with procedures and arrays. Boogie Program Language (BoogiePL) [29] can be used to represent programs written in an imperative source language (like an object-oriented .NET language) and is accepted as input to Boogie, the Spec♯ static program verifier.

GamaBoogie will work with Boogie language, where will be implemented the Contract-based Slicing [25] algorithm and then the Boogie Compiler will be responsible for generate the SMT code.

1.1 Objectives

In this context, the objectives of this master thesis work are :

1. Study the different areas involved, namely program verification and semantic slicing.
2. Study Boogie system (architecture and functionalities) and Boogie Programming Language.
3. Develop a tool for slicing Boogie programs.

1.2 Outcomes

To accomplish the objectives above a deep study of Boogie system and BoogiePL was done, aiming at using this language as an intermediate representation for annotated programs.

A tool called GamaBoogie was developed to slice the intermediate Boogie program. That tool generates the verification conditions in SMT just for the computed slice; these conditions can be then passed to the Z3 Prover to ensure that all contracts are preserved when invoking annotated procedures.

Moreover, GamaBoogie offers more than its main functionality in slicing and verifying. Actually it allows to inspect and visualise boogie programs; this functionality seems to be very useful for Boogie program comprehension.

1.3 Outline

The state-of-art concerning Program Verification using VCGens appears in Chapter 2 and Chapter 3.

Chapter 4 introduces Boogie framework, i.e. Boogie language and program verifier.

In Chapter 5, the concept of slicing is presented and several variants are defined. The special case, crucial for this project, of slicing programs with contracts is introduced in Chapter 6; there Assertion-based Slicing is defined and the tool GamaSlicer is described.

Chapter 7 gives all details about GamaBoogie, the main contribution of the present Master Thesis.

This dissertation closes in Chapter 8 with some conclusions and future work.

Chapter 2

Program Verification

Program Verification (PV) is a technique that ensures that a given program is correct for a given specification. This technique appeared in the early 1970s; after a long period almost in standby, it became more and more active in the last decade [51].

Along the years, Software Verification was applied to different contexts and with different purposes. Some of these contributions are:

Reliability higher assurance in program correctness by automated proofs.

Developer productivity feedback on errors while typing program.

Refactoring check that suggested program transformations preserve program meaning (e.g. refactoring using Type Constraints; automated support for program refactoring using invariants).

Program Optimisations automatically transform program so that it takes less time and memory power.

2.1 Software verification Techniques

Verification techniques can be divided into two major groups:

- *dynamic* verification
- *static* verification

2.1.1 *Dynamic* verification

Dynamic verification techniques refer to techniques that check the software for bugs during the execution of software.

This phase of software verification is also know as *Testing* or *Experimentation*, because it involves the execution of a system or a component. Basically, a number of test cases are chosen, where each test cases consist of test data. The goal of this phase is to check that a software:

- meets the business and technical requirements that guided the design and development of the application; and
- has the correct behaviour (works as expected).

Although the tests phase can be performed at any time in the development process, depending on the testing method employed, most of the effort relies after the requirements have been defined and the coding process has been completed. Test Driven Development differs from traditional models, as the effort of testing is on the hands of the developer and not in a formal team of testers.

Testing methods

There are two main methodologies: *white-box* and *black-box*. These approaches describe the point-of-view taken by a test engineer when designing the test cases.

Black-Box *Black box is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions* [11].

This method takes into account the inputs for an application and what are the expected results outputs for each one of these inputs.

One of the advantages of this method is its ease to use, as testers do not have to concern/know the implementation code.

White-Box *White box testing is testing that takes into account the internal mechanism of a system or component* [11].

Also known as *glass-box* testing because the testers have knowledge of the internal code.

Essentially, the purpose of white box testing is to cover testing as many of the statements, decision point, and branches in the code base as much as possible.

2.1.2 Static verification

Static verification techniques refer to those that inspect the code before its execution. They are also known as *Analysis* techniques. They are useful for proving correctness of a program.

The group of static verification techniques can also be divided in two groups: the *manual or semi-automatic techniques* and the *fully automated techniques*.

Techniques requiring substantial manual effort, like interaction with tools that construct a proof; techniques that use refinement steps to construct programs from specifications; and techniques that require annotations from the programmer fall in the first group *Manual/Semi-automated Techniques*. Techniques such as (bounded) model checking and (abstract) static analysis fall in the second group Fully Automated Techniques.

PV process is applied to a program annotated with a specification.

To implement verification the verification process, a tool (a language processor) is required. This tool recognises and transform the annotations. To verify automatically a program, the specification should be written in a formal language and added to the program.

In this context, Design-by-Contract (DbC) has appeared in 80's [54], this approach allows developers to specify programs using pre- and postconditions, loop invariants and additional assertions; these logic statements are embedded in the source code in the form of annotations.

Nowadays, there are some languages that support these kinds of annotations: a subset of Pascal [37] from 1986; Spec♯ [6]; Verifier for Concurrent C (VCC) [59, 26]; Java Modeling Language (JML) Esc/Java [34]; KeY [4] and Code Contracts [33, 32]

2.2 Manual/Semi-automated Techniques

There are two tasks involved in Manual/Semi-automated Techniques: reason about the program structures, transforming them to classical logic expressions; and reason about the data types. To handle these tasks, there are two ways: using a tool-chain of subparts; or a monolithic logic/system [51].

Tool-chain of subparts or semi-automatic system, is when we can choose what techniques and tools we will use. These actions can be separate or together, depends the tool that we use. In the end there is a tool-chain of subparts and we obtain the result.We will focus in this approach.

Monolithic logic/system is one unique system that does everything and the user interactions are easier, like the KeY tool [5].

2.2.1 Tool-chain of subparts

In a tool-chain of subparts there are two stages. The first stage is to translate the annotation into a set logic formulas; this step is called Verification Condition Generator (VCGen). In the second stage, the set of conditions are passed to an automatic theorem prover. This workflow is depicted in Figure 2.1.

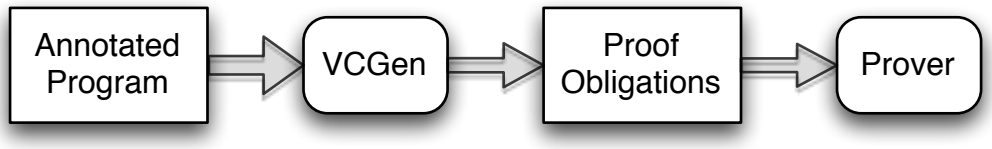

Figure 2.1: Program Verification Architecture

Tool chains based in Spec♯, JML and Code Contracts work on this way.

But there is a huge difference between Spec♯ or JML and Code Contracts or Boogie.

Spec♯ needs to track the evolution of C♯ and JML of Java. So if there is a new version of the main language, the tool will no longer support the same language.

On the other hand, Code Contracts are embedded in the program language itself (C♯), thus taking advantage of the existing language compiler, its supporting IDE and tools [33, 32]. Another advantage is that the developer does not need to learn a new specification language.

Listing 2.1 shows a small example (method `maxInteger`) with its annotations written in JML. Through this specification we are requiring that both variables must be greater or equal than zero ($precondition = a \geq 0 \;\&\&\; b \geq 0$) and after program termination, the result should reflect the maximum number of the two integers received as parameters ($postcondition = (a > b\,?\,a\;:\;b)$).

```
1  //@ requires a >= 0 && b >=0;
   //@ ensures \result == (a > b ? a : b);
3  public int maxInteger(int a, int b){
     int result;
5    if( a >= b ) {
       result = a;
7    }
     else {
9      result = b;
     }
11   return result;
   }
```

Listing 2.1: Method `maxInteger` annotated in JML

```
   procedure maxInteger(P#a: int, P#b: int) returns ($result: int);
2    requires P#a >= 0;
     requires P#b >= 0;
4    ensures $result == (if P#a > P#b then P#a else P#b);

6  implementation maxInteger(P#a: int, P#b: int) returns ($result: int){
     var L#result: int where $in_range_i4(L#result);
8
     anon1:
10     // result := @ite(>=(a, b), a, b);
       L#result := (if P#a >= P#b then P#a else P#b);
12     // return result;
       $result := L#result;
14     goto #exit;

16   anon2:
       // empty
18
     #exit:
20 }
```

Listing 2.2: Method `maxInteger` transformed in Boogie

Using VCC tool, we can also translate code written in C language to an intermediate language called Boogie. Then it is converted into the Satisfiability Modulo Theories (SMT) language before being passed to a prover, a SMT solver. In Listing 2.2 we have the method `maxInteger` translated into Boogie language.

Spec♯ works in a similar way as VCC. This is possible because several languages of Microsoft translate to Boogie. Currently there are five tools that transform an annotated program into Boogie, as depicted in Figure 2.2.

Each of these tools generates annotated Common Intermediate Language (CIL) bytecode, which Boogie translates to Boogie Program Language (BoogiePL) and then uses the weakest precondition calculus to generate verification conditions, which are then passed to an automatic theorem prover.

These tools are:

Spec♯ Spec♯ is an extension to language C♯, that allows to add annotations in the form of a *precondition*, *postcondition* and *invariants*.

HAVOC HAVOC (Heap Aware Verifier Of C) is a tool for specifying and checking properties of software systems written in C, in the presence of pointer manipulations, unsafe casts and dynamic memory allocation. HAVOC *deals with the low-level intricacies of C and provides reachability as a fundamental primitive in its specification language* [19].

VCC As referred, VCC, is a language that extends C with design-by-contract features, like pre-conditions, postconditions and invariants. VCC is also a tool that proves correctness of annotated concurrent C programs or finds problems in them.

Chalice Chalice is a language that explores specification and verification of concurrency in programs. *The Chalice program verifier analyses annotated programs and checks that the given annotations are never violated* [52].

Dafny Dafny, is an imperative language that supports static verification. Include *standard pre- and postconditions, framing constructs, and termination metrics* [50].

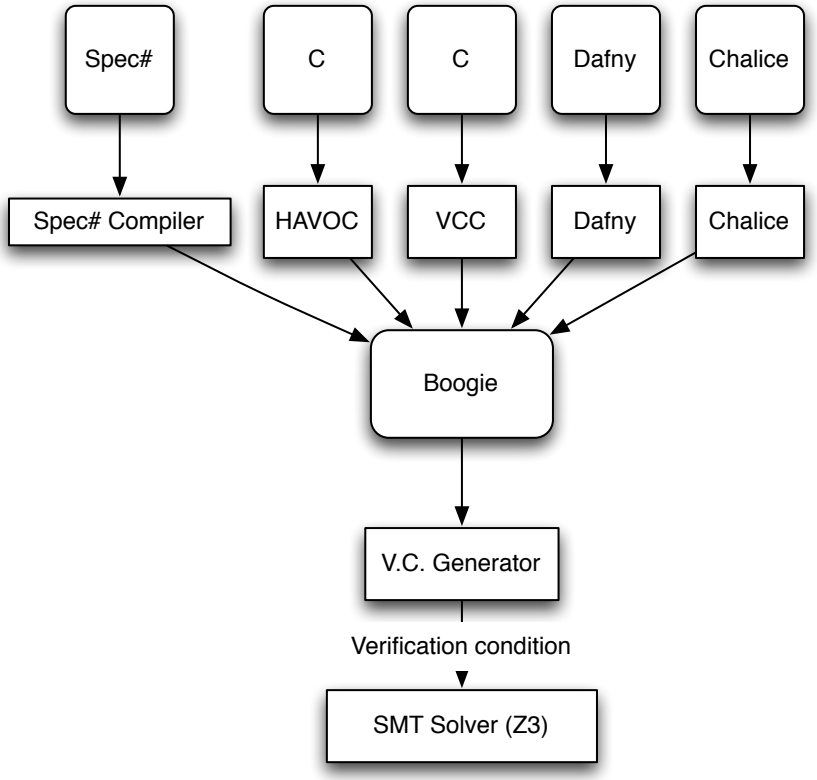

Figure 2.2: Boogie Architecture

Finally, in Listing 2.3 we have an example of Code Contracts. It is very similar to the JML illustrated in Listing 2.1, but instead of the annotations we have contracts.

```
public int maxInteger(int a, int b){
2   Contract.Requires(a >= 0 && b >= 0);
    Contract.Ensures(Contract.Result<int>() == (a > b ? a : b));
```

```
4   int result;

6   if (a >= b) {
        result = a;
8   }
    else {
10      result = b;
    }
12  return result;
}
```

Listing 2.3: Method `maxInteger` written in C♯ language with Code Contracts

Chapter 3

Verification Condition Generator

A Verification Condition Generator (VCGen) reduces the problem of proving the correctness of a program with respect to its specification to a set of Verification conditions (VCs) which only deal with the underlying data types of the variables in the language. This set of verification conditions are first-order logic formulas. In particular, all iterative or recursive reasoning with respect to proving the correctness of loops or of recursive procedures are resolved by the VCGen, automatically, and does not appear in the verification conditions produced.

At the end, this set of VCs will be passed to a theorem prover and the program is correct if all the proof obligations are valid. Otherwise the program is said to be wrong with respect to its specification.

The classic algorithm of a VCGen is based on the use of the *weakest-precondition calculus* [42] and takes as input a program and a postcondition (the precondition in this case only gives rise to a verification condition as will be shown).

In the following subsections a brief overview of this algorithm is given. Also, a description of some of the technologies used to implement a VCGen nowadays is presented.

3.1 The algorithm

The theoretically most efficient algorithms for generating VCs are based upon weakest precondition (WP) [43].

The weakest precondition of a program is calculated by the function `wp` defined in Figure 3.1. In the same figure, the function `VCG` calculates the set of VC using the previous functions. Please notice that the notation $Q[x \mapsto e]$ used denotes the substitution of x for e in Q.

This is a backward algorithm that iterates from the last statement to the first. We start with the postcondition and the last statement, with this we obtain an intermediate condition that will be a postcondition for the previous statement.

The clause for *while loops* contains two verification conditions. The first condition refers to the *loop initialisation and termination*, respectively. The second designates the *preservation of the loop invariant*.

As an example we will apply the VCGen algorithm to `power` procedure and specification, show in Figure 3.2. We start by calculating in Figure 3.3, `VC`(**power**,w=x^y). The same figure, shows the calculation of `VCG`({y$\geq$0} **power** {w=x^y}).

$$
\begin{aligned}
\textbf{wp}(\textbf{skip}, Q) &= Q \\
\textbf{wp}(x := e, Q) &= Q[x \mapsto e] \\
\textbf{wp}(C_1; C_2, Q) &= \textbf{wp}(C_1, \textbf{wp}(C_2, Q)) \\
\textbf{wp}(\textbf{if } b \textbf{ then } C_t \textbf{ else } C_f, Q) &= (b \rightarrow \textbf{wp}(C_t, Q)) \wedge (\neg b \rightarrow \textbf{wp}(C_f, Q)) \\
\textbf{wp}(\textbf{while } b \textbf{ do } \{I\} C, Q) &= I
\end{aligned}
$$

$$
\begin{aligned}
VC(\textbf{skip}, Q) &= \emptyset \\
VC(x := e, Q) &= \emptyset \\
VC(C_1; C_2, Q) &= VC(C_1, wp(C_2, Q)) \cup VC(C_2, Q) \\
VC(\textbf{if } b \textbf{ then } C_t \textbf{ else } C_f, Q) &= VC(C_t, Q) \cup VC(V_f, Q) \\
VC(\textbf{while } b \textbf{ do} \{I\} C, Q) &= \{(I \wedge b) \rightarrow wp(C, I), (I \wedge \neg b) \rightarrow Q\} \cup VC(C, I)
\end{aligned}
$$

$$VCG(\{P\}C\{Q\}) = \{P \rightarrow wp(C, Q)\} \cup VC(C, Q)$$

Figure 3.1: VCGen algorithm

$$
\begin{aligned}
&int\ power(x, y) \\
&\quad \{y >= 0\} \\
&\quad z := y; w := 1; \\
&\quad while(z > 1) do \{w = x^{y-z} \wedge z \geq 0\} \\
&\quad \{ \\
&\quad\ \ w := w * x; \\
&\quad\ \ z := z - 1; \\
&\quad \} \\
&\quad return w; \\
&\quad \{w = x^y\}
\end{aligned}
$$

Figure 3.2: Procedure `power`

VC(power,w=x^y)

= VC(z:=y;w:=1,wp(**while** z$\geq$1 **do** {I}C_w,w=x^y))
∪ VC(**while** z$\geq$1 **do** {I}C_w,w=x^y)

= VC(z:=y;w:=1,I) ∪ {I∧z$\geq$1→wp(C_w,I)} ∪ {I∧z<1→w=x^y}
∪ VC(C_w,I)

= VC(z:=y,wp(w:=1,I)) ∪ VC(w:=1,I)
∪ {w=x^{y-z} ∧ z$\geq$0 ∧ z$\geq$1→wp(w:=w*x,wp(z:=z-1,I))}
∪ {w=x^{y-z} ∧ z$\geq$0 ∧ z<1→w=x^y}
∪ VC(w:=w*x,wp(z:=z-1,I) ∪ VC(z:=z-1,I)

{ I ≡ w=x^{y-z}∧ z$\geq$0}

= ∅ ∪ ∅ ∪ {w=x^{y-z} ∧ z$\geq$0 ∧ z$\geq$1→wp(w:=w*x,w=$x^{y-(z-1)}$∧z-1$\geq$0)}
∪ {w=x^{y-z} ∧ z$\geq$0 ∧ z<1→w=x^y} ∪ ∅ ∪ ∅

= {w=x^{y-z} ∧ z$\geq$0 ∧ z$\geq$1→w*x=$x^{y-(z-1)}$∧z-1$\geq$0,
w=x^{y-z} ∧ z$\geq$0 ∧ z<1→w=x^y}

VCG({y$\geq$0}power{w=x^y})

= {y$\geq$0→wp(power,w=x^y)} ∪ VC(power,w=x^y)

= {y$\geq$0→wp(z:=y;w:=1,wp(**while** z$\geq$1 **do** {I} C_w,w=x^y))}
∪ {w=x^{y-z} ∧ z$\geq$0 ∧ z$\geq$1→w*x=$x^{y-(z-1)}$∧z-1$\geq$0,
w=x^{y-z} ∧ z$\geq$0 ∧ z<1→w=x^y}

= {y$\geq$0→wp(z:=y;w:=1,I),
w=x^{y-z} ∧ z$\geq$0 ∧ z$\geq$1→w*x=$x^{y-(z-1)}$∧z-1$\geq$0,
w=x^{y-z} ∧ z$\geq$0 ∧ z<1→w=x^y}

{ I ≡ w=x^{y-z}∧ z$\geq$0}

= {y$\geq$0→wp(z:=y;wp(w:=1,w=x^{y-z}∧ z$\geq$0)),
w=x^{y-z} ∧ z$\geq$0 ∧ z$\geq$1→w*x=$x^{y-(z-1)}$∧z-1$\geq$0,
w=x^{y-z} ∧ z$\geq$0 ∧ z<1→w=x^y}

= {y$\geq$0→1=x^0∧ z$\geq$0,
w=x^{y-z} ∧ z$\geq$0 ∧ z$\geq$1→w*x=$x^{y-(z-1)}$∧ z-1$\geq$0,
w=x^{y-z} ∧ z$\geq$0 ∧ z<1→w=x^y}

Figure 3.3: Applying the VCGen algorithm

At the end of this computation process, the result that outcomes is the following set of proof obligations.

1. y $\geq$ 0 $\rightarrow$ 1 = x^0 $\wedge$ z $\geq$ 0
2. w = x^{y-z} $\wedge$ z $\geq$ 0 $\wedge$ z $\geq$ 1 $\rightarrow$ w * x = $x^{y-(z-1)}$ $\wedge$ z-1 $\geq$ 0
3. w = x^{y-z} $\wedge$ z $\geq$ 0 $\wedge$ z < 1 $\rightarrow$ w = x^y

Nowadays, the modern verification condition generators use SMT language as the target language of the verification conditions and Z3 as theorem prover to check the validity of such conditions. Following, a description of this tool is given.

3.2 Satisfiability Modulo Theories solver

The problem of determining whether a formula expressing a constraint has a solution (also known as *satisfiability* problem) is one of the most fundamental problems in theoretical computer science. One of the ways to the constraint satisfaction problem is given by the *propositional satisfiability* SAT. In this approach, the goal is to decide whether a formula over boolean variables, formed using logical connectives, can be made *true* by choosing true/false values for its variables. But, usually, problems are described in a more expressive logic, such as first-order logic, where formulas are formed using logical connectives, variables, quantifiers, function and predicate symbols. It is in this context that the SMT [17] appears where the interpretation of some symbols is constrained by a background theory.

To sum up, a SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and the SMT solver aims at determining whether such a formula is *satisfiable*.

3.3 Z3

Z3 is a SMT solver from Microsoft Research that integrates a host of theory solvers in an expressive and efficient combination[28]. Z3 integrates several decision procedures and is used in program analysis, verification and test-case generation. It is possible with Z3 to solve decision problems for quantifier-free formulas with respect to combinations of theories, such as arithmetic, bit-vectors, arrays, and uninterpreted functions. In the next version, Z3 2.0, it will be available: proofs; non-linear arithmetic (Gröbner Bases) [1]; and improved array & bit-vector theories.

Chapter 4

Boogie

Boogie is both the name of a language and a tool.

> "Boogie is an intermediate verification language, designed to make the prescription of verification conditions natural and convenient. It serves as a common intermediate representation for static program verifiers of various source languages, and it abstracts over the interfaces to various theorem provers."[56]

Along this chapter, it will be discussed the different features of both the Boogie Program Verifier and the Boogie language. In Section 4.1 it is explained the most important language statements for this work with a brief description. In Section 4.2 it is explained the verification conditions generation through Boogie.

4.1 Boogie Program Language

Boogie is an intermediate language for program analysis and program verification [29, 35]. The Boogie Language was previous known as Boogie Program Language (BoogiePL).

BoogiePL concepts relations, and features will be introduced based on BoogiePL grammar. The grammar is written extended BNF where the meta-level symbols $*$, $+$, $?$ will be used respectively to indicate a sequence, a nonempty sequence, and an optional syntactic entity. It will be used $|$ for alternatives.

A Boogie program consists of a theory that is used to encode the semantics of the source language, followed by an imperative part.

The imperative part is composed of a set of declarations and has the following form:

```
Program  ::=  Decl*
   Decl  ::=  TypeDecl | ConstantDecl | FunctionDecl | AxiomDecl
          |   VarDecl | ProcedureDecl | ImplementationDecl
```

The most important alternatives are ProcedureDecl and ImplementationDecl. In Subsection 4.1.1 we describe the ProcedureDecl specification. At Subsection 4.1.2 we have more details about the implementation declaration. Subsection 4.1.3 contains the information about blocks and the commands that are part of such basic block.

4.1.1 Procedures

The *procedure* is a name for a parameterized operation on the state space.

ProcedureDecl	::=	**procedure** Id Signature ";" $Spec^*$
	|	**procedure** Id Signature $Spec^*$ Body
Signature	::=	ParamList [**returns** ParamList]
Spec	::=	**requires** Expression ";"
	|	**modifies** [IdList] ";"
	|	**ensures** Expression ";"

Basically, a procedure specification consists of *three* kinds of clauses.

A precondition (**requires** clause) specifies a boolean condition that holds in the initial state of each execution trace of the procedure. Generally, it is the caller's responsibility to establish the precondition at a call site, and the implementation gets to assume the precondition to hold on entry.

A postcondition (**ensures** clause) specifies a boolean condition that relates the initial and final states of each finite execution trace of the procedure. Generally, it is the implementation's responsibility to establish the postcondition on exit, and the caller gets to assume the postcondition to hold upon return.

The **modifies** clause lists those global variables that are allowed to change during the course of the procedure's execution traces.

Syntactically, the specification consists of any number of preconditions (**requires** clauses), **modifies** clauses, and postconditions (**ensures** clauses).

A *signature* declares zero or more arguments, being in-parameters, or out-parameters.

4.1.2 Implementations

A procedure implementation is declared as follows:

ImplementationDecl	::=	**implementation** Id Signature Body
Body	::=	"{" $LocalVarDecl^*$ $Block^+$ "}"
LocalVarDecl	::=	**var** IdTypeList ";"

The implementation is composed of zero or more local variables followed by one or more basic blocks, see Subsection 4.1.3.

The execution of an implementation consists of a sequence of basic blocks, beginning with the first listed basic block, (*entry*) and then continuing to other basic block, as per the block's transfer-of-control manifesto, ending when the **return** statement is reach.

For every implementation P, the program must also contain a procedure declaration for P.

4.1.3 Basic blocks

A block has a label and a sequence of commands, followed by a control transfer command.

Block	::=	label cmd* transfercmd;
cmd	::=	passive | assign | call
passive	::=	**assert** expr ; | **assume** expr ;
assign	::=	var ([expr , expr])? := expr ; | **havoc** var+;
call	::=	**call** var* := procname (expr*) ;
transfercmd	::=	**goto** label+; | **return** ;

The **assert** and **assume** commands indicate conditions to be check or logical formulas to be use, respectively, in the verification. If the given expression evaluates to true, then each of these commands proceeds like a no-op. If the condition evaluates to false, the assert command goes

wrong, which is a terminal failure. For the assume command, if the condition evaluates to false, one is freed of all subsequent proof obligations, thus indicating a terminal success.

The **havoc** command assigns an arbitrary value to each indicated variable; when present, the variable's where clause constrains this value.

The **goto** command jumps nondeterministically to one of the indicated blocks.

The **return** command ends the implementation.

The **call** command is defined in terms of the specification of the procedure being called.

The **assert** and **assume** commands result from one of the transformations performed during the translation of a program in Common Intermediate Language (CIL) to Boogie. Such transformation is called *passivization* and it will be explained in the next subsection.

4.1.4 Passivization

The weakest precondition for a program assumes stateless blocks. So, there are two ways of getting rid of assignments:

- Establish *dynamic single assignment form* (DSA), i.e. there is at most one definition for each variable on each path.
  - Replace definitions/uses with new incarnations. For instance, the assignment $x := x + 1$ will be $x_{i+1} := x_i + 1$.
  - Replace **havoc** x with new incarnations x_{n+1}
  - At join points unify variable incarnations.
- Eliminate assignments by replacing: $x := E$ with **assume** $x = E$.

Thus, a procedure p is said to be passive if:

- variables are assigned at most once in every execution path of body of p

This means that a program is transformed into a *single-assignment program*. Single-assignment forms were introduced in the 1980s in the area of compiler design, as intermediate representations of code.

However, loops introduce back edges in control flow graph. But the technique explained above can only deal with acyclic graphs. To get rid of such back edges, we will need to:

- Duplicate loop invariant P by using: $assert\ P = assert\ P; assume\ P;$
- Check loop invariant at loop entry and exit.
- Delete back edges after "havoc"-ing loop targets.

As depicted in Figure 4.1.

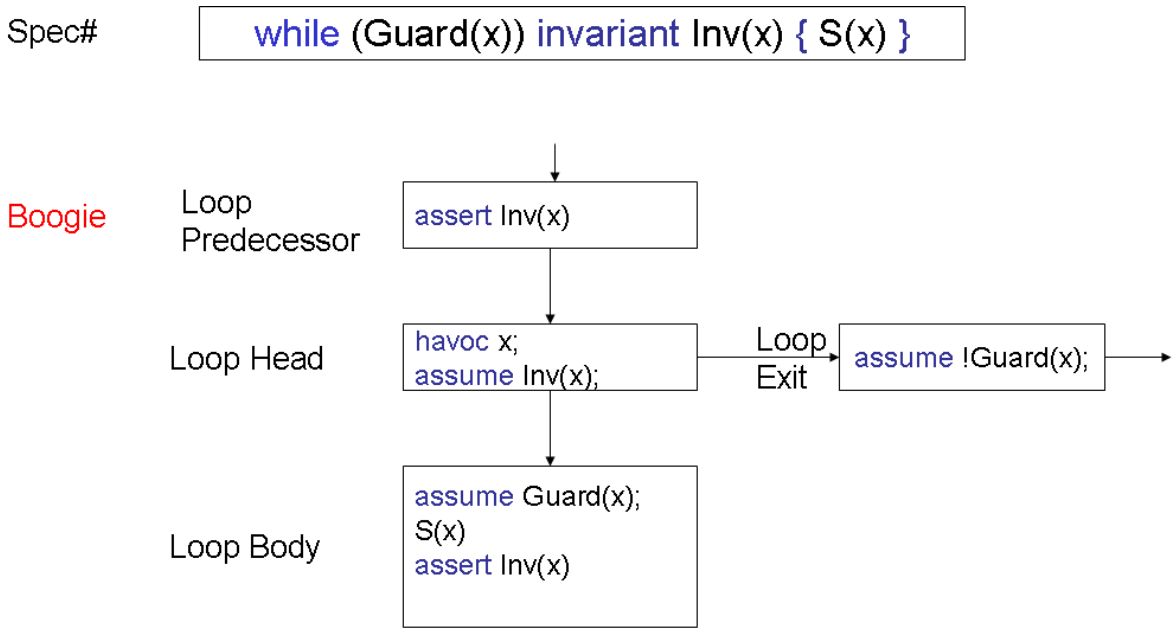

Figure 4.1: Transforming a loop in Boogie

To illustrate the idea, consider the Spec# source program in Listing 4.1, taken from[7].

```
1  int M(int x)
     requires 100 <= x:
3    ensures result == 0;
   {
5    while {0 < x}
       invariant 0 <= x;
7    {
       x = x - 1;
9    }
     return x;
11 }
```

Listing 4.1: Spec# program example

After translate the program to Boogie we obtain the following code:

```
Start:      assume 100 <= x;
            goto LoopHead;
LoopHead:   assert 0 <= x;
            goto Body, After;
Body:       assume 0 < x;
            x := x - 1;
            goto LoopHead;
            return;
After:      assume ¬(0 < x);
            r := x;
            assert r = 0;
            return r;
```

The next step is to cut the back edges, so the loop-free program is:

```
Start:      assume 100 <= x;
            assert 0 <= x;
            goto LoopHead;
LoopHead:   havoc x;
            assert 0 <= x;
            goto Body, After;
Body:       assume 0 < x;
            x := x - 1;
            assert 0 <= x;
            return;
After:      assume ¬(0 < x);
            r := x;
            assert r = 0;
            return r;
```

The passive form of the program is then:

```
Start:      assume 100 <= x_0;
            assert 0 <= x_0;
            goto LoopHead;
LoopHead:   skip;
            assert 0 <= x_1;
            goto Body, After;
Body:       assume 0 < x_1;
            x_2 := x_1 - 1;
            assert 0 <= x_2;
            return;
After:      assume ¬(0 < x_1);
            r_1 := x_1;
            assert r_1 = 0;
            return r;
```

4.2 Boogie Program Verifier

To verify a program, Boogie generates a set of verification conditions in a first step — logical formulas whose validity implies that the program satisfies the correctness properties under consideration. The verification conditions are then processed with the help of a theorem prover, where a successful proof attempt shows the correctness of the program, and a failed proof attempt may give an indication of a possible error in the program.

The Boogie pipeline is depicted in Figure 4.2.

There are different ways for the generation of the verification conditions. But each one can have a dramatic impact on the performance of the underlying theorem prover. Boogie performs a series of transformations on the program, essentially producing one snippet of the verification condition from each basic block of the Boogie language procedure implementation being verified. The verification condition is represented as a formula in first-order logic. It is then passed to a first-order automated theorem prover to determine the validity of the verification condition (and thus the correctness of the program).

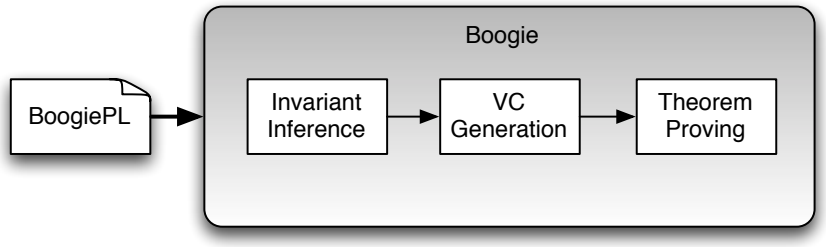

Figure 4.2: Boogie pipeline

Stmt	wp(Stmt,Q)	wlp(Stmt,Q)
x := E	Q[x:=e]	Q[x:=e]
assert E	E ∧ Q	E ⇒ Q
assume E	E ⇒ Q	E ⇒ Q
S ; T	wp(S,wp(T,Q))	wlp(S,wlp(T,Q))
S □ T	wp(S,Q) ∧ wp(T,Q)	wlp(S,Q) ∧ wlp(T,Q)

Table 4.1: Weakest precondition and Weakest liberal precondition of a statement

In particular, Boogie is based on the *weakest liberal precondition* (wlp) to the calculation of the verification conditions.

For any statement S and predicate Q on the post-state of S, the WP of S with respect to Q, denoted $wp(S, Q)$, is a predicate on the pre-state of S, characterizing all pre-states from which every non-blocking execution of S does not go wrong and terminates in a state satisfying Q. $wlp(S, Q)$ characterizes the pre-states from which every non-blocking execution of S either goes wrong or terminates in a state satisfying Q. Table 4.1 show the calculation of both wp ans wlp for a given statement.

The way how Boogie generates these VC make possible to construct an error trace from a failed proof.

where Q[x := E] denote the substitution of E for x in Q, that is:

$$Q[x := E] = \mathbf{let}\ x = E\ \mathbf{in}\ Q\ \mathbf{end}$$

The problem of use WP is the redundancy. In wp(S; T,Q), which expand to $wp(S, Q) \wedge wp(T, Q)$, Q is duplicated and that result in an exponential size of the formula generated. But using weakest liberal precondition (WLP) to compute the WP can reduce such complexity to quadratic instead of exponential. The connection between WP and WLP is described in [30]:

$$(\forall Q \bullet wp(S, Q) \equiv wp(S, true) \wedge wlp(S, Q)) \tag{4.1}$$

In [49], the authors call to the following property, a ”dream property”:

$$wlp(S, Q) \equiv (wlp(S, false) \vee Q) \tag{4.2}$$

With this we can compute wp(S,Q) for a Q that is not the literal true, compute wp(S,true) ∧ wlp(S,Q) compute wlp(S,Q) for a Q that is not the literal false, compute wlp(S,false) ∨ Q compute

wp(S,true) and wp(S,false), apply the syntactic transformations suggested in (1).

4.3 Summary

In this chapter we present the Boogie framework. It was introduced and explained the internals of Boogie Program Verifier and described the Boogie programming language.

Chapter 5

Program Slicing

Slicing was first proposed in 1979, by Weiser in his PhD thesis [66] and was then used to ease program debugging [60, 3, 65, 44], software testing [15, 40], software metrics [47, 57], software maintenance [21, 36], program comprehension [41, 53] and so on. Slicing can be use to extract the statement relevant to a particular computation.

Program slicing, in its original version, is a decomposition technique that extracts from a program the statements relevant to a particular computation. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a *slicing criterion*.

Weiser defined a program slice S as a reduced, *executable program* obtained from a program P removing statements, such that S preserves the original behaviour of the program with respect to a subset of variables of interest and at a given program point.

Executable means that the slice is not only a closure of statements, but also can be compiled and run. *Non-executable* slices are often smaller and thus more helpful in program comprehension.

The slices mentioned so far are computed by gathering statements and control predicates by way of a *backward traversal* of the program, starting at the slicing criterion. Therefore, these slices are referred to as **backward slices** [62]. In [12], Bergeretti and Carr were the first to define a notion of a *forward* slice. A **forward slice** is a kind of ripple effect analysis, this is, it consists of all statements and control predicates dependent on the slicing criterion. A statement is dependent of the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine if the statement under consideration is executed or not.

Both *backward* or *forward* slices are classified as *static* slices. **Static** means that only statically available information is used for computing slices, this is, all possible executions of the program are taken into account; no specific input I is taken into account.

Since the original version proposed by Weiser [64], various slightly different notions of program slices, which are not static, have been proposed, as well as a number of methods to compute slices. The main reason for this diversity is the fact that different applications require different program properties of slices.

Some of these variants include:

- **Dynamic slicing**: *Korel and Laski* proposed an alternative slicing definition, named *dynamic slicing* in [45, 46], where a slice is constructed with respect to only one execution of the program corresponding just to one given input. It does not include the statements that have no relevance for that particular input.

- **Quasi-static slicing** *Venkastesh* introduced in 1991 the *quasi-static* slicing in [63], which is a slicing method between static slicing and dynamic slicing. A quasi-static slice is constructed

with respect to some values of the input data provided to the program. It is used to analyse the behaviour of the program when some input variables are fixed while others vary.

- **Simultaneous dynamic slicing** *Hall* proposed the *simultaneous dynamic slicing* in [39], which computes slices with respect to a set of program executions. This slicing method is called *simultaneous dynamic program slicing* because it extends dynamic slicing and simultaneously applies it to a set of test cases, rather than just one test case.
- **Conditioned slicing** *Canfora et al* introduced in [18] the concept of conditioned slicing. A conditioned slice consists of a subset of program statements which preserves the behaviour of the original program with respect to a slicing criterion for any set of program executions. The set of initial states of the program that characterise these executions is specified in terms of a first order logic formula on the input.
- **Union Slicing** *Beszedes* et al [14, 13] introduced the concept of *union slice* and the computing algorithm. A union slice is the union of dynamic slices for a finite set of test cases; actually is very similar to simultaneous dynamic program slicing. An union slice is an approximation of a static slice and is much smaller than the static one.

In this chapter, are only discussed the most relevant ones with respect to the work discussed in this document: static slicing (in Section 5.1), dynamic slicing (in Section 5.2) and conditioned slicing (in Section 5.3).

Program example Listing 5.1 corresponds to a program, taken from [18]. This program will be used as running example in the next sections.

Initially the program reads the integers `test0`, `n` and a sequence of `n` integers `a` as input and compute the integers `possum`, `posprod`, `negsum` and `negprod`. The integers `possum` and `posprod` accumulate the sum and product of the positive numbers in the sequence, respectively. On the other hand the integers `negsum` and `negprod` accumulate the sum and product of the absolute value of the negative numbers in the sequence, respectively. Whenever an input is zero, the greatest sum and product are reset if the value of test is non zero (true).

The program prints the greatest sum and product computed.

```
1  main() {
     int a, test0, n, i, posprod, negprod, possum, negsum, sum, prod;
3    scanf("%d", %test0); scanf("%d",&n);
     i = posprod = negprod = 1;
5    possum = negsum = 0;
     while(i<=n){
7      scanf("%d",&a);
       if (a > 0) {
9        possum += a;
         posprod *= a;
11     } else if (a < 0){
         negsum -= a;
13       negprod *= (-a);
       } else if (test0){
15       if (possum >= negsum){
           possum = 0;
17       } else { negsum = 0; }
         if (posprod >= negprod){
19         posprod = 1;
         } else { negprod =1; }
21     }
       i++;
23   }
     if (possum >= negsum) {
25     sum = possum;
     }
27   else { sum = negsum; }
     if ( posprod >= negprod) {
29     prod = posprod;
     } else {
31     prod = negprod;
     }
33   printf("Sum: %d\n", sum);
     printf ("Product : %d\n" ,  prod );
35 }
```

Listing 5.1: Program example 1

5.1 Static slicing

Based on the original definition of Weiser, a static slice consists of only the parts of the program that affect the values computed at some point of interest referred to as a *slicing criterion* `C=(p,V)`, where `p` is a program point and `V` is a set of variables of interest.

A static slice preserves the behaviour of the original program with respect to the slicing criterion for the program execution.

Definition 1 (Static slicing)**.** *A* static slice S *of a program* P *on slicing criterion* $C = (p, V_s)$ *is any syntactically correct and executable program with the following properties:*

- S *can be obtained from* P *by deleting zero or more statements from* P*.*
- *Whenever* P *halts, on input* I*, with state trajectory* T*, then* S *also halts, with the same input* I*, with the trajectory* T'*, and* $Proj_C(T) = Proj_C(T')$*.*

where

Definition 2 (Projection). *Let $C = (p, V_s)$ be a static slicing criterion of a program P and $T =< (p_1, \sigma_1), (p_2, \sigma_2), ..., (p_k, \sigma_k) >$ a state trajectory of P on input I. $\forall i, 1 \leq i \leq k$:*

$$Proj'_C(p_i, \sigma_i) = \begin{cases} \lambda & if p_i \neq p \\ < (p_i, \sigma_i | V_s) > & if p_i = p \end{cases}$$

Listing 5.2 shows a static slice of the program 5.1 on the slicing criterion C = (I, 34, prod).

```
1  main() {
     int a, test0, n, i, posprod, negprod, prod;
3    scanf("%d", %test0); scanf("%d",&n);
     i = posprod = negprod = 1;
5    while(i<=n){
       scanf("%d",&a);
7      if (a > 0) {
         posprod *= a;
9      } else if (a < 0){
         negprod *= (-a);
11     } else if (test0){
         if (posprod >= negprod){
13         posprod = 1;
         } else { negprod =1; }
15     }
       i++;
17   }
     if ( posprod >= negprod) {
19     prod = posprod;
     } else {
21     prod = negprod;
     }
23   printf ("Product : %d\n" ,  prod );
   }
```

Listing 5.2: A static slicing of program 5.1

5.2 Dynamic slicing

The dynamic slicing (DS) was proposed by *Korel* and *Laski* [2, 45], where a slice is constructed with respect to only one execution of the program corresponding just to one given input. It does not include the statements that have no relevance for that particular input.

Definition 3 (Dynamic Slicing). *A* dynamic slice *of a program* P *on a dynamic slicing criterion $C = (I, p, V_s)$ is any syntactically correct and executable program* P' *obtained from* P *by deleting zero or more statements, and whenever* P *halts, on input* I*, with state trajectory* T*, then* P' *also halts, on the same input* I*, with state trajectory* T'*, and $Proj_{(p,V_s)}(T) = Proj_{(p,V_s)}(T')$.*

The difference between static and dynamic slicing is that dynamic slicing assumes fixed input for a program, whereas static slicing does not make assumptions regarding the input. [62]

To clarify the difference between static and dynamic slicing, consider a program unit with an iteration block containing an `if-else` block. Consider that the static slicing (SS) contain the if-else block. In the case of DS we consider a particular execution of the program, where only one

part of the block is executed. So, in this particular execution case, the dynamic slice would contain only the statements in the `if` block or `else` block.

The *slicing criterion* in this variant is `C=(I,p,V)`, where `I` is a set of tuples, variable and value.

Listing 5.3 shows a dynamic slice of the program 5.1 on the slicing criterion C = (I, 34, prod) where I =< (test, 0), (n, 2), (a1, -1), (a2, -2) >.

```
main() {
  int a, test0, n, i, posprod, negprod, prod;
  scanf("%d", %test0); scanf("%d",&n);
  i = posprod = negprod = 1;
  while(i<=n){
    scanf("%d",&a);
      if (a < 0){
      negprod *= (-a);
    }
    i++;
  }
  if ( posprod >= negprod) {
    prod = posprod;
  } else {
    prod = negprod;
  }
  printf ("Product : %d\n" ,  prod );
}
```

Listing 5.3: A dynamic slicing of program 5.1

5.3 Conditioned slicing

The conditioned slicing (CS) was presented by Canfora, Cimitile, and Lucia [18]. This slicing can see as a bridge between the two extremes of static and dynamic analysis [41].

A conditioned slice consists of a subset of program statements which preserves the behaviour of the original program with respect to a slicing criterion for any set of program executions. The set of initial states of the program that characterise these executions is specified in terms of a first order logic formula on the input. This allows a programmer to further specialise a program by eliminating statements which do not contribute to the computation of the variables of interest when the program is executed in one of the initial states of interest.

Definition 4 (Conditioned Slicing Criterion)**.** *Let V_i be the set of input variables of a program P, and F be a first order logic formula on the variables in V_i. A* conditioned slicing criterion *of a program P is a triple $C = (F(V_i), p, V_s)$ where p is a statement in P and V_s is the subset of the variables in P which will be analysed in the slice.*

Definition 5 (Conditioned Slicing)**.** *A* conditioned slice *of a program* P *on a conditioned slicing criterion $C = (F(V_i), p, V_s)$ is any syntactically correct and executable program* P' *such that: P' is obtained from* P *by deleting zero or more statements; whenever* P *halts, on input* I*, with state trajectory* T*, where $I \in C(I', V_i')$, $I' \in S(F(V_i))$, V_i' is the set of input variables of P, and S is the satisfaction set, then* P' *also halts, on input* I*, with state trajectory* T'*, and $Proj_{(p,V_s)}(T) = Proj_{(p,V_s)}(T')$.*

Listing 5.4 shows a conditioned slice of the program 5.1 on the slicing criterion C = (*F*(V_i), 34, prod) where V_i = {n} $\cup_{1 \leq i \leq n}$ $\{a_i\}$ and *F*(V_i)=$\forall$ i, 1 $\leq$ i $\leq$ n , a_i > 0.

```
main() {
  int a, test0, n, i, posprod, negprod, prod;
  scanf("%d", %test0); scanf("%d",&n);
  i = posprod = negprod = 1;
  while(i<=n){
    scanf("%d",&a);
      if (a > 0){
    posprod *= a;
    }
    i++;
  }
  if ( posprod >= negprod) {
    prod = posprod;
  }
  printf ("Product : %d\n" ,  prod );
}
```

Listing 5.4: A conditioned slicing of program 5.1

As discussed in the previous sections, program slicing is a well-recognised technique that is used mainly at source code level to highlight code statements that impact upon other statements. Slicing has many applications because it allows a program to be simplified by focusing attention on a sub-computation of interest for a chosen purpose. Some of these applications include: debugging, software maintenance, reverse engineering, program comprehension, testing and measurement.

Chapter 6

Slicing programs with contracts

Program verification goal is to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behaviour of the implementation matches that of the specification (this is usually called the *functional* behaviour of the program), and moreover the program does not 'go wrong', for instance no errors occur during evaluation of expressions (the so-called *safety* behaviour).

In recent years program verification has been closely linked with the so-called Design-by-Contract (DbC) approach to software development [55], which facilitates modular verification and certified code reuse. The contract for a software component can be regarded as a form of enriched software documentation that fully specifies the behaviour of that component. In terms of verification terminology, a contract for a component is simply a pair consisting of a precondition and a postcondition written in a formal language (usually a logical one) generally designated as an annotation language. It certifies the results that can be expected after execution of the component, but it also constrains the input values of the component. The development and broad adoption of annotation languages for the major programming languages reinforces the importance of using DbC principles in program development. These include for instance the Java Modeling Language (JML) [16]; Spec# [8], a formal language for C# API contracts; and the ANSI/ISO C Specification Language (ACSL) [10].

Program verification and slicing techniques used in the context of source code analysis, are apparently unrelated areas. However, one point of contact that has been identified between slicing and verification is that traditional dependency-based slicing, applied a priori, facilitates the verification of large programs. In this paper we explore the idea that it makes sense to slice programs based on semantic, rather than syntactic, criteria – the contracts used in DbC and program verification are excellent candidates for such criteria.

A typical case of being useful to calculate the slice of a program based on a specification is the reuse of annotated code. Suppose one is interested in reusing a module whose advertised contract consists of precondition P and postcondition Q, in situations in which a stronger precondition P' is known to hold, or else the desired postcondition Q' is weaker than the specified Q. Then from a software engineering perspective it would be desirable to eliminate, at source-level, the code that may be spurious with respect to the specification (P', Q').

Although the basic ideas have been published for over 10 years now, assertion-based slicing is still not very popular . The widespread usage of code annotations, as explained above, is however an additional argument for promoting it.

From now one, the expression *assertion-based slicing* [9] will be used to encompass postcondition-based (Section 6.1), precondition-based (Section 6.2), and specification-based (Section 6.3) forms of slicing. All these algorithms work at the intra-procedural level. To read about the differences among these algorithms and improved versions of them please see [9].

Besides introducing this concept of *assertion-based slicing* and improving slicing algorithms,

```
x := x+100;
2 x := x+50;
x:= x-100
```

Listing 6.1: Example for postcondition-based slicing

Daniela da Cruz in [9] also presents a tool, GamaSlicer, that implements all the algorithms and animates there execution. Due to the central role of GamaSlicer in this master work, it will be introduced in Section 6.4. In that section it is described GamaSlicer architecture and features, and also exemplified all how it applies the slicing to a given program. Although brief, this presentation clarifies what is expected from GamaBoogie.

6.1 Postcondition-based Slicing

The idea of slicing programs based on their specifications was introduced by Comuzzi et al. [22] with the notion of *predicate slice* (*p-slice*), also known as postcondition-based slice.

The algorithm proposed by Comuzzi runs in *quadratic time* on the length of the sequence. The algorithm first tries to slice the entire program by removing its longest removable suffix, and then repeats this task, considering successively shorter prefixes of the resulting program, and removing their longest removable suffixes. Schematically:

$$\begin{array}{l}\textbf{for } j = n+1, n, \ldots, 2 \\ \quad \textbf{for } i = 1, \ldots, j-1 \\ \quad\quad \textbf{if } \text{valid}\left(\overline{\text{wprec}}_i(S,Q) \rightarrow \overline{\text{wprec}}_j(S,Q)\right) \textbf{ then } S \leftarrow \text{remove}(i, j-1, S)\end{array}$$

For instance in a program with 999 statements the following pairs (i, j) would be considered in this order:

$$(1,1000),(2,1000),\ldots,(999,1000),(1,999),(2,999),\ldots,(998,999),(1,998),\ldots$$

To understand the idea of p-slices, consider a program S and a given postcondition Q. It may well be the case that some of the commands in the program do not contribute to the truth of Q in the final state of the program, i.e. their presence is not required in order for the postcondition to hold. In this case, the commands may be removed. A crucial point here is that the considered set of executions of the program is restricted to those that will result in the postcondition being satisfied upon termination. In other words, not every initial state is admissible – only those for which the *weakest precondition* of the program with respect to Q holds.

Consider for instance Listing 6.1. The postcondition $Q = x \geq 0$ yields the weakest precondition $x \geq -50$. If the program is executed in a state in which this precondition holds and the commands in lines 2 and 3 are removed from it, the postcondition Q will still hold. To convince ourselves of this, it suffices to notice that after execution of the instruction in line 1 in a state in which the weakest precondition is true, the condition $x \geq 50$ will hold, which is in fact stronger than Q.

Applications In their paper Comuzzi and Hart give a number of examples of the usefulness of postcondition-based slicing, based on their experience as software developers and maintainers. Their emphasis is on applying slicing to relatively small fragments of big programs, using postconditions corresponding to properties that should be *preserved* by these fragments. Suppose one

```
1 x := x+100;
  x := x-200;
3 x := x+200
```

Listing 6.2: Example for precondition-based slicing

suspects that a problem was caused by some property Q being false at line k of a program S with n lines of code. We can take the subprogram S_k consisting of the first k lines of S and slice it with respect to the postcondition Q. This may result in a suffix of S_k being sliced off, say from lines i to k, which means that in order for Q to hold at line k, it must also hold at line i. The resulting slice is where the software engineers should now concentrate in order to find the problem (a similar reasoning applies if the sequence of lines removed is not a suffix of S_k).

6.2 Precondition-based Slicing

Chung and colleagues [20] later introduced *precondition-based slicing* as the dual notion of postcondition-based slicing. The idea is still to remove statements whose presence does not affect properties of the final state of a program. The difference is that the considered set of executions of the program is now restricted directly through a first-order condition on the initial state. Statements whose absence does not violate any property of the final state of any such execution can be removed. This is the same as saying that the assertion calculated as the strongest postcondition of the program (resulting from propagating forward the given precondition) is not weakened in the computed slice.

Schematically the algorithms are very similar to the postcondition-based, but is uses the strongest postcondition calculus:

$$\begin{array}{l}\textbf{for } j = n+1, n, \ldots, 2 \\ \quad \textbf{for } i = 1, \ldots, j-1 \\ \quad\quad \textbf{if } \text{valid}\left(\overline{\mathsf{spost}}_i(S,P) \rightarrow \overline{\mathsf{spost}}_j(S,P)\right) \textbf{ then } S \leftarrow \mathsf{remove}(i, j-1, S)\end{array}$$

As an example of a precondition-based slice, consider now Listing 6.2, and the precondition $P = x \geq 0$. The effect of the first two instructions are to weaken the precondition. If these instructions are sliced off and the resulting program is executed in a state in which P holds, whatever postcondition held for the initial program will still hold for the sliced program.

Applications Redundant code is code that does not produce any effect: removing it results in a program that behaves in the same way as the original. Note that we say "the code does not produce any effect" in the sense of observable effects on the final state. Removing redundant code may of course result in code that is different regarding the execution traces; in particular the resulting code may be faster to execute. A major application of precondition-based slicing is the removal of *conditionally redundant code*, i.e. code that is redundant for executions of the program specified by a given precondition. Naturally, redundant code is a special case of conditionally redundant code.

```
1 x := x*x;
  x := x+100;
3 x := x+50
```

Listing 6.3: Example for specification-based slicing

6.3 Specification-based Slicing

A *specification-based slice* can be calculated when both a precondition P and a postcondition Q are given for a program S. The set of relevant executions are restricted to those for which Q holds upon termination when the program is executed in a state satisfying P. Programs resulting from S by removing a set of statements, and which are still correct regarding (P, Q), are said to be specification-based slices of S with respect to (P, Q).

The method proposed in [20] to compute such slices are based on a theorem proved by the authors, which states that the composition, in any order, of postcondition-based slicing (with respect to postcondition Q) and precondition-based slicing (with respect to precondition P) produces a specification-based slice with respect to (P, Q).

Although this method does compute specification-based slices, it does not compute minimal slices, as can be seen by looking at program in Listing 6.3 with specification $(\top, x \geq 100)$. One have:

$$\begin{aligned}
&\overline{\mathsf{spost}}_0(S, P) = \top \\
&\overline{\mathsf{spost}}_1(S, P) = \exists v.x = v * v \\
&\overline{\mathsf{spost}}_2(S, P) = \exists w.(\exists v.w = v * v) \wedge x = w + 100 && \equiv \exists v.x = v * v + 100 \\
&\overline{\mathsf{spost}}_3(S, P) = \exists w.(\exists v.w = v * v + 100) \wedge x = w + 50 && \equiv \exists v.x = v * v + 150
\end{aligned}$$

and

$$\begin{aligned}
&\overline{\mathsf{wprec}}_4(S, Q) = x \geq 100 = Q \\
&\overline{\mathsf{wprec}}_3(S, Q) = x \geq 50 \\
&\overline{\mathsf{wprec}}_2(S, Q) = x \geq -50 \\
&\overline{\mathsf{wprec}}_1(S, Q) = \top
\end{aligned}$$

It is obvious that the postcondition is satisfied after execution of the instruction in line 2, which means that if line 3 is removed the sliced program will still be correct with respect to $(\top, x \geq 100)$. However, precondition-based and postcondition-based slicing both fail in removing this instruction, since no forward implications are valid among the $\overline{\mathsf{spost}}_i(S, P)$ or the $\overline{\mathsf{wprec}}_i(S, Q)$. Composing precondition-based and postcondition-based slicing will of course not solve this fundamental flaw.

Thus, *Barros et al* proposed an alternative principle in [9] to compute specification-based slices. The basic idea is to find valid implications among the strongest postconditions and the weakest preconditions, instead of trying to find valid implications among the strongest postconditions and then among the weakest preconditions (or in the other way around).

$$\begin{aligned}
&\textbf{for } j = n+1, n, \ldots, 2 \\
&\quad \textbf{for } i = 1, \ldots, j-1 \\
&\quad\quad \textbf{if } \text{valid}\left(\overline{\mathsf{spost}}_i(S, P) \rightarrow \overline{\mathsf{wprec}}_j(S, P)\right) \textbf{ then } S \leftarrow \mathsf{remove}(i, j-1, S)
\end{aligned}$$

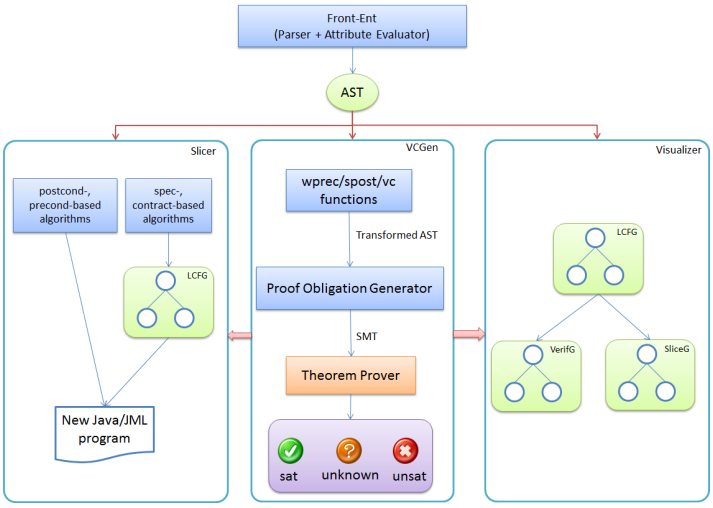

Figure 6.1: GamaSlicer architecture

Solved the flaw of the original algorithm to compute specification-based slices, *Barros et al* also proposed an algorithm to find minimal slices. This algorithm is based on labeled control flow graph (LCFG) and in the construction of a Slice Graph.

For each valid implication found, a new edge is added to the LCFG. When the slice graph is complete, it will contain the entire set of specification-based slices of a program, and obtaining the minimal slice is simply a matter of selecting the shortest subsequences using the information in the graph.

6.4 Assertion-based Slicing and GamaSlicer

GamaSlicer is a framework[1] that includes a Verification Condition Generator (VCGen), a parameterizable slicer with a choice of algorithms, and a visualisation functionality. It works on Java programs with JML annotations (the standard specification language for Java [48]). Instead of programs consisting of sets of procedures, one has classes with their methods, sharing a set of class/instance variables instead of global variables.

6.4.1 GamaSlicer Architecture

The architecture of GamaSlicer, inspired by that of a compiler (or generally speaking a language processor), is depicted in Figure 6.1. It is composed of the following blocks: a Java/JML front-end (a parser and an attribute evaluator); a verification conditions generator; a slicer; and a LCFG visualizer.

Since the underlying logic of slicing algorithms as well as the VCGen is first-order logic, the tool outputs proof obligations written in the SMT-Lib (Satisfiability Modulo Theories library) language. We chose SMT-Lib since it is nowadays the language employed by most provers used in program verification, including, among many others, Z3 [27], Alt-Ergo [23], and `Yices` [31].

[1] Available for download at `http://gamaepl.di.uminho.pt/gamaslicer`. Version 2.0 will be released soon as a desktop version.

After uploading a file containing a piece of Java code together with a JML specification, the code is recognised by the front-end (an analyser produced automatically from an attribute grammar with the help of the AnTLR parser generator [58]), and is transformed into an Abstract Syntax Tree (AST). During this first step also an identifiers table is built.

The intermediate information that becomes available at each step is displayed in a window distributed by nine main tabs (as can be seen in Figure 6.2):

- *Tab 1*: contains the Java/JML source program to be analyzed.
- *Tab 2*: contains the syntax tree generated by the front-end.
- *Tab 3*: contains the identifiers table built during the analysis phase.
- *Tab 4, Tab 5 and Tab 6*: these three tabs are used to the verify the correctness of code with respect to contracts. The first two are used during the standard verification process through a VCGen.

  In tab 4, the rules applied along the generation of the verification conditions are displayed in tree format.

  In tab 5, the generated SMT code is shown; also a table with the verification status of each formula (`sat`, `unsat`, `unknown`) after calling a theorem-prover will be displayed.

  Tab 6 is used to perform an interactive verification.
- *Tab 7*: in this tab, the user can select which slicing algorithm to apply. After applying contract-based slicing algorithms to the original program, it will contain the new program produced by the slicer; notice that useless statements identified by the slicer are not actually removed, but shown in red and strike-out style.
- *Tab 8*: displays a LCFG as the visual representation of the program, giving an immediate and clear perception of the program complexity, with its procedures and the relationships among them.
- *Tab 9*: in this tab, the user can select which algorithm to animate. The algorithm selected will be animated through the use of the LCFG of the program, allowing the user to control the animation process.

6.4.2 Slicing with GamaSlicer

GamaSlicer implements the precondition-, postcondition- and specification-based slicing algorithms (the original version and the new one). For the latter, it also provides an animator enabling the user to see the algorithm performing in a step-by-step mode through the animation of the LCFG.

To illustrate how this works on GamaSlicer, let us start with precondition-based slicing (postcondition-based works in a similar way).

Please consider the program written in Java/JML notation in Listing 6.4.

After submitting and parsing this source code, we can start by the slicing operation, choosing the precondition-based slicing algorithm. As expected, the statements in lines 18–20 were sliced off since the precondition $P = y > 10$ makes the statements in the else branch useless. The result exhibited in GamaSlicer is depicted in Figure 6.3.

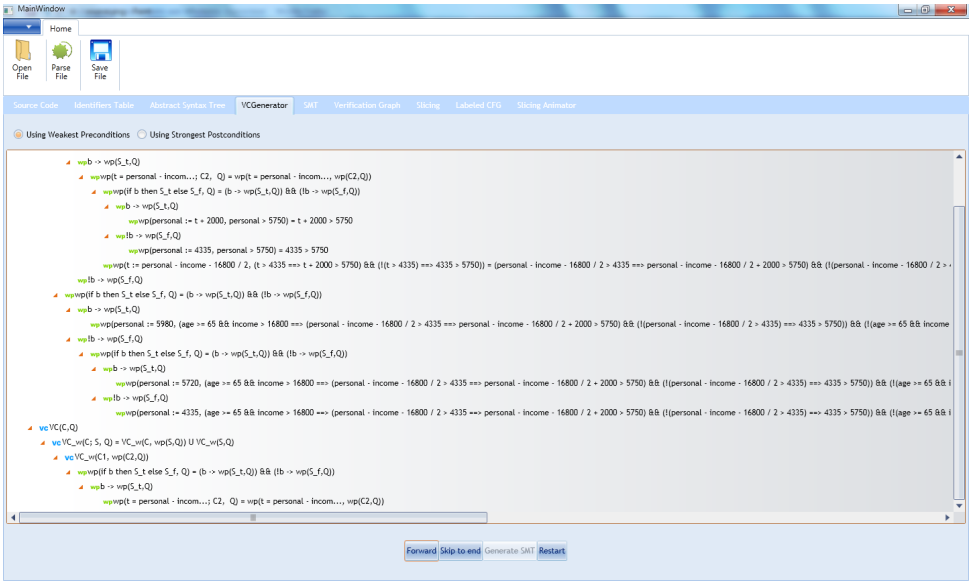

Figure 6.2: VCGenerator

Now suppose that we are interested in using both the precondition and the postcondition to slice the program, i.e., we intend to apply a specification-based slicing. This time, the resulting program is even smaller, since the statements in lines 14 and 15 are also sliced off (after line 13 the postcondition is met and thus all the statements after can be deleted). The result produced by GamaSlicer is depicted in Figure 6.4.

Consider now the annotated component in Listing 6.5 (for the sake of simplicity, a small procedure was deliberately selected).

Computing the weakest precondition and the strongest postcondition pair for each statement in the procedure 6.5, we obtain:

$$\overline{spost}_0 = x \geq 0 \qquad \overline{wprec}_1 = x > 0$$

x = x + 100

$$\overline{spost}_1 = \exists v.v \geq 0 \wedge x = v + 100 \qquad \overline{wprec}_2 = x > 100$$

x = x - 200

$$\overline{spost}_2 = \exists v.v \geq 0 \wedge x = v - 200 \qquad \overline{wprec}_3 = x > 300$$

x = x + 200

$$\overline{spost}_3 = \exists v.v \geq 0 \wedge x = v + 100 \qquad \overline{wprec}_4 = x > 100$$

(notice that $\overline{sp}_0 \equiv P$ and $\overline{wp}_4 \equiv Q$)

The specification-based slicing algorithm looks for valid implications among the $\overline{sp}_i$ and the $\overline{wp}_j$, for $0 \leq i \leq 3$ and $1 \leq j \leq 4$. For that, it will test the implications: $\overline{sp}_0 \rightarrow \overline{wp}_2$, $\overline{sp}_1 \rightarrow \overline{wp}_3$, $\overline{sp}_0 \rightarrow \overline{wp}_4$, $\overline{sp}_1 \rightarrow \overline{wp}_3$, ..., $\overline{sp}_2 \rightarrow \overline{wp}_4$.

The presented framework, GamaSlicer, incorporates different slicing algorithms that uses semantic annotations present in source code in order to slice programs in a more aggressive sense — precondition-based, postcondition-based, specification-based and contract-based slicing. GamaSlicer can also be used for program verification; in particular it allows the user to generate verification conditions in an interactive way and includes visualisation capabilities for slicing and verification.

The goal of the work here reported is to prove that the presented algorithms scale up. For that, we propose a new prototype tool that works (i.e. slices programs) at the Boogie Intermediate Language (IL) level.

```
1  public class Ifslicing
2  {
3      int x, y;
4      public Ifslicing() {}
5
6      /*@ requires y > 10;
7        @ ensures x >= 0;
8        @*/
9      public void testIF()
10     {
11         if (y > 0)
12         {
13             x = 100;
14             x = x + 50;
15             x = x - 100;
16         }
17         else {
18             x = x - 150;
19             x = x - 100;
20             x = x + 100;
21         }
22     }
23 }
```

Listing 6.4: Example for precondition and specification-based slicing

```
1  /*@ requires x >= 0;
2    @ ensures x > 100;
3    @*/
4    public int changeX() {
5      x = x + 100;
6      x = x - 200;
7      x = x + 200;
8  }
```

Listing 6.5: Simple sequence of assignments

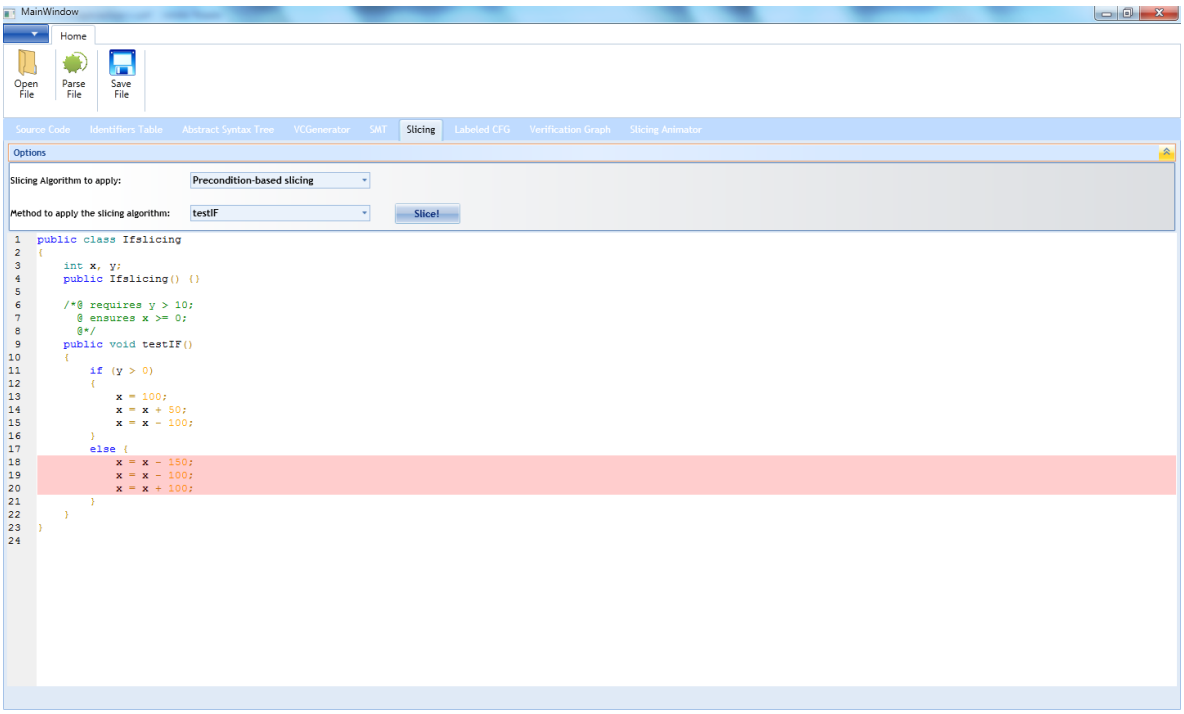

Figure 6.3: Precondition-based slicing applied to program in Listing 6.4

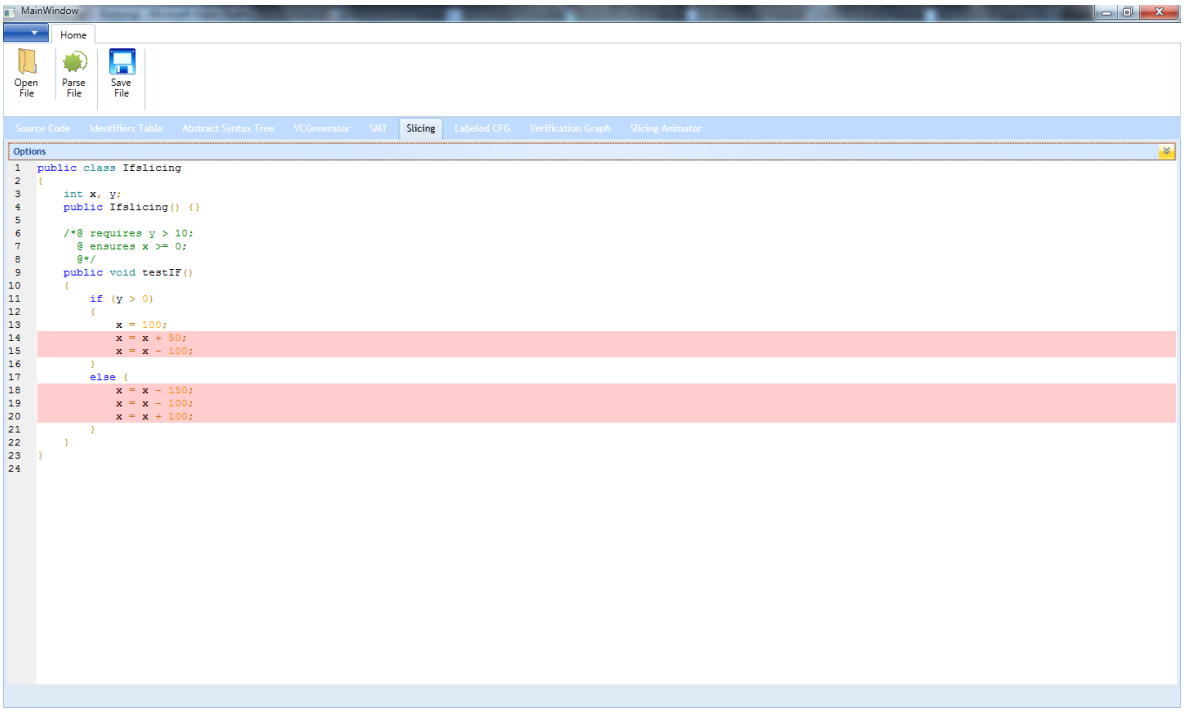

Figure 6.4: Specification-based slicing applied to program in Listing 6.4

Chapter 7

GamaBoogie

In this chapter, we introduce GamaBoogie, a tool built to implement the assertion-based slicing over Boogie programs.

The program, actually a C♯ class annotated in Spec♯, shown in Listing 7.1 will be used as example to illustrate the features of GamaBoogie. `maxarray` calculates the maximum number in an array.

As previously said, a Boogie program consists of a theory that is used to encode the semantics of the source language, followed by an imperative part.

Translating the source code of Listing 7.1 into Boogie language results in a file with a total of 1205 lines. Most of these lines corresponds to axioms and functions needed for the generation of the VCs (for instance, types disappear during the VCs generation; they are, if necessary, encoded as axioms).

```
1  public class maxarray
2  {
3      int max;
4      int[] vec = new int[100];
5
6      private void maxarray1()
7      requires vec!=null;
8      ensures 0 <= max && max <= vec.Length;
9      ensures forall{int a in (0:vec.Length) ; vec[a]<=vec[max]};
10     {
11         int i = 0;
12         max = 0;
13
14         while (i < vec.Length)
15         invariant 0 <= i && i <= vec.Length;
16         invariant 0 <= max && max <= i ;
17         invariant forall{ int a in (0:i); vec[a] <= vec[max]} ;
18         {
19             if (vec[i] > vec[max]) { max = i; }
20             i = i +1;
21         }
22     }
23 }
```

Listing 7.1: Program example: Maximum of an Array

Before starting the development of GamaBoogie, it was necessary to study Boogie language (already introduced in Chapter 4) and Boogie Verifier (it was also require to analyse its source code), in order to identify the parts to change to include slicing functionality into Boogie[1].

Figure 7.1 depicts an abstract representation of Boogie internals (a part of it) until the prover call.

[1] Appendix B contains the entire context free grammar of Boogie.

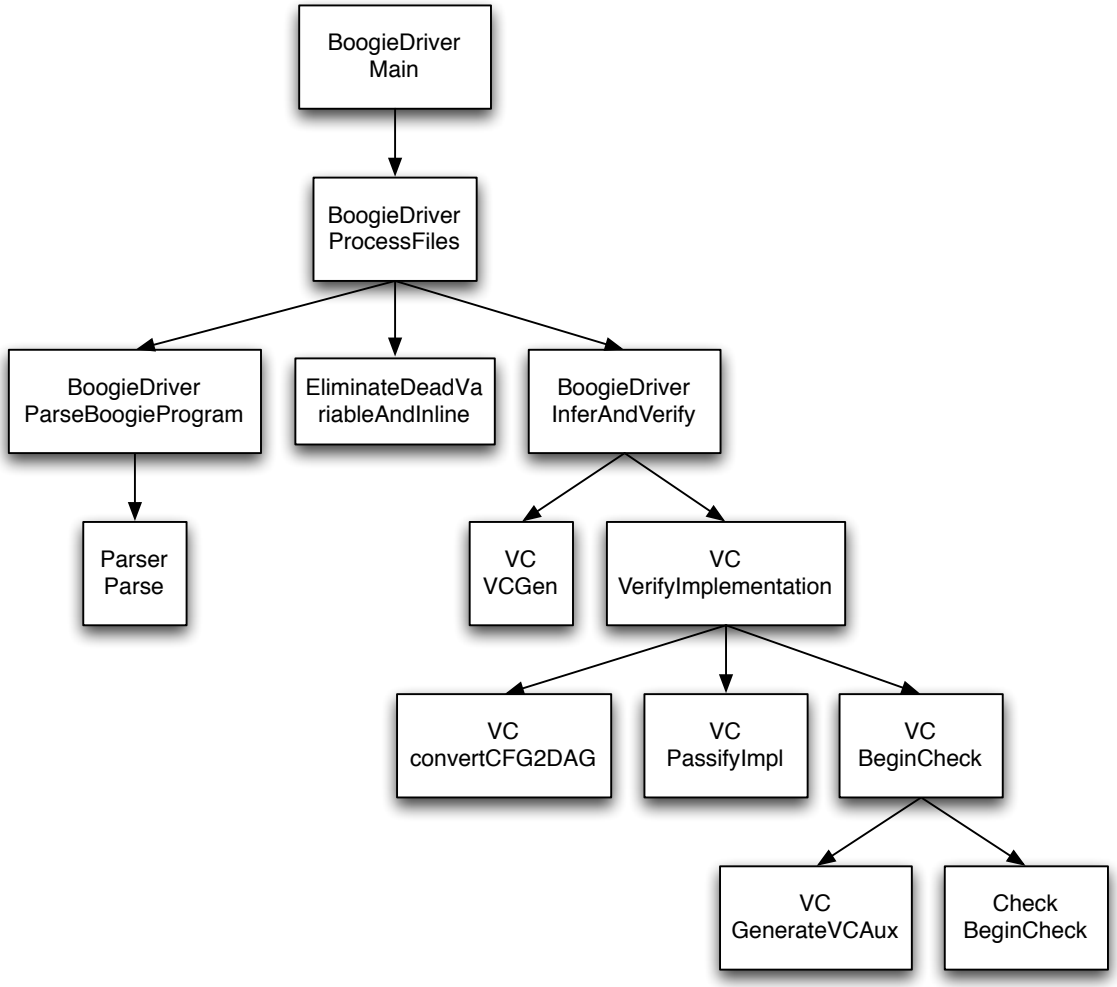

Figure 7.1: Boogie Internal Structure

One of the things learned during this study, was that Boogie already builds an internal control flow graph for each implementation in a Boogie program. In fact, it constructs both the Control Flow Graph (CFG) and Directed Acyclic Graph (DAG).

The CFG for program in Listing 7.1 is shown in Figure 7.2 and the DAG in Figure 7.3. The names in the graph are generated automatically by Boogie.

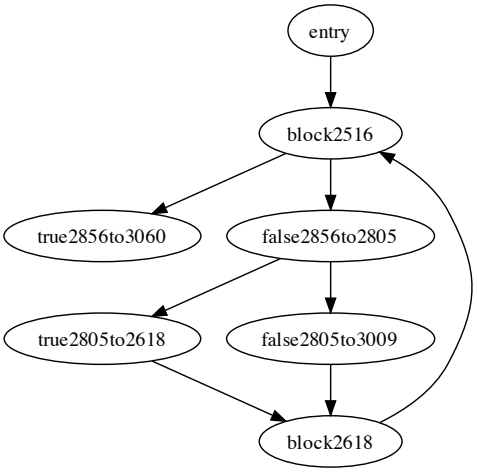

Figure 7.2: CFG of `maxarray`

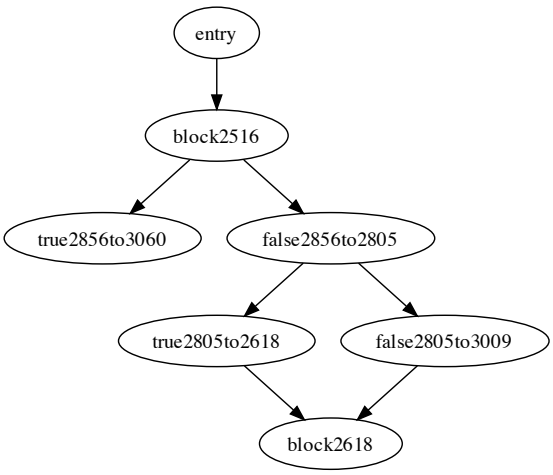

Figure 7.3: DAG of `maxarray`

We also found that Boogie eliminates the empty blocks, with the method `EliminateDeadVariablesAndInline`. The CFG in Figure 7.4 represents the CFG before the elimination of empty *blocks*.

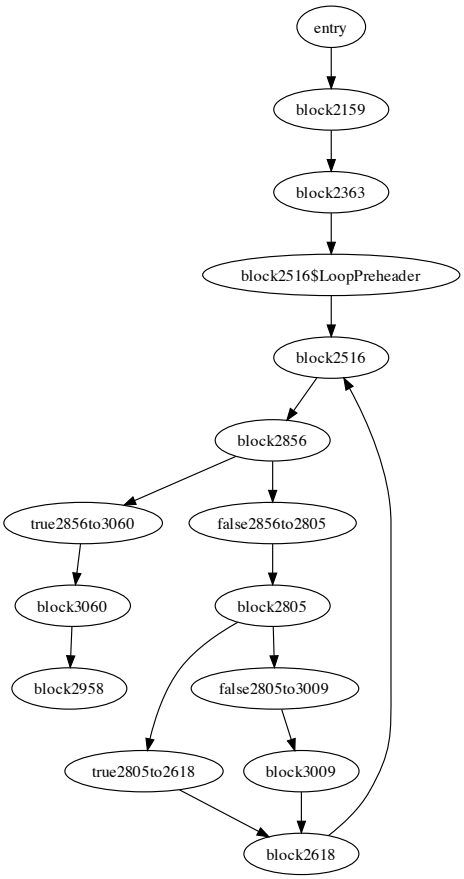

Figure 7.4: CFG of `maxarray` with dead blocks

In the rest of the chapter, we give an overview of GamaBoogie, discussing the two visualising features, Editor and Visual Inspector in Section 7.1, and the third feature, Boogie Slicer, in Section 7.3. To give a theoretical support for this last feature, Section 7.2 revisits the *assertion-based slicing* algorithms introduced in Chapter 6 to describe the adaptations done in the context of GamaBoogie.

7.1 GamaBoogie Architecture

Figure 7.5 depicts the GamaBoogie architecture. Basically it consists of three components: an Editor, a Visual Inspector and a Slicer Module. Each one of these components will be explained in the following subsections.

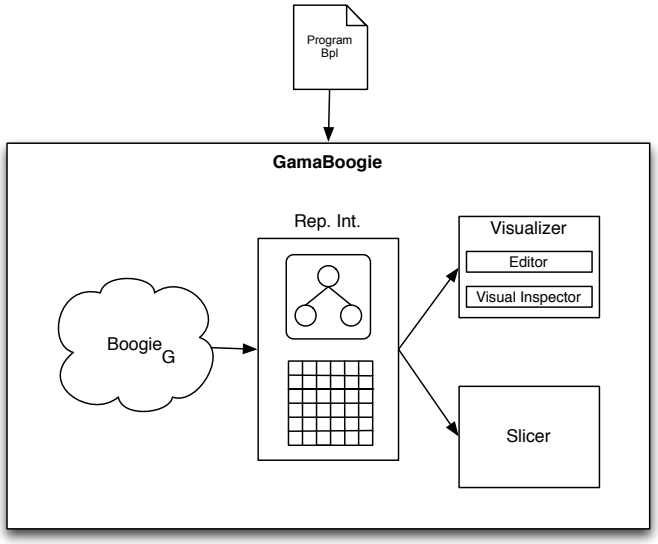

Figure 7.5: GamaBoogie Architecture

7.1.1 Editor

As can be seen in Figure 7.6, the editor component has two parts, the Spec# Source Code and Boogie Source Code editors.

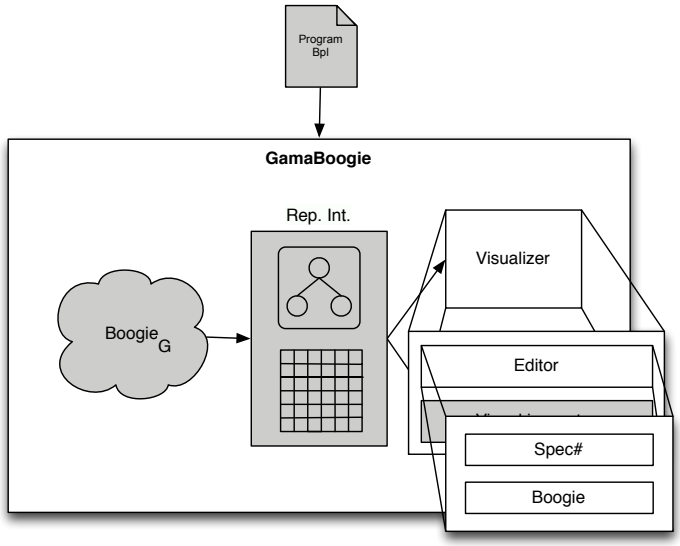

Figure 7.6: GamaBoogie Editor component architecture

In order to display the code to the user in a friendly manner, we are using the ScintillaNet[2] component for syntax highlighting. With this feature, the user can get a better visualization of the source code and explore it.

Source Code Editor for Spec#

With this source code editor (see Figure 7.7) we can create or edit a Spec# file and then verify if the specification is correct or not. If correct the Spec# file can be translated to the Boogie language. If not correct a list of the errors appears in the bottom. When the user clicks an item in the error list, the corresponding line in the source code is shown.

Source Code Editor for Boogie

We can load a Boogie file and run Boogie to parse the program.

Figure 7.8 depicts syntax highlighting feature. As can be seen at right, the relevant information about the program is shown — the procedures, marked with a red P, and the implementations of that procedures, marked with a green I. This way, the user can inspect each one of these entities by clicking on it, and thus being redirected to the line that corresponds to such declaration.

Besides that, the user can hide the code automatically generated by Boogie with respect to types and axioms, displaying only the code for procedures and implementations. This allows the user to focus only on the code he is interested in.

As in the previous editor for Spec#, we also have a list of errors at the bottom of the window.

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

```
1  public class maxarray
2  {
3      static int LENGTH = 100;
4      int max;
5      int[] vec = new int[LENGTH];
6
7      private void maxarray1()
8      requires vec!=null;
9      ensures 0 <= max && max <= LENGTH ;
10     ensures forall{int a in (0:LENGTH) ; vec[a]<=vec[max]};
11     {
12         int i = 1;
13         max = 0;
14
15         while (i < LENGTH)
16         invariant 1 <= i && i <= LENGTH ;
17         invariant 0 <= max && max < i ;
18         invariant forall{ int a in (0:i); vec[a] <= vec[max]} ;
19         {
20             if (vec[i] > vec[max]) { max = i; }
21                 i = i +1;
22         }
23     }
24 }
25
```

Figure 7.7: Spec# Source Code editor

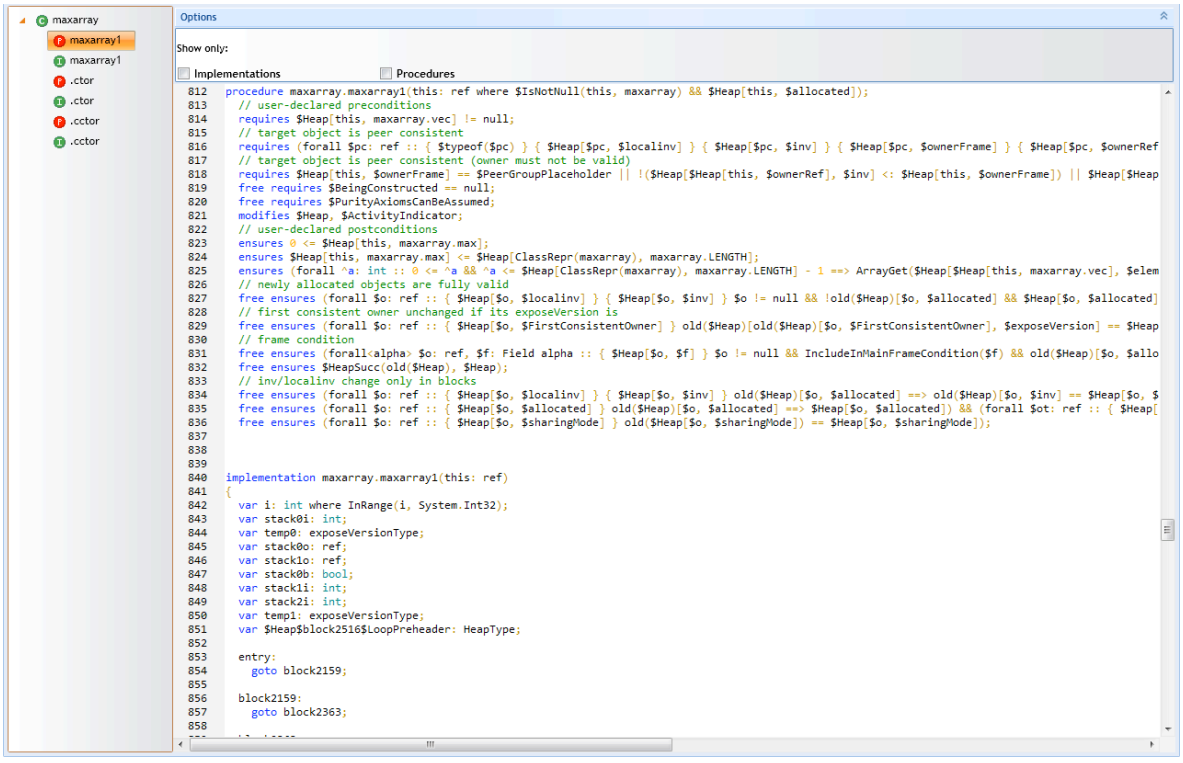

Figure 7.8: Boogie Source Code editor

7.1.2 Visual Inspector

Figure 7.9 shows the zoomed-in architecture of GamaBoogie with respect to the Visual Inspector component. This component provides visual representation for the following Boogie elements : identifier table; flow graphs; passified commands for a block; weakest preconditions for a block; and information sent to the prover.

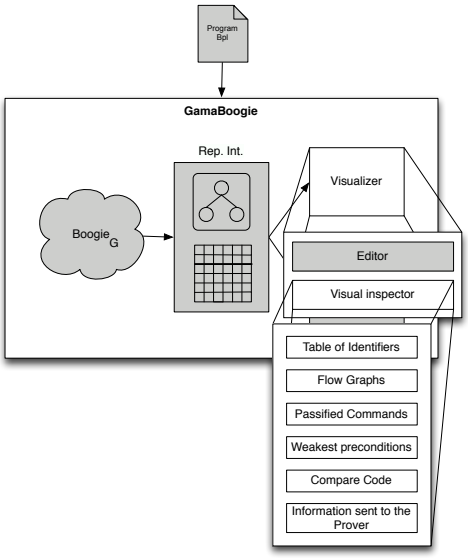

Figure 7.9: GamaBoogie Visual component architecture

Identifier Table

After loading a Boogie program, its Identifier Table is shown. This table is built during the parsing phase, and contains information about the identifiers declared in each implementation. The information displayed in this table includes:

- Name of variable,
- Class where it belongs,
- Method where it is declared,
- Type,
- Line where it is declared,

Figure 7.10 shows the Identifier Table for the program in Listing 7.1.

Flow Graphs

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

Identifier	Class	Method	Type	Line
stack1o	maxarray	maxarray1	ref	846
temp0	maxarray	.ctor	ref	1080
stack0o	maxarray	maxarray1	ref	845
stack0o	maxarray	.ctor	ref	1079
temp0	maxarray	maxarray1	exposeVersionType	844
stack0i	maxarray	maxarray1	int	843
stack0i	maxarray	.cctor	int	1184
i	maxarray	maxarray1	int	842
stack0b	maxarray	maxarray1	bool	847
SHeapSblock2516SLoopPreheader	maxarray	maxarray1	HeapType	851
temp1	maxarray	maxarray1	exposeVersionType	850
stack2i	maxarray	maxarray1	int	849
stack1i	maxarray	maxarray1	int	848
stack0i	maxarray	.ctor	int	1078
temp1	maxarray	.ctor	exposeVersionType	1081

Figure 7.10: Part of the Identifier Table

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

The user can filter, once again, the information he wants to see: he can start with a global perspective of the graph and then go deeper by inspecting the control flow of a block inside the implementation. Figure 7.11 shows the CFG of an implementation and Figure 7.12 shows the CFG of a block inside such implementation, this last one can only be obtained from a DAG of an implmentation, Figure 7.13.

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

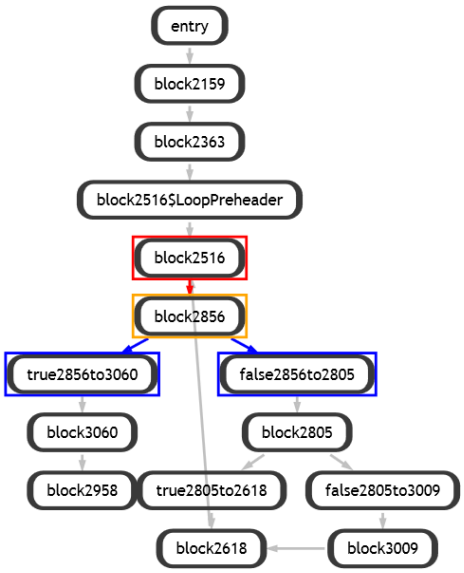

Figure 7.11: Block Flow Graph window

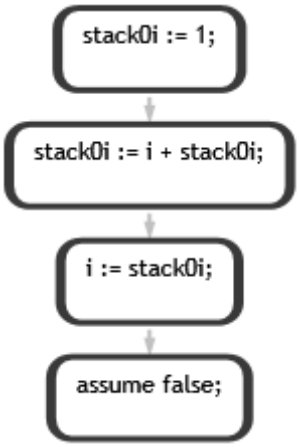

Figure 7.12: Command Flow Graph window

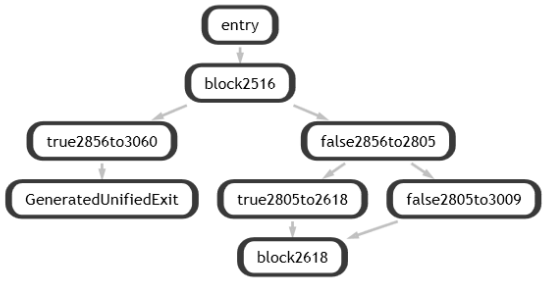

Figure 7.13: Block Acyclic Graph window

Passified Commands

One of the main things that we felt need to see was the program in its passified form, since this was a crucial step to the slicing part.

The assertion-based slicing algorithms proposed in Chapter 6 were no longer directly applicable, because the way how weakest preconditions are calculated on Boogie differs from the ones calculated in GamaSlicer (while Boogie computes them over a single-assignment program, where the commands are in the guarded commands language, GamaSlicer computes them directly over Java statements).

For the Listing 7.1, the passified commands are shown in Figure 7.14.

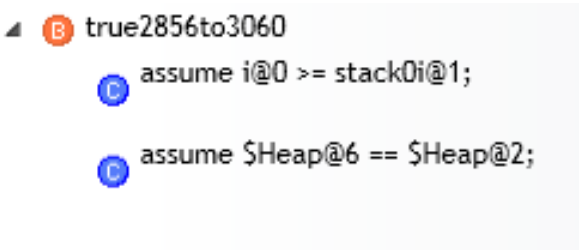

Figure 7.14: Passified Commands window

This deep inspection of passified programs has enabled us to understand which modifications to the original slicing algorithms will be needed to do.

Weakest preconditions

Another crucial step to the adaption of assertion-based slicing algorithms is the calculation of weakest preconditions performed by Boogie.

To obtain this information, we had to modify the original Boogie, since the weakest preconditions were only calculated at the time of the verification of a program. However, we would like to compute these WLPs without the need of checking the correctness of a program.

In order to get the WLP we have to modify the class of `Implementation` to have another method to compute the WLP.

Figure 7.15 shows, in gray, which Boogie modules (see Figure 7.1) were changed by us.

In order to make easier to inspect and comprehend the weakest preconditions calculated from

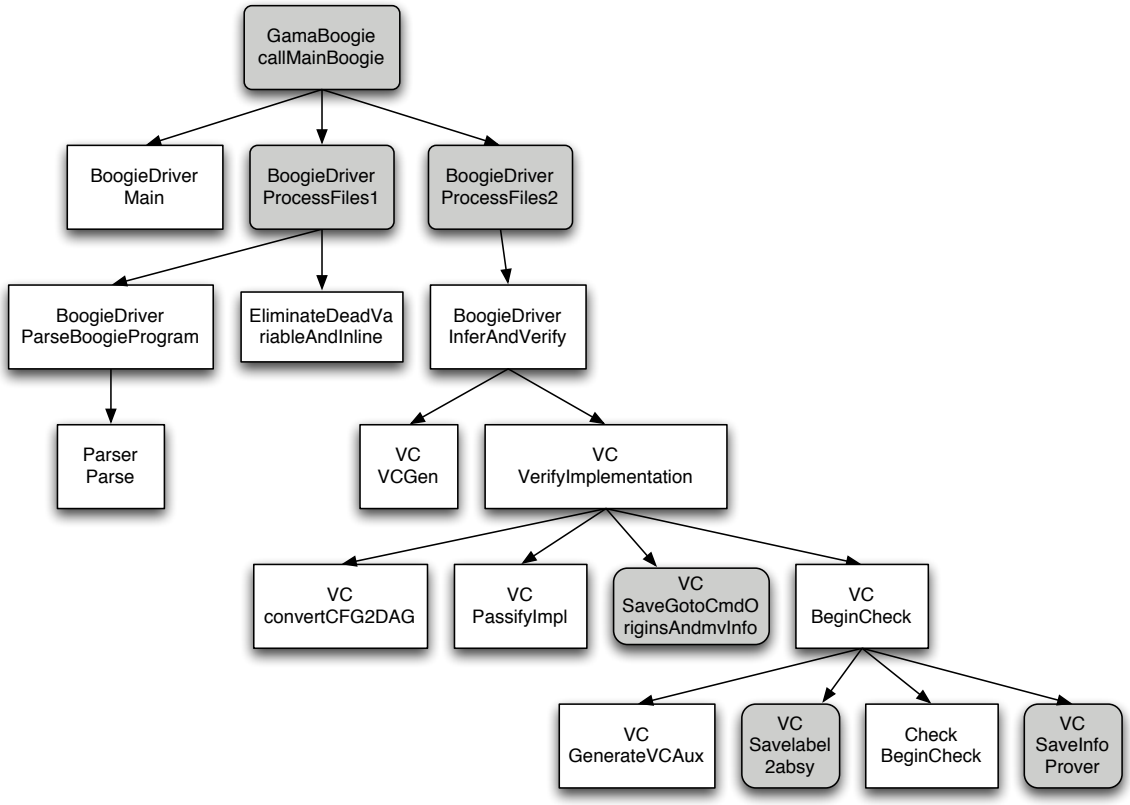

Figure 7.15: GamaBoogie Data Flow Graph

a given implementation, we added another feature in GamaBoogie: for each implementation, it is shown a tree with its WLP.

On top of the tree we have the entire WP and below we have its structure. Figure 7.16 depicts the weakest precondition window for Listing 7.1.

Figure 7.16: Weakest Precondition window

Finally, there is a window where the weakest preconditions and the passified commands are related. Figure 7.17 shows that relation for Listing 7.1.

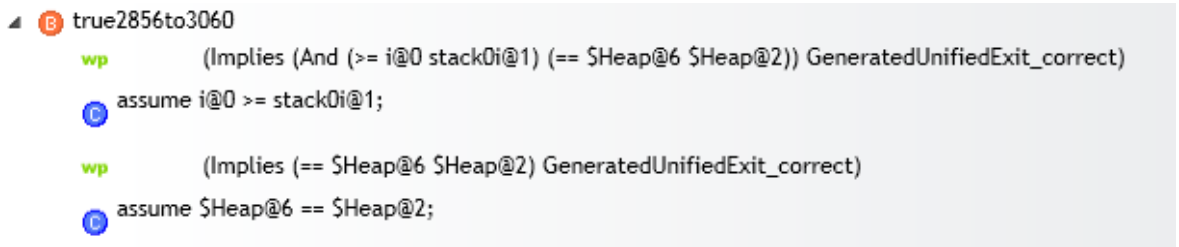

Figure 7.17: Weakest Preconditions & Passify Commands window

Information sent to the Prover

With this feature, we can see (look at Figure 7.18) the information that is sent to the prover, including the prover used to make the correctness proof of the program and the options used to

invoke such prover.

To display the information that is sent to the prover we created four new methods:

- `getProverName` - Contains the name of the Prover
- `getProverOptions` - Contains all the command options of the Prover
- `getProverInfo` - Contains all the information send to the Prover, expect the VC.
- `getProverVcString` - Contains the VC.

`ProverInterface` is an abstract class, so we had to create these methods in all existing Boogie provers. To get this, we modify the constructor `Implementation` to have a list with this information.

```
Prover:      C:\Program Files (x86)\Microsoft Research\Z3-2.15\bin\z3.exe
Options:       /si /@ /cex:5 /t:0
1  (SETPARAMETER MODEL_PARTIAL true)(SETPARAMETER MODEL_HIDE_UNUSED_PARTITIONS false)(SETPARAMETER MODEL_V1 true)(SETPAR
2  ; Boogie 2 universal background predicate for Z3 (Simplify notation with types)
3  ; Copyright (c) 2004-2009, Microsoft Corp.
4
5  (DEFTYPE $int :BUILTIN Int)
6  (DEFTYPE $bool :BUILTIN bool)
7  (DEFTYPE U)
8  (DEFTYPE T)
9
10 (DEFOP <: U U $bool)          ; used for translation with type premisses
11 (DEFOP <:: T U U $bool)       ; used for translation with type arguments
12
13 (DEFOP tickleBool $bool $bool) ; used in triggers to exhaustively instantiate quantifiers over booleans
14
15 (BG_PUSH (AND
16
17   ; false is not true
18
19   (DISTINCT |@false| |@true|)
20
21   ; we assume type correctness of the operations here
22   ; a-l>=0 ==> (v ++ w:l)[a:b] = v[a-l:b-l]
23   (FORALL (v lv w lw lvw a b)
24     (QID bv:e:c1)
25     (PATS ($bv_extract ($bv_concat v lv w lw) lvw a b))
26     (IMPLIES
27       (>= (- a lw) 0)
28       (EQ ($bv_extract ($bv_concat v lv w lw) lvw a b) ($bv_extract v lv (- a lw) (- b lw)))))
29
30 ; b<=l ==> (v ++ w:l)[a:b] = w[a:b]
```

Figure 7.18: Information sent to the Prover window

7.2 Slicing Algorithms

As previously referred, the biggest challenge was to adapt the original assertion-based slicing algorithms to work with Boogie programs due to the passivization technique.

Both precondition- and postcondition-based slicing algorithms can be seen as special cases of specification-based slicing.

- To compute a precondition-based slicing when only a precondition is given, we can compute a specification-based slicing with respect to the contract $(P, \text{spost}(P))$. This means, we consider as postcondition, the strongest postcondition computed from the precondition given.

- To compute a postcondition-based slicing when only a postcondition is given, we can compute a specification-based slicing with respect to the contract $(\mathsf{wprec}(Q), Q)$. This means, we consider as precondition, the weakest precondition computed from the postcondition given.

Thus, in GamaBoogie we start to implement to adapt the specification-based slicing algorithm tow work with Boogie programs.

Because Boogie does not compute the strongest postconditions for a program, the first step was to extend Boogie to do such computations.

However, after the passive form of a program is obtained all assignments `x := e` are transformed into assumptions of the form `assume x=e`. At this point, a command is either **assert** ψ or **assume** ψ, where ψ is some first-order logic formula. Thus, the strongest postcondition of a given command is given by [38]:

$$\mathsf{sp}(\_\psi, P) = P \wedge \psi$$

Next step was the algorithm adaptation. The first problem found when doing this was when we try to find valid implications among the strongest postconditions and weakest preconditions, we can not find a direct link between the variables in the left side of the implication with the ones in right side due to the passivization step.

To illustrate the idea consider the program in Listing 7.2.

```
1  class test {
2      static void Main(string[] args)
3      {
4      }
5
6      void tester(int x)
7          requires x >= 0;
8      {
9          x = x + 100;
10         x = x - 200;
11         x = x + 200;
12         assert x >= 100;
13     }
14 }
```

Listing 7.2: Program example: Source Example 2

The strongest postconditions and the weakest preconditions for this program would be:

$$\overline{spost}_0 = x \geq 0 \qquad \overline{wprec}_1 = x > 0$$

x = x + 100

$$\overline{spost}_1 = \exists v.v \geq 0 \wedge x = v + 100 \qquad \overline{wprec}_2 = x > 100$$

x = x - 200

$$\overline{spost}_2 = \exists v.v \geq 0 \wedge x = v - 200 \qquad \overline{wprec}_3 = x > 300$$

x = x + 200

$$\overline{spost}_3 = \exists v.v \geq 0 \wedge x = v + 100 \qquad \overline{wprec}_4 = x > 100$$

And when performing the specification-based slicing algorithm, we will try to prove the validity of the following implications:

- $\overline{spost}_0 \rightarrow \overline{wprec}_2 \equiv x \geq 0 \rightarrow x > 100 \equiv \bot$
- $\overline{spost}_0 \rightarrow \overline{wprec}_3 \equiv x \geq 0 \rightarrow x > 300 \equiv \bot$

- $\overline{spost}_0 \rightarrow \overline{wprec}_4 \equiv x \geq 0 \rightarrow x > 100 \equiv \bot$
- $\overline{spost}_1 \rightarrow \overline{wprec}_4 \equiv \exists v.v \geq 0 \wedge x = v + 100 \rightarrow x > 100 \equiv \top$
- ...

Because the implication $\overline{spost}_1 \rightarrow \overline{wprec}_4$ is valid, we can slice off instructions in lines 2 and 3. However, when considering a program in its passified form, these implications does not make sense.

For the program above, we will have the following strongest postconditions:

$$
\begin{array}{ll}
\overline{spost}_0 = x_0 \geq 0 & \\
 & \mathbf{assume\ x_1} = x_0 + 100 \\
\overline{spost}_1 = x_0 \geq 0 \wedge x_1 = x_0 + 100 & \\
 & \mathbf{assume\ x_2} = x_1 - 200 \\
\overline{spost}_2 = x_0 \geq 0 \wedge x_1 = x_0 + 100 & \\
\wedge\ x_2 = x_1 - 200 & \\
 & \mathbf{assume\ x_3} = x_2 + 200 \\
\overline{spost}_3 = x_0 \geq 0 \wedge x_1 = x_0 + 100 & \\
\wedge\ x_2 = x_1 - 200 \wedge x_3 = x_2 + 200 &
\end{array}
$$

And the following weakest preconditions:

$$
\begin{array}{ll}
 & \overline{wprec}_1 = x_1 = x_0 + 100 \rightarrow x_2 = x_1 - 200 \rightarrow \\
 & x_3 = x_2 + 200 \rightarrow x_3 > 100 \\
\mathbf{assume\ x_1} = x_0 + 100 & \\
 & \overline{wprec}_2 = x_2 = x_1 - 200 \rightarrow x_3 = x_2 + 200 \rightarrow x_3 > 100 \\
\mathbf{assume\ x_2} = x_1 - 200 & \\
 & \overline{wprec}_3 = x_3 = x_2 + 200 \rightarrow x_3 > 100 \\
\mathbf{assume\ x_3} = x_2 + 200 & \\
 & \overline{wprec}_4 = x_3 > 100
\end{array}
$$

One of the advantages when considering the strongest postconditions of passified programs, is the absence of existential quantifiers (usually, the presence of existential quantifiers in a formula poses a problem to the provers).

After compute both the spost and the wprec for the program above, one of the implications we will try to prove is:

$$
\begin{array}{rl}
 & \overline{spost}_1 \rightarrow \overline{wprec}_4 \\
\equiv & (x_0 \geq 0 \wedge x_1 = x_0 + 100) \rightarrow (x_3 > 100)
\end{array}
$$

But the prover will return *invalid*, as we do not have anything relating the variables on the left side (x_0 and x_1) with the one in right side (x_3). At this point, we need some kind of "magic" to relate the variables. Thus, we have found that we can make them be related by performing the following transformation over the implication that will be sent to the prover:

- Get the last variable assigned at left side (in this case, x_1) and store it in `lastVar`;
- Get the last command with the same index occurring in the weakest precondition and get the variable name (in this case, x_3) and store it in `subsVar`;

- Replace every occurrence of `subsVar` by `lastVar` in the right side of the implication (in this case, will result in $x_1 > 100$).

After perform these steps, the final implication will be:

$$\begin{aligned} & \overline{spost}_1 \rightarrow \overline{wprec}_4 \\ \equiv \quad & (x_0 \geq 0 \wedge x_1 = x_0 + 100) \rightarrow (x_1 > 100) \end{aligned}$$

This implication is *valid* and as expected, we can slice of instructions in lines 2 and 3.
Considering other valid implication:

$$\begin{aligned} & \overline{spost}_1 \rightarrow \overline{wprec}_3 \\ \equiv \quad & (x_0 \geq 0 \wedge x_1 = x_0 + 100) \rightarrow (x_3 = x_2 + 200 \rightarrow x_3 > 100) \end{aligned}$$

Performing the transformations we get:

$$\begin{aligned} & \overline{spost}_0 \rightarrow \overline{wprec}_3 \\ \equiv \quad & (x_0 \geq 0 \wedge x_1 = x_0 + 100) \rightarrow (x_2 = x_1 + 200 \rightarrow x_2 > 100) \end{aligned}$$

Because the implication $\overline{spost}_0 \rightarrow \overline{wprec}_3$ is valid, it means that we can slice off, alternatively, instructions in lines 2 and 3 (after remove it, the postcondition still holds in the final state of the program).

We have also that $\overline{spost}_1 \rightarrow \overline{wprec}_3$, and thus we can slice off, alternatively, only the instruction in line 2. However, the best slice is the one that removes the highest number of instructions.

In the next section, we will show how to slice Boogie programs using GamaBoogie.

7.3 Slicing in GamaBoogie

The Slicer is the most important feature expected to be delivered during this master work (see Figure 7.19) to understand that component architecture.

After slicing, we generate two views to display the results: one over Boogie code and the other related to the source code in Spec♯. Concerning the first view, GamaBoogie creates two windows. The first exhibiting the complete Boogie program enhancing the sliced statements with a different colour, and the second displaying only the new program after removing from the source the sliced statements. Concerning the Spec♯ view, the approach followed is similar to the last one; just the new program without the slice is displayed.

Figure 7.20 gives a flavour of the GamaBoogie GUI.

To use GamaBoogie Slicer we have to choose the slicing algorithm and the Boogie implementation to work on. From the implementation, we get the procedure and extract the precondition (requires) and the postconditions (ensure). To ensure that the original pre- and postconditions are extracted we analyse the comments associate with the commands; if the comment is the string ”user-declared preconditions” or ”serialized AssertStatement” we deduce that the associate command this to be considered because it belongs to the original specification.

After selecting the implementation and slicing algorithm, the prover is used to perform the slicing and then the following four windows are available:

Original Boogie code This window shows the integral Boogie code emphasising the lines that should disappear in different colour, see Figure 7.21;

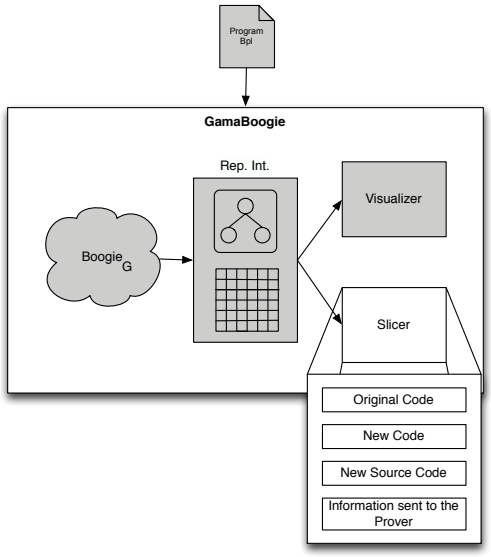

Figure 7.19: GamaBoogie Slicer component architecture

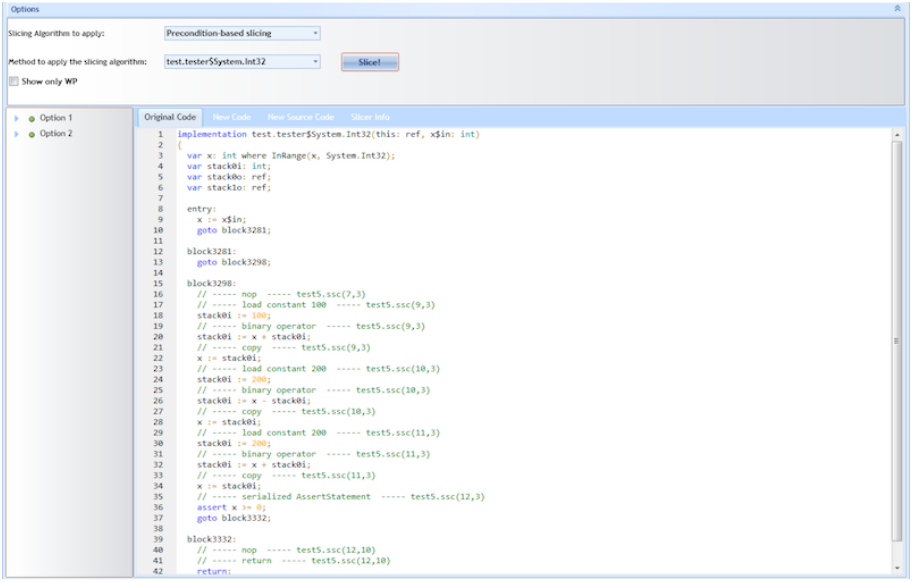

Figure 7.20: GamaBoogie Slicer overview

New Boogie Code This window just displays the Boogie sliced code, see Figure 7.22;

New Source Code This window just displays the Spec♯ sliced code, see Figure 7.23;

Information to the Prover Shows all the information that is sent to the prover during the slicing process, see Figure 7.24.

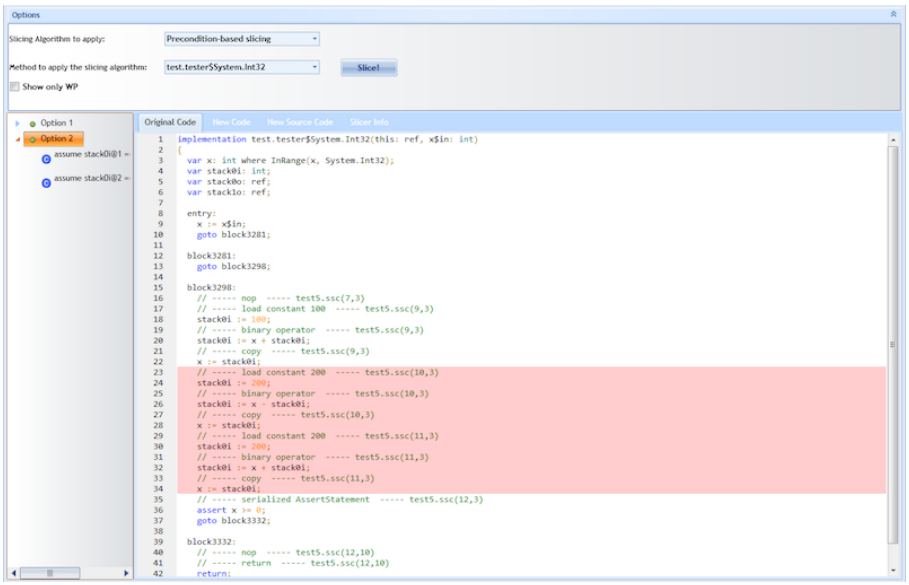

Figure 7.21: GamaBoogie Slicer Original Code

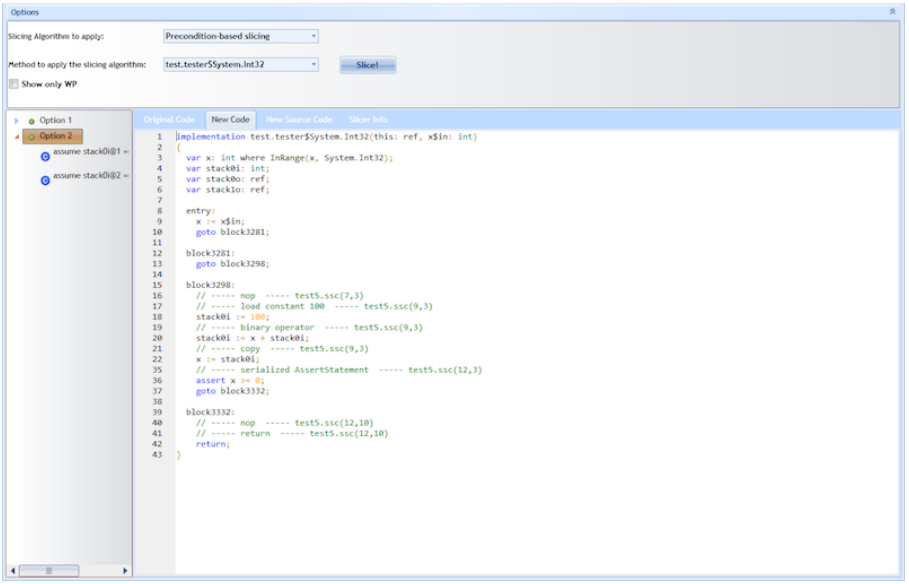

Figure 7.22: GamaBoogie Slicer New Code

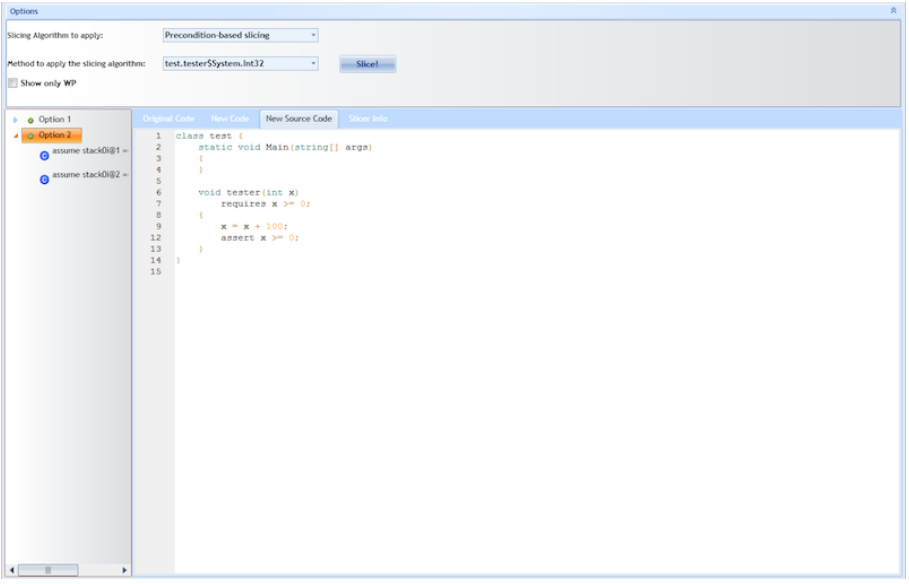

Figure 7.23: GamaBoogie Slicer New Source Code

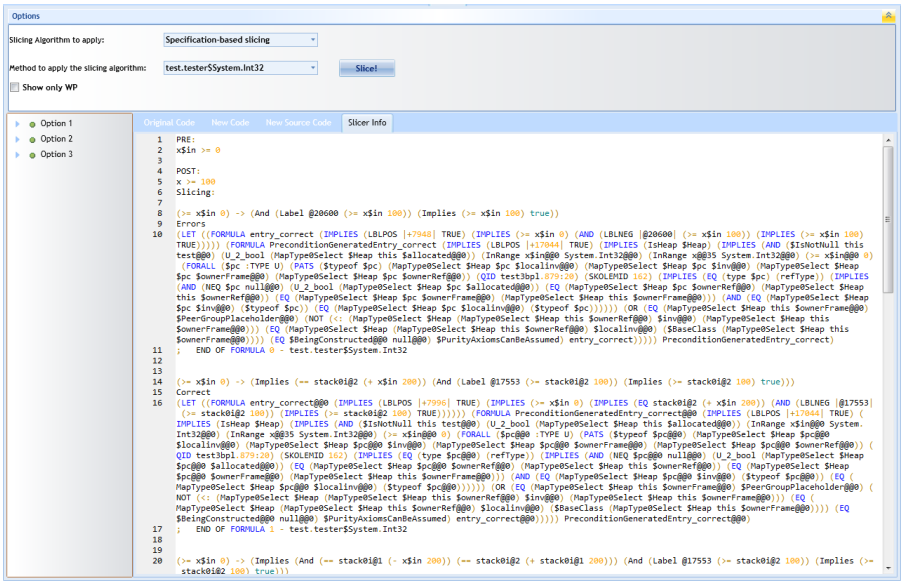

Figure 7.24: GamaBoogie Information to the Prover

7.4 Summary

This chapter was devoted to the main contribution of this master thesis: the GamaBoogie tool. As the theoretical background was presented in previous chapters, the GamaBoogie overview and details were illustrated using screenshots of the tool.

Chapter 8

Conclusion

In this dissertation, we have shown the importance of Program Verification (PV). As more and more complex and critical software systems are built, a stronger demand for automatic verification of programs arises, for the sake of everyone's security. Although dynamic approaches, like software testing, have been mentioned, the focus along the dissertation was on manual and semi-automatic approaches to ensure that a program is correct according to its specification.

After reviewing these concepts of PV in Chapter 2 and Chapter 3., we introduced Boogie system and language in Chapter 4. It was possible to understand that Boogie is very powerful, because of the expressiveness of Boogie Program Language (BoogiePL) that can be considered an adequate intermediate language for several specification (contract-based) languages; also the passification is very important, as well as the use of several provers.

We have also seen that Slicing (Chapter 5) is a technique that contributes in many ways to the software life cycle. We have distinguished static from dynamic slicing. In recent years, new algorithms were developed for slicing, but this time based on contracts, as presented along Chapter 7.

A tool was presented (Chapter 7) to slice of Boogie programs. The tool is capable of applying the Contract-based Slicing approach to a program and compute precise slices. In addition to this main feature, GamaBoogie includes an editor and a visual inspector for Boogie programs. This new features, added during the work to satisfy our needs, proved to be very useful for the comprehension of Boogie code.

However the tool still presents some limitations in the contract-based slicing. Basically because of the replacement method, that has to be refined in order to accept only numerical variables operations.

8.1 Future Work

As future work, we intend to:

- improve GamaBoogie main algorithms;
- improve the generation of BoogiePL from Spec♯ source code;
- display the graph of contract-based slicing.

Bibliography

[1] W.W. Adams and P. Loustaunau. *An introduction to Gröbner bases*. Amer Mathematical Society, 1994. ISBN 0821838040.

[2] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

[3] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw., Pract. Exper.*, 23(6):589–616, 1993.

[4] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The key tool. *Software and System Modeling*, 4(1):32–54, 2005.

[5] Wolfgang Ahrendt, Bernhard Beckert, Martin Giese, and Philipp Rümmer. Practical aspects of automated deduction for program verification. *KI*, 24(1):43–49, 2010.

[6] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, 2005.

[7] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Michael D. Ernst and Thomas P. Jensen, editors, *PASTE*, pages 82–87. ACM, 2005. ISBN 1-59593-239-9.

[8] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS : construction and analysis of safe, secure, and interoperable smart devices*, volume 3362, pages 49–69. Springer, Berlin, March 2004.

[9] Jose Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, pages 93–102, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4153-2. doi: http://dx.doi.org/10.1109/SEFM.2010.18. URL http://dx.doi.org/10.1109/SEFM.2010.18.

[10] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 2010.

[11] B. Beizer and J. Wiley. Black box testing: Techniques for functional testing of software and systems. *Software, IEEE*, 13(5):98, 2002. ISSN 0740-7459.

[12] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/2363.2366.

[13] A. Beszedes and T. Gyimothy. Union slices for the approximation of the precise slice, 2002.

[14] A. Beszedes, C. Farago, Z. Szabo, J. Csirik, and T. Gyimothy. Union slices for program maintenance, 2002. URL citeseer.ist.psu.edu/article/beszedes02union.html.

[15] David Binkley. The application of program slicing to regression testing. *Information & Software Technology*, 40(11-12):583–594, 1998.

[16] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005. ISSN 1433-2779. doi: http://dx.doi.org/10.1007/s10009-004-0167-4.

[17] James Burns and Jean-Luc Gaudiot. Smt layout overhead and scalability. *IEEE Trans. Parallel Distrib. Syst.*, 13(2):142–155, 2002.

[18] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information & Software Technology*, 40(11-12):595–607, 1998.

[19] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2007. ISBN 978-3-540-71208-4.

[20] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM. ISBN 1-58113-287-5. doi: http://doi.acm.org/10.1145/372202.372784.

[21] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance*, 8:145–178, May 1996. ISSN 1040-550X. doi: 10.1002/(SICI)1096-908X(199605)8:3<145::AID-SMR127>3.3.CO;2-0. URL http://portal.acm.org/citation.cfm?id=250750.250751.

[22] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME*, volume 1051 of *Lecture Notes in Computer Science*, pages 557–575. Springer, 1996. ISBN 3-540-60973-3.

[23] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006. URL http://ergo.lri.fr/papers/ergo.ps.

[24] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gamaslicer: an online laboratory for program verification and analysis. In *LDTA '10: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, pages 1–8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0063-6. doi: http://doi.acm.org/10.1145/1868281.1868284.

[25] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Contract-based slicing. In *ISoLA'10 — Fourth International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (FLDVES track)*, Creta, Greece, Oct 2010.

[26] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. Vcc: Contract-based modular verification of concurrent c. In *ICSE Companion*, pages 429–430, 2009.

[27] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008. doi: 10.1007/978-3-540-78800-3_24. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[28] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. ISBN 978-3-540-78799-0.

[29] Robert DeLine and K. Rustan M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical report, May 2005.

[30] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[31] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[32] Manuel Fähndrich. Static verification for code contracts. In Radhia Cousot and Matthieu Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2010. ISBN 978-3-642-15768-4.

[33] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. doi: http://doi.acm.org/10.1145/1774088.1774531. URL http://doi.acm.org/10.1145/1774088.1774531.

[34] C. Flanagan and K. Leino. Houdini, an annotation assistant for ESC/Java. *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517, 2001.

[35] Karin Freiermuth. Using program slicing to improve error reporting in boogie. Master's thesis, ETH Zurik, September 2007.

[36] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Eng.*, 17(8):751–761, 1991.

[37] D. Gray. A pedagogical verification condition generator. *Comput. J.*, 30(3):239–248, 1987.

[38] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 6:1–6:7, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-540-6. doi: http://doi.acm.org/10.1145/1557898.1557904. URL http://doi.acm.org/10.1145/1557898.1557904.

[39] R.J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2:33–53, 1995. An algorithm to automatically extract a correctly functioning subset of the code of a system is presented. The technique is

based on computing a simultaneous dynamic program slice of the code for a set of representative inputs. Experiments show that the algorithm produces significantly smaller subsets than with existing methods.

[40] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Softw. Test., Verif. Reliab.*, 5(3):143–162, 1995.

[41] Mark Harman, Robert M. Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *ICSM*, pages 138–147, 2001.

[42] P. V. Homeier and D. F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, 1995. doi: 10.1093/comjnl/38.2.131. URL http://comjnl.oxfordjournals.org/content/38/2/131.abstract.

[43] I. Jager and D. Brumley. Efficient Directionless Weakest Preconditions (CMU-CyLab-10-002). *CyLab*, page 27, 2010.

[44] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384. Springer Berlin / Heidelberg, 1992. URL http://dx.doi.org/10.1007/3-540-55844-6_148.

[45] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[46] Bogdan Korel and Janusz W. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[47] Arun Lakhotia. Rule-based approach to computing module cohesion. In *ICSE*, pages 35–44, 1993.

[48] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2004.

[49] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[50] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. ISBN 978-3-642-17510-7.

[51] K. Rustan M. Leino. Learning to do program verification. *Commun. ACM*, 53(6):106, 2010.

[52] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009. ISBN 978-3-642-03828-0.

[53] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviors through program slicing. In *WPC*, pages 9–10. IEEE Computer Society, 1996.

[54] Bertrand Meyer. Applying ”design by contract”. *IEEE Computer*, 25(10):40–51, 1992.

[55] Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, 1992. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.161279.

[56] K. Rustan M.Leino. *This is Boogie 2*. Microsoft Research, Redmond, WA, USA, June 2008.

[57] L.M. Ott and J.J. Thuss. Slice based metrics for estimating cohesion. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 71 –81, May 1993. doi: 10.1109/METRIC.1993.263799.

[58] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007. ISBN 0978739256.

[59] W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. *status: published*.

[60] Josep Silva. Debugging techniques for declarative languages: Profiling, program slicing and algorithmic debugging. *AI Commun.*, 21(1):91–92, 2008.

[61] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 2002.

[62] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[63] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI ’91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: http://doi.acm.org/10.1145/113445.113455.

[64] Mark Weiser. Program slicing. In *ICSE ’81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.

[65] Mark Weiser and Jim Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp. ISBN 0-89391-388-X. URL http://portal.acm.org/citation.cfm?id=21842.28894.

[66] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Ann Arbor, MI, USA, 1979.

Index of Terms

Appendix A

Boogie Example

In this Appendix it will be listed the Listing 7.2 translated to Boogie.

```
1
2
3  procedure test.tester$System.Int32(this: ref where $IsNotNull(this, test) && $Heap[this, $
       allocated], x$in: int where InRange(x$in, System.Int32));
4    // user-declared preconditions
5    requires x$in >= 0;
6    // target object is peer consistent
7    requires (forall $pc: ref :: { $typeof($pc) } { $Heap[$pc, $localinv] } { $Heap[$pc, $inv] } {
         $Heap[$pc, $ownerFrame] } { $Heap[$pc, $ownerRef] } $pc != null && $Heap[$pc, $allocated]
         && $Heap[$pc, $ownerRef] == $Heap[this, $ownerRef] && $Heap[$pc, $ownerFrame] == $Heap[this
         , $ownerFrame] ==> $Heap[$pc, $inv] == $typeof($pc) && $Heap[$pc, $localinv] == $typeof($pc
         ));
8    // target object is peer consistent (owner must not be valid)
9    requires $Heap[this, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[this, $ownerRef], $
         inv] <: $Heap[this, $ownerFrame]) || $Heap[$Heap[this, $ownerRef], $localinv] == $BaseClass
         ($Heap[this, $ownerFrame]);
10   free requires $BeingConstructed == null;
11   free requires $PurityAxiomsCanBeAssumed;
12   modifies $Heap, $ActivityIndicator;
13   // newly allocated objects are fully valid
14   free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } $o != null && !old
         ($Heap)[$o, $allocated] && $Heap[$o, $allocated] ==> $Heap[$o, $inv] == $typeof($o) && $
         Heap[$o, $localinv] == $typeof($o));
15   // first consistent owner unchanged if its exposeVersion is
16   free ensures (forall $o: ref :: { $Heap[$o, $FirstConsistentOwner] } old($Heap)[old($Heap)[$o,
         $FirstConsistentOwner], $exposeVersion] == $Heap[old($Heap)[$o, $FirstConsistentOwner], $
         exposeVersion] ==> old($Heap)[$o, $FirstConsistentOwner] == $Heap[$o, $FirstConsistentOwner
         ]);
17   // frame condition
18   free ensures (forall<alpha> $o: ref, $f: Field alpha :: { $Heap[$o, $f] } $o != null &&
         IncludeInMainFrameCondition($f) && old($Heap)[$o, $allocated] && (old($Heap)[$o, $
         ownerFrame] == $PeerGroupPlaceholder || !(old($Heap)[old($Heap)[$o, $ownerRef], $inv] <:
         old($Heap)[$o, $ownerFrame]) || old($Heap)[old($Heap)[$o, $ownerRef], $localinv] == $
         BaseClass(old($Heap)[$o, $ownerFrame])) && ($o != old(this) || !($typeof(old(this)) <:
         DeclType($f)) || !$IncludedInModifiesStar($f)) && true ==> old($Heap)[$o, $f] == $Heap[$o,
         $f]);
19   free ensures $HeapSucc(old($Heap), $Heap);
20   // inv/localinv change only in blocks
21   free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } old($Heap)[$o, $
         allocated] ==> old($Heap)[$o, $inv] == $Heap[$o, $inv] && old($Heap)[$o, $localinv] == $
         Heap[$o, $localinv]);
22   free ensures (forall $o: ref :: { $Heap[$o, $allocated] } old($Heap)[$o, $allocated] ==> $Heap[
         $o, $allocated]) && (forall $ot: ref :: { $Heap[$ot, $ownerFrame] } { $Heap[$ot, $ownerRef]
          } old($Heap)[$ot, $allocated] && old($Heap)[$ot, $ownerFrame] != $PeerGroupPlaceholder ==>
          $Heap[$ot, $ownerRef] == old($Heap)[$ot, $ownerRef] && $Heap[$ot, $ownerFrame] == old($
         Heap)[$ot, $ownerFrame]) && old($Heap)[$BeingConstructed, $NonNullFieldsAreInitialized] ==
         $Heap[$BeingConstructed, $NonNullFieldsAreInitialized];
23   free ensures (forall $o: ref :: { $Heap[$o, $sharingMode] } old($Heap[$o, $sharingMode]) == $
         Heap[$o, $sharingMode]);
24
25
```

```
26
27 implementation test.tester$System.Int32(this: ref, x$in: int)
28 {
29   var x: int where InRange(x, System.Int32);
30   var stack0i: int;
31   var stack0o: ref;
32   var stack1o: ref;
33
34   entry:
35     x := x$in;
36     goto block3281;
37
38   block3281:
39     goto block3298;
40
41   block3298:
42     // ------ nop ------ test3.ssc(7,3)
43     // ------ load constant 100 ------ test3.ssc(9,3)
44     stack0i := 100;
45     // ------ binary operator ------ test3.ssc(9,3)
46     stack0i := x + stack0i;
47     // ------ copy ------ test3.ssc(9,3)
48     x := stack0i;
49     // ------ load constant 200 ------ test3.ssc(10,3)
50     stack0i := 200;
51     // ------ binary operator ------ test3.ssc(10,3)
52     stack0i := x - stack0i;
53     // ------ copy ------ test3.ssc(10,3)
54     x := stack0i;
55     // ------ load constant 200 ------ test3.ssc(11,3)
56     stack0i := 200;
57     // ------ binary operator ------ test3.ssc(11,3)
58     stack0i := x + stack0i;
59     // ------ copy ------ test3.ssc(11,3)
60     x := stack0i;
61     // ------ serialized AssertStatement ------ test3.ssc(12,3)
62     assert x >= 100;
63     goto block3332;
64
65   block3332:
66     // ------ nop ------ test3.ssc(12,10)
67     // ------ return ------ test3.ssc(12,10)
68     return;
69 }
```

Listing A.1: Boogie example

Appendix B

Context-free Grammar

In this Appendix it will be listed the Context-free Grammar (in extended BNF notation) for Boogie. That grammar was the basis for the development (automatic generation) of our tool GamaBoogie.

Body	::=	{ LocalVarDecl* StmtList }
LocalVarDecl	::=	**var** Attribute* $IdsTypeWhere^{,+}$;
StmtList	::=	LStmt* $LEmpty^{?}$
LStmt	::=	Stmt | Id : LStmt
LEmpty	::=	Id : LEmpty?
Stmt	::=	**assert** Attribute* Expr ;
	|	assume **Attribute*** Expr ;
	|	**havoc** $Id^{,+}$;
	|	$Lhs^{,+}$:== $Expr^{,+}$;
	|	**call** $CallLhs^{?}$ Id ($Expr^{,*}$) ;
	|	**call forall** Id ($WildcardExpr^{,*}$) ;
	|	IfStmt
	|	**while** (WildcardExpr) LoopInv* BlockStmt
	|	**break** $Id^{?}$;
	|	**return** ;
	|	**goto** $Id^{,+}$;
Lhs	::=	Id MapSelect*
MapSelect	::=	[$Expr^{,+}$]
CallLhs	::=	$Id^{,+}$:=
WildcardExpr	::=	Expr | *
BlockStmt	::=	{ StmtList }
IfStmt	::=	**if** (WildcardExpr) BlockStmt $Else^{?}$
Else	::=	**else** BlockStmt | **else** IfStmt
LoopInv	::=	**free**$^{?}$ **invariant** Attribute* Expr ;

Figure B.1: Boogie statement grammar.