Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

José Pinheiro
Departamento de Informática - Universidade do Minho

Exploring Frama-C to improve Assertion-based Slicing

Master dissertation

Supervised by: Pedro Rangel Henriques

Daniela da Cruz

Braga, May 17, 2015

ABSTRACT

This document describes a master thesis in Informatics, in the areas of Program Slicing and Formal Verification of Programs, and the synergies between both.
The project, entitled as *Exploring Frama-C to improve Assertion-based Slicing*, is aimed at the exploration of a well-known program analysis and verification tool, Frama-C, to understand how it can be effective to implement the semantic slicing approach, called Assertion-based Slicing, used to analyze and reason about programs developed with Contracts.
As a proof of concept, a tool will be developed and tested specific case studies.

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1 INTRODUCTION

Program Slicing has been around for a long time, and was first defined by Mark Weiser, Weiser (1981). According to him, "Program slicing is a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior".

Since then several different kind of slicing techniques have been developed from several different types of static slicing (slicing applied only to the source code with no other transformation) to different types of dynamic slicing (which uses a particular execution of the program). Program slicing can be applied to debugging programs, program analysis, flow control, optimization and software maintenance. Although slicing techniques have been around since the eighties, they haven't received a wide acceptance by the software industries, mainly to the difficulty of implementation. This thesis will explore a new form of slicing using programs with Design by contract approach, called "Assertion-based-slicing".

Design by Contract, was first introduced by Bertrand Meyer, Meyer (1992), when he designed the Eiffel programming language. Design by contract is an approach to software development, that sates that software developers should define a precise formal and verifiable specification for software components, with preconditions, postconditions and invariants. These specifications are called "contracts". Design by contract has its it roots in Hoare Logic, formal verification and specification. For example: A function *f()* is only executed if it is called in a state which satisfies its precondition specification and when it terminates it must guarantee that the postcondition specification is verified. Several more different programming languages have also native support Design by Contract but others like C programming language have third party support and currently the verification tool is Frama-C.

Frama-C is a set of program analyzers for the C programming Language, and was developed by Commissariat à l'Énergie Atomique et aux Énergies Alternatives and INRIA[1]. Frama-C does static analysis of C programs and can be used for formal verification (using ACSL[2]), value analysis, slicing

1 Institut national de recherche en informatique et en automatique, which translates to English as Institute for Research in Computer Science and Automation

2 ANSI/ISO C Specification Language

and others. Also, Frama-C is easily extended in the form of plugins due to its modular architecture design.

This project is entitled ”Exploring Frama-C to improve Assertion-based Slicing”. As stated before, “Assertion-based slicing” is a technique that applies slicing to programs developed with contracts; Frama-C will be used to develop a plugin to apply this type of slicing to C programs.

In the next chapter, an overview of Slicing and its different techniques will be given. The techniques include static slicing, dynamic slicing, quasi-static slicing, conditioned slicing and mainly assertion-based slicing. After that, Frama-C and the plugins will be analyzed. In this chapter, it is explained the plugins that use slicing (value analysis plugin, slicing plugin, impact analysis plugin, spare code plugin and PDG plugin) report the results with contracts, and check if they already use any kind of semantic slicing. Frama-C slicing plugins strengths and weakness are also explored. In the end, the developed plugin will be explained, the different development approaches are discussed, and some examples will be shown.

1.1 OBJECTIVES

The aim of this master thesis is split in two main categories. A theoretical one, and another more practical for proof of concepts. This objectives of this master thesis are the following:

- To explore Frama-c slicing plug-in, to know how the plug-in works and what methods of slicing it applies;
- To develop a Frama-c plug-in in OCaml[3], in order to apply assertion-based slicing to C programs;
- To use the developed plug-in in large scale C tests (a unix kernal library for example), to test assertion-based slicing to discover its strengths and limitations.

The main objective of this thesis is to improve assertion based slicing using Frama-C and test if it can be applied to large C programs.

1.2 CONTRIBUTIONS

This master work contributed with a new plugin for Frama-C:

- It allows to apply Assertion-based slicing;
- It corroborates the easiness of extending Frama-C.

3 Ocaml home page: http://caml.inria.fr/

1.3 DOCUMENT STRUCTURE

This dissertation is divided in five chapters:

- **Introduction** - Introduction to the dissertation;
- **Slicing** - Overview of slicing techniques and assertion-based slicing;
- **Frama-C** - Overview of Frama-C and it's plugins;
- **GamaSlicer** - Description of the developed plugin;
- **Conclusion** - Conclusion and reflections of this dissertation.

2

SLICING

Program slicing, Weiser (1981), is the computation of a set of program statements (program slice) using a statement of the same program(slicing criteria). It is used for debugging, software testing, software metrics, software maintenance, program comprehension, etc .

Slicing has been around since 1979, but until now it is still not a widely known field in Computer Science. Its due to the several different techniques that exists, with most being difficult to implement which its correlated to their difficulty to leave the academic world .

Those several techniques include static slicing (the original), dynamic slicing, quasi-static slicing, conditioned slicing, assertion based slicing, and so on. For more detailed and thorough explanation of the different kind of slicing methods, it is recommend to read: Silva (2012).

Static slicing, Weiser (1981), consists of choosing a statement as slicing criteria and applying forward, Bergeretti and Carré (1985), or backward, Tip (1995), slicing. Static means that only statically available information is used for computing the slices, this is, all possible executions of the program are taken in account; no specific input is considered.

Dynamic slicing, Korel and Laski (1988), builds a slice with respect to only one execution of the program corresponding just to one given input. Quasi-static slicing, Venkatesh (1991), is a slicing method that combines static slicing and dynamic slicing, it slices a program with an specific input to a certain variable.

Conditioned slicing, Jiang et al. (1991), creates program slices that consist of a subset of program statements which preserve the behavior of the original program with respect to a slicing criterion for any set of program executions. The set of initial states is specified with a first order logic formula on input.

Assertion-based slicing, Barros et al. (2012), is a term that encompass: postcondition-based, precondition-based and specification based forms of slicing. Postcondition-based slicing uses as slicing criterion the postcondition of the program. Precondition-based slicing is the same as postcondition-based slicing but uses the precondition as criterion. Specification-based slicing is a combination of both, using the postcondition and precondition as slicing criterion.

Program slicing can also be used to enhance verification techniques, that combine static and dynamic analysis, as shown in Chebaro et al. (2012). In this technique, a program is first analyzed by value analysis (using Frama-c value analysis plugin [1]) and labeled with alarms of possible runtime errors. Then using PathCrawler (a structural test generation tool [2]) it generates tests to prove the validity of the alarms. This was the first Sante Method (Static ANalysis and TEsting), Chebaro et al. (2010). The problem with this approach is that the number of paths can be exponential in the program size. By using program slicing to reduce the number of paths, depending on the options of the tool it can have gains between 24 % to 97 % in program reduction, depending on the complexity of the program and the generated alarms from Frama-c value analysis plugin. Using program slicing the Sante method was vastly improved.

A new dynamic program slicing algorithm based on an abstract machine was presented by Hua-Xiao et al. (2013). The main difference with this approach is that the execution path of program is judged by an abstract machine and the dynamic program slicing is based on input and makes use of control dependency and data dependency information. The abstract state machine is defined by a quadruple: (Stack, Econ, Stmcon, Denv) . Stack is where it stores the intermediate results of calculation and the value of a variable in an assignment expression. Econ is where it is stored the pending expression. Stmcon is where pending statements are stored. Denv is mapping of variable name to value. The initial state, final state and the state transition rules of the abstract machine are detailed in Hua-Xiao et al. (2013). The abstract machine confirms the programming execute path, by marking each statement with true or false. If the statement has been executed the mark would be true, otherwise false. After the execution of the abstract machine the statement which are true will be exported and construct the execution path of program. The principle of assigning values to executive marks is explained in Hua-Xiao et al. (2013). The authors state that this approach has higher accuracy and lower time and space complexity. The main problem with this approach is the lack of pointer support, which they state that in the future will be solved.

When slicing Objected Oriented programs, there have been several approaches. The challenge with object oriented programs is because of features such as class, objects, inheritance and polymorphism, Jain and Garg (2013). All these features strengthen the expression of ideas in object oriented programs but at the same time pose a challenge to slicing, due to this features cannot be represented in a normal System Dependency Graph. Although there are several graph implementations of control flow and data dependence from object oriented programs (including Class Dependence Graph, Object Oriented Program Dependence Graph, Class Control Flow Graph, etc) generating a graph from an object oriented program is itself very difficult and generating slices from dependencies very complicated. The need for a graph is normally due to the need of a graphic representation of the program. Since a graphic representation of program is not always needed, another approach was proposed to calculate slices based on definition-use (d-u) chains, Jain and Garg (2013). A d-u chain is a sequence of adjacent d-u pairs, and a d-u pair is a definition of a variable and the uses of that variable. Using d-u chains is much easier to represent the program and the space complexity is lesser, which leads to faster slice results.

1 Frama-c value analysis home page: `http://frama-c.com/value.html`
2 Frama-c pathcrawler home page: `http://frama-c.com/pathcrawler.html/`

There exist some tools that can apply slicing and have significant performance. One of those is Frama-c[3]. Frama-c is a suite of tools dedicated to the analysis of source code written in C. Frama-c applies slicing to source code using a slicing plug-in. This plug-in uses the results of the value analysis plug-in and of the function dependencies computation. It supports slicing criteria for code observation and slicing criteria for proving properties. Although being an open source system, it is not crystal clear which slicing methods Frama-c actually applies in each of the above mentioned variants.

2.1 ASSERTION-BASED-SLICING

A program slice takes in account a slicing criteria (ρ, v) where ρ is a program statement and v is a subset of the program's variables. This means a slice consist of statements which affect the values of variables at the slicing statement with a criteria. Although it preserves the behavior of the original program with respect to (ρ, v), normally contains non-essential statements if the slicing criteria is strengthen with more context of applying the slicing technique.

One way to have more context is to re-use assertions from existing software satisfying a specification (like ACSL), and to apply a slicing criteria that uses all variables from ρ and that v contains all variables from the specification. To do this we can't only use the syntactic information but also use semantic information and use a relation between specification and the semantic information of the program, this technique is called assertion-based-slicing Barros et al. (2012). It is here that this slicing technique diverges from normal slicing techniques, because normally slicing techniques use only syntactic information and are called syntactic slicing, and techniques using semantic information are called semantic slicing(but also use syntactic information).

For example, suppose we wish to calculate the slice of a program based on its specification, where it consists of (*P*,*Q*) where *P* is the precondition and *Q* is the postcondition and that exists a stronger precondition *P'* than *P*, or else the desired postcondition is *Q'* is weaker than the specified *Q*. From a software engineering perspective it would be desirable to eliminate code that may be superfluous with respect to the specification (*P'*,*Q'*). We will now "assertion-based-slicing" to refer to slicing techniques that use axiomatic semantics of programs taking as criteria assertions (postconditions and preconditions) annotated in programs. This includes: precondition-based slicing, postconditon-based slicing and specification-based slicing. Assertion-based slicing is more powerful and flexible than syntactic slicing, since the criteria can be as expressive as any set of first-order formulas on the initial and final states of the program.

2.1.1 *Postcondition-based Slicing*

From Chung et al. (2001): A postcondition based slice with respect to a postcondition *Q* is a backward static slice that consists of a subset of the statements and control predicates of a program that might

3 Frama-c home page: `http://Frama-c.com/`

affect the postcondition when the program is executed. The idea of slicing programs based on their specifications was introduced by Comuzzi et al Comuzzi and Hart (1996), with the notion of predicate slice (*p*-slices). To understand the idea of *p*-slices, Cruz (2011), 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.

2.1.2 *Preconditon-based Slicing*

Chung and colleagues Chung et al. (2001), later introduced precondition-based slicing as the dual notion of postcondition-based slicing. A precondition based slice with respect to a precondition *P* is a forward static slice that consists of a subset of the statements and control predicates of a program that might be executed and change the program state when the program start execution in a state satisfying *P* . The idea is still to remove statements who do not affect the specification of the final state of program, the difference is that the set of executions of the program is now restricted to the first-order condition on the initial state. If a statement does not violate any property of the final state it can be removed, this is equal to saying that the strongest postcondition of the program is not weakened in the computed slice.

2.1.3 *Specification-based Slicing*

A specification base slice can be calculated when both precondition *P* and postcondition *Q* are given in a specification of a program. The set of relevant slices is restricted to those for which *Q* holds upon termination when the program is executed in a sate that satisfies *P*. Programs resulting from these slices and which are still correct to (*P*,*Q*) are said to be specification-based slices. This method is also proposed by Chung and colleagues , Chung et al. (2001), and relies on a theorem proved by the authors which states that the composition in any order of postconditon-based slicing (in regard to *Q*) and precondition-based slicing (in regard to *P)* produces a specification-based slice in regard to (*P*,*Q*).

3

FRAMA-C

Frama-C is a suite of tools dedicated to the analysis of source code written in C, Cuoq et al. (2012). With the help of Frama-C the user can observe sets of possible values for the variables of the program in execution, slice programs into simplified ones, navigate the dataflow of a program and, by using ACSL[1] annotations, use Frama-C to prove formal properties. Frama-C is coded in OCaml and it is organized with a plugin architecture (like Eclipse and Gimp Cuoq et al. (2012)). A common kernel centralizes information and analyses the code. The plugins interact with each other with the help of an interface defined by the kernel: this makes Frama-C easy to extend and any plugin can use the results of other as input.

3.1 CIL: C INTERMEDIATE LANGUAGE

CIL is important in this thesis because Frama-C relies on CIL for parsing and abstract syntax tree.

As the title states, CIL, stands for C Intermediate Language, Necula et al. (2002). An intermediate language is the language of an abstract machine designed to aid in understanding and analyzing programs. CIL, is a high level representation of the C language, that permits easy analysis and source code transformation of C programs. The C programming language is well known for its flexibility when dealing with low-level constructs, but also known for its difficulty to understand and analyze. CIL was developed to tackle that difficulty but still represent programs in a form that resembles the original code.

CIL features a reduce number of syntactic and conceptual forms. For example all syntactic sugar is eliminated, all functions are given explicit return statements and all loops variants are reduced to one form. CIL also deals with *lvalues*. Lvalue is an expression referring to a region of computer memory. Only an lvalue can appear on the left side of an assignment. In CIL, an lvalue is a pair of a base and an offset. The base address can be a starting address for a variable or a pointer expression. An offset can be empty or can be offset in a variable (or memory region) and which is denoted by the base that consists of a sequence of field or index designators.

CIL syntax is divided in three basic concepts:

1 ANSI/ISO C Specification Language

- Expressions, represent functional computation without side efects or control flow;
- Instructions, express side effects, but have no control flow;
- Statements, capture control flow.

CIL also provides control flow graph, with every statement annotated with successor and predecessor control flow information, so with CIL from a individual statement one can discover all other statements that can be reachable. The successors are the statements that will be executed sequentially after the original statement, the predecessor are the statements that were executed before the original statement.

CIL language was extended by Frama-C to support ACSL [2] with logic constructs in the CIL abstract syntax tree, which can be used to express preconditions and postconditions.

3.2 PLUGINS

Since the objective of this thesis is to explore Frama-C to improve assertion based slicing, the main focus of this chapter will be on Frama-C and its different plugins that are focused or use slicing techniques.

3.2.1 *Value Analysis Plugin*

The Value Analysis Plugin automatically computes variation domains for the variables of programs. It can show sets of inferred possible values for variables and can be used to infer the absence of run-time errors. It is used by most Frama-C plugins (include the slicing ones) this being the main reason it is displayed here. An example of it is use is shown bellow with the input being in Listing 3.1:

```c
int y, z=1;

int f(int x) {
  int y = x+1;
  return y;
}
void main (void) {
  for (y=0; y<20; y++) z = f(y);
}
```

Listing 3.1: val1.c

Calling Frama-C in batch mode with the *-val* option (Value Analysis Plugin) produces the output shown in Listing 3.2:

2 ANSI/ISO C Specification Language

```
$ frama-c -val -slevel 2 val1.c
(...)
[value] ====== VALUES COMPUTED ======
[value] Values at end of function f:
  y in [1..20]
[value] Values at end of function main:
  y in {20}
  z in [2..20]
```

Listing 3.2: Result from val1.c

3.2.2 *Slicing Plugin*

The slicing plugin given an input program produces an output which is made of a subset of statements of the analyzed code. This subset is produced though a slicing criterion, specificed by the user. The output is supposed to be compilable code and have the same behavior as the analyzed program from the point of view of the provided slicing criterion. Frama-C applies slicing to source code using a slicing plug-in. This plug-in uses the results of the value analysis plug-in and of the function dependencies computation. It supports slicing criteria for code observation and slicing criteria for proving properties. There are several slicing criterion including the use of ACSL expressions with slice pragmas. Although being an open source system, it is not crystal clear which slicing methods Frama-C actually applies in each of the above mentioned variants due to the lack of information and there is no paper describing precisely how the slicing module works, but it seems to be a combination of forward and backward slicing.

3.2.3 *Spare Code Plugin*

The goal of Spare Code Plugin is to remove unnecessary code in a program, where the output is guaranteed to be compilable code. The Spare code plugin is almost equal to the slicing plugin but always use as entry point the main function and the output code is from the point of view of values assigned to the main function. It can also use ACSL expressions in slice pragmas, but as the restriction of only analyzing the annotations found inside of a body of a function. It also depends on the Value analysis Plugin.

3.2.4 *Impact Analysis Plugin*

The Impact Analysis Plugin allows the automatic computation of the set of statements impacted by the side effects of a statement of a C program. Statements not appearing in this set are guaranteed not to be impacted by the selected statement. It relys on Frama-C graphic user interface to select the statement that will impact the rest of the program, you can also use it in batch mode but we have to write impact pragmas on the statements. It seems to do forward slicing, but like the slicing plugin there is no information to confirm that.

3.2.5 *PDG Plugin*

The PDG (Program Dependency Graph), as the name implies, given as input a program and an entry function, creates a program dependency graph in the form of .dot that latter can be converted to pdf. It is written in a special notation that is:

- The color of the edge represents the data dependencies, blue for yes, black otherwise.
- The shape of the arrow represent control dependencies, circle for yes, normal arrow otherwise.
- The lines represent address dependencies, dotted for yes, plain otherwise.

3.3 TESTING THE PLUGINS

In this section we will test Frama-C and its several plugins that are focused on slicing or use slicing techniques, such as:

- Slicing plugin;
- Spare Code plugin;
- Impact Analysis plugin;
- PDG plugin.

To test the plugin's we will use two simple programs with ACSL notations.

The first is a very simple function called `square.c` (see Listing 3.3) that squares a value and has an ACSL notation that guarantees that the result of the function is equal or greater than one hundred:

```
/*@ ensures \result >= 100;
*/
int f(){

  int x=1;

  x = x*x;
  x = x+100;
  x = x+50;
  return x;
}
```

Listing 3.3: square.c

The second is called `condi.c` (shown in Listing 3.4) and it is a simple program with a `main()` function that calls a function with an *if then else*; and the precondition requires that the function `g()` is given as input a value greater than ten and postcondition assures the result to be equal or greater than zero:

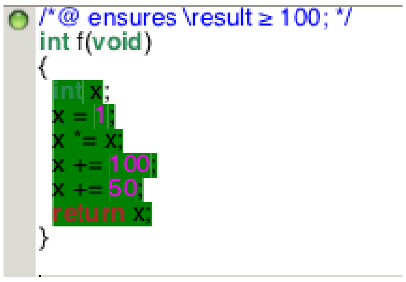
```
/*@ requires y > 10;
  @ ensures \result >= 0;
*/

int g(int y){
  int x=0;

  if(y>0){
    x=100;
    x=x+50;
    x=x-100;
  }else{
    x = x - 150;
    x=x-100;
    x=x+100;
  }
  return x;
}

int main(){
  int a = g(11);
  return a;
}
}
```

Listing 3.4: condi.c

3.3.1 *Slicing Plugin*

This is the main plugin to do slicing in Frama-C. We will try to discover what kind of slicing it does, its strengths and its limitations. We always use the same slicing criteria, that being slice-return, which slices the result value of a function.

We will first test the plugin with `square.c` (see Listing 3.3). Using `f()` as entry function, and after running the value analysis plugin(its required to do so) the result is shown in Figure 1 :



```
/*@ ensures \result ≥ 100; */
int f(void)
{
  int x;
  x = 1;
  x *= x;
  x += 100;
  x += 50;
  return x;
}
```

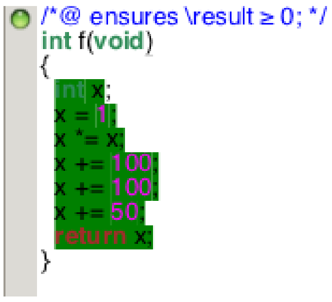
Figure 1: Slice result to square with f()

The result is the expected one that being assuming that Frama-C uses the traditional syntactic slicing, but it doesn't give us any new information about its slicing algorithm. If we add a replicated statement,`x=x+100` to the function `f()` and change the postcondition to be greater or equal to zero (see Listing 3.5):

```
/*@ ensures \result >= 0;
*/

int f(){

  int x=1;
    x = x*x;
  x = x+100;
  x = x+100;
  x = x+50;
  return x;
}
```

Listing 3.5: Modified square

When running on the modified square 3.5 on the slicing plugin the result is:



```
/*@ ensures \result ≥ 0; */
int f(void)
{
  int x;
  x = 1;
  x *= x;
  x += 100;
  x += 100;
  x += 50;
  return x;
}
```

Figure 2: Slice result to modified square with f()

Still, no surprise it didn't slice the statement `x=x+100;`.

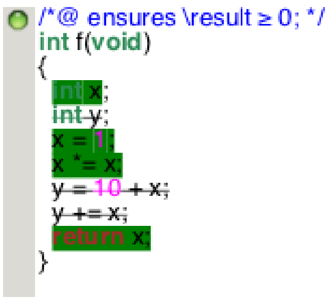
One final modification to 3.3:

```
/*@ ensures \result >= 0;
```

```
*/

int f(){
  int x=1;
  x = x*x;
  int y = 10+x;
  y += x;
  return x;
}
```

Listing 3.6: Alternative modified square

In this modified square version (Listing 3.6), we add two new statements, that will not impact the final result, but will use the variable x. Now the result from running the slicing plugin:



```
/*@ ensures \result ≥ 0; */
int f(void)
{
  int x;
  ~~int y;~~
  x = 1;
  x *= x;
  ~~y = 10 + x;~~
  ~~y += x;~~
  return x;
}
```

Figure 3: Slice result to alternative modified square with f()

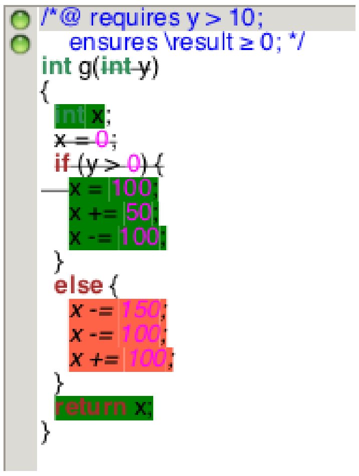
As we can see in Figure 3 it slices all statements with the variable y. This proves it does syntactic slicing, how it does is unclear due to lack of documentation of this plugin. The expected result if it did assertion based slicing would be:

```
/*@ ensures \result >= 100;
*/
int f(){

  int x=1;
  x = x*x;
  x = x+100;
  return x;
}
```

Listing 3.7: Assertion based slicing result label

It would slice the statement `x= x+50;` because the statement `x=x+100;` guarantees the postcondition is respected and all new modifications (like `x= x+50;`) are unnecessary to respect the postcondition (although they also respect the postcondtion), and can be sliced.

Now, using `condi.c` (see Listing 3.4), using the `g()` function with slice return:

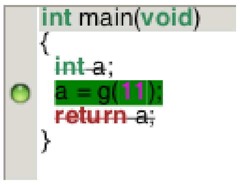


```
/*@ requires y > 10;
    ensures \result ≥ 0; */
int g(int y)
{
  int x;
  x = 0;
  if (y > 0) {
    x = 100;
    x += 50;
    x -= 100;
  }
  else {
    x -= 150;
    x -= 100;
    x += 100;
  }
  return x;
}
```

Figure 4: Slice result to condi g() function

It sliced the `if(y>0)` statement, so we can infer it uses the precondition to do the slicing. Why it sliced the statement `x=0;` and `x=100;` is unclear and goes against the definition of slicing, since those attributions are necessary to the the result of the function.

Also the `main()` slice:

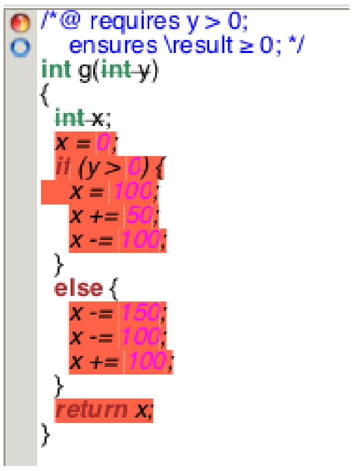


```
int main(void)
{
  int a;
  a = g(11);
  return a;
}
```

Figure 5: Slice result to condi main() function

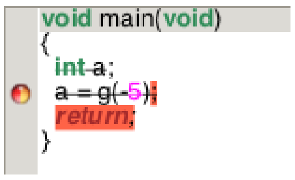
It sliced the declaration type variable a, which is very wrong, since C requires variables type to be declared. Also sliced the return statement which is also wrong with the definition of slicing. This code would not be compilable.

Lets, now change the input value of function `g()` to -5, the result from `main()` and `g()` is:



```
/*@ requires y > 0;
    ensures \result ≥ 0; */
int g(~~int y~~)
{
  ~~int x~~;
  x = 0;
  if (y > 0) {
    x = 100;
    x += 50;
    x -= 100;
  }
  else {
    x -= 150;
    x -= 100;
    x += 100;
  }
  return x;
}
```

Figure 6: Slice result to condi alternative g() function



```
void main(void)
{
  ~~int a~~;
  ~~a = g(-5)~~;
  return;
}
```

Figure 7: Slice result to condi alternative main() function

We can see that it sliced everything, the red code is from the value analysis and the slicing plugin never slices when the value analysis reports the code is dead(in red). From this we know it respected the precondition, but this was achieved with the value analysis plugin and not the slicing plugin.

This combination of both plugins seems to be the main strength of the slicing plugin since it uses pre and post conditions in the slice and at the same time its weakness, because it is dependent on the value analysis of the program and in reality the slicing plugin is oblivious to pre and post conditions and just relies on the results of the value analysis.

3.3.2 *Spare Code Plugin*

The result from running the Spare Code plugin on Listing 3.3 using as entry function, `f()`, is that nothing is sliced which is correct.

When running the Spare Code plugin on Listing 3.4, we have to use two different entry function, `g()` (show in Listing 3.8) and `main()` (show in 3.9). With function `g()`, the result is the following:

```
/* Generated by Frama-C */
int g(void)
{
  int x;
  x = 100;
  x += 50;
  x -= 100;
  return x;
}
```

Listing 3.8: Spare Code result for condi.c using g as entry function

We can see, that it removed the if control statement and the code inside the else statement. The difference from the slicing plugin is that it didn't remove the return statement, but removed any indication of variable y as input of the function and the attribution of 0 to the variable x.

Last, using the function `main()` as entry:

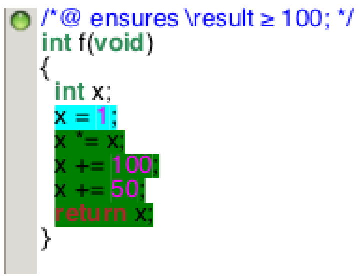
```
void main(void)
{
  return;
}
```

Listing 3.9: Spare Code result for condi.c using main as entry function

The result is strange. First it didn't propagate the slice to the function g and removed the declaration of the variable a and added a return statement.

3.3.3 *Impact Analysis Plugin*

When running the Impact Analysis Plugin on `Square` (see Listing 3.3) using the statement `x=1;` (see image 8):

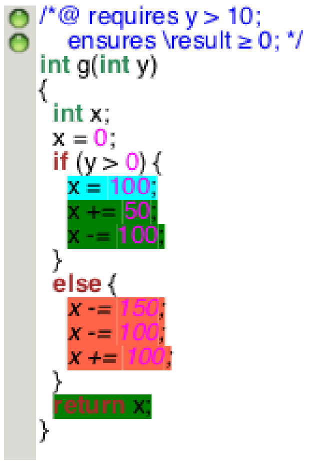


```
/*@ ensures \result ≥ 100; */
int f(void)
{
  int x;
  x = 1;
  x *= x;
  x += 100;
  x += 50;
  return x;
}
```

Figure 8: Impact Analysis on square

The result (listed in Figure 8) is the expected one, as all statements that are affected by `x=1`(in blue) are marked(in green).

Using the statement `x=100;`(in blue) in `condi` (Listing3.4):



```
/*@ requires y > 10;
    ensures \result ≥ 0; */
int g(int y)
{
  int x;
  x = 0;
  if (y > 0) {
    x = 100;
    x += 50;
    x -= 100;
  }
  else {
    x -= 150;
    x -= 100;
    x += 100;
  }
  return x;
}
```

Figure 9: Impact Analysis on condi

The only statements that are impacted by `x=100` are the ones inside the if block and the `return x` statement(in green), as show in Figure 9.

3.3.4 *PDG Plugin*

The resulting PDG [3] for `square.c` (show in Listing3.3) by using the `(`f()) function as entry point is:

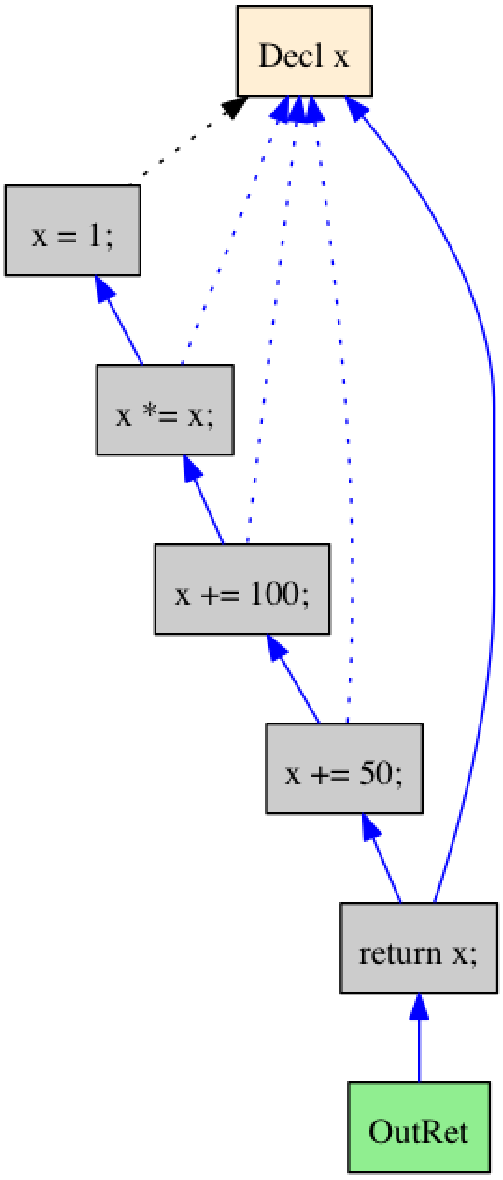


Figure 10: Square PDG for the f() function

All statements, except `x=1` are data dependent to `Decl x`, all are normal dependencies and all statements except ”return x” are address dependent to `Decl x`.

3 All graphs were generated from *.dot*.

The resulting PDG for `condi.c` (show in Listing 3.4) using the g function as entry point is showed in Figure 11:

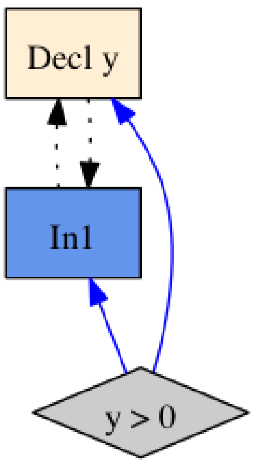
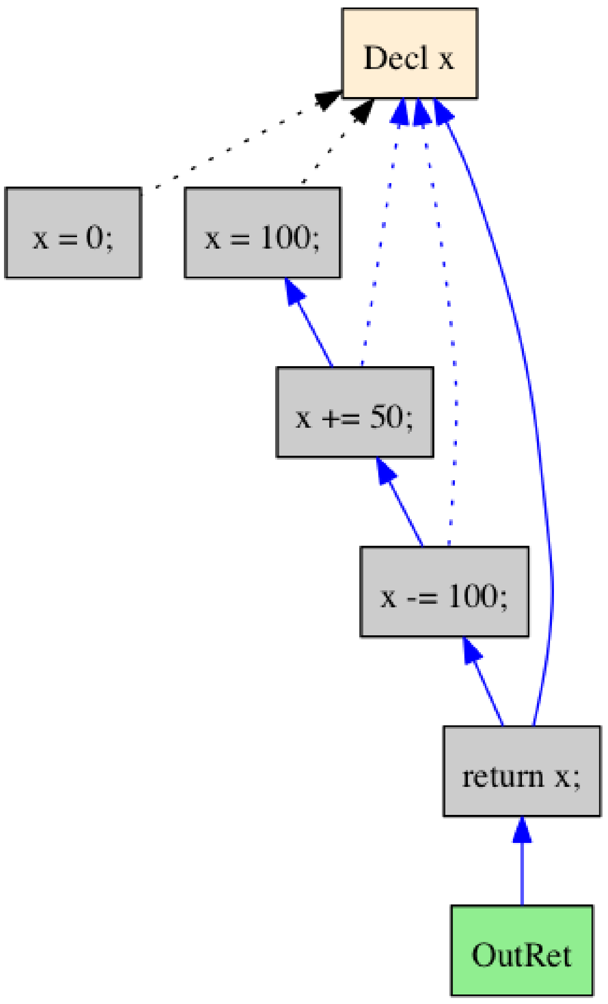



Figure 11: Condi PDG for the g() function

The statement `y>0`, in the control flow point, is data and address dependent to `Decl y` and `In1` which is the input from the function `g()`. `Decl y` and `In1` are address dependent to each other. All other statements are normal dependencies to `Decl x`. `return x;`, `x-=100;` and `x+=100;` are data dependencies to `Decl x`. Lastly all statements except `return x` address dependent to `Decl x`.

For last the resulting PDG for `condi.c` (show in Listing 3.4) by using the `main()` function as entry point is:

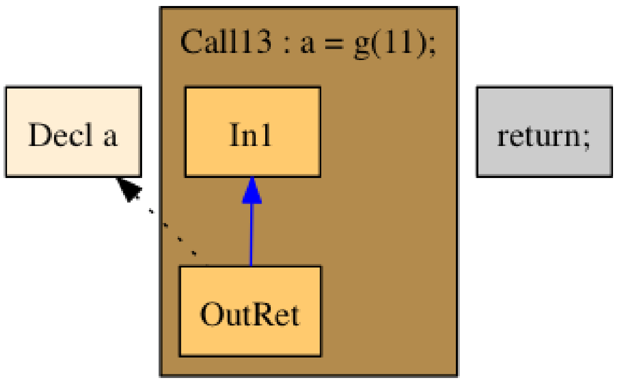


Figure 12: Condi PDG for the main() function

The output for the main faction is is data dependent to `In1` and is address dependent to `Decl a`.

4

GAMASLICER

The aim of this thesis is to develop a Frama-c plugin in OCaml to apply assertion-based slicing to C programs. To achieve this, it is needed: a parser, an Abstract Syntax Tree, a verification condition generator (that generates both weakest preconditions and strongest postconditions per statement), a slicegraph and a way to print the shortest path in the slicegraph. It was decided that the developed plugin would be called GamaSlicer.

In this Chapter, first it will be explained the different approaches to implementation, then what is a Frama-C plugin and its implementation, and finally results of the plugin.

4.1 APPROACHES

4.1.1 *Initial Approach*

After studying Frama-C and it's plugins, the initial proposal as shown in figure 13 was to reuse the Frama-C parser, it's Cil_AST [1] and WP Frama-C plugin to generate the weakest preconditions [2].

The input file would be parsed by Frama-C, converted into a Cil_Ast, and the WP plugin would generate the weakest preconditions required to apply Postcondition-based Slicing. After generating the proof obligations the WP plugin would send it to an SMT solver [3](also called prover) and the prover would report the proof obligations validity. The Cil_AST, would also be used to create a slicegraph (a control flow graph that can have additional edges connecting the statements).

Then the module Slicing from GamaSlicer would collect the validity of proof obligations, and depending on that validity add new edges to the slicegraph. A shortest path algorithm would be applied to the slicegraph, and the corresponding path would be printed by the printer, giving a sliced output file.

1 Cil stands for, C Intermediate Language, the main goal of Cil is to aid in program analysis and transformation.
2 To note that this was possible to do by using Frama-C API and modules.
3 Satisfiability Modulo Theories solver.

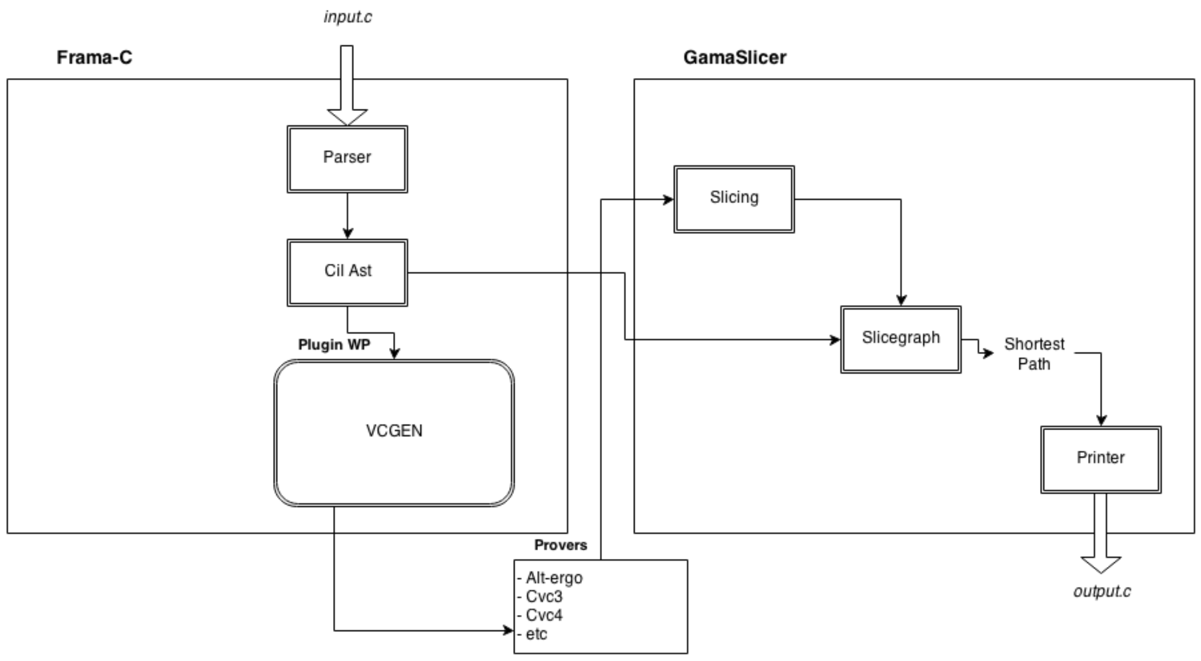


Figure 13: GamaSlicer initial approach

This approach failed mainly due to the lack of documentation and high dificulty in use of the API of the Weakest Precondition Plugin. Although it is fully functional to use WP in the command line, the documentation of the API is lacking and very confusing. To note, that only the API of this plugin is lacking, not the Frama-C API itself. This could be mainly due to WP plugin being new compared to other Frama-C plugins and Frama-C itself. Also, with this approach it would be required to develop a strongest poscondition Vcgen in the GamaSlicer plugin, due to the fact that WP plugin only calculates weakest preconditions proof obligations.

4.1.2 *Second Approach*

Due to the failure of the initial approach, a second approach was devised, as shown in figure 14. As stated early, the WP plugin is fully functional at the command line, and can be used to generate proof obligations of a program into a folder.

The goal was to take advantage of this, so this approach was nearly identical to the first one, but now the GamaSlicer plugin would invoke externally at the command line level the WP plugin with the input file. The results would be stored in a folder, and after the termination of the execution of the WP plugin , a GamaSlicer module would parse the results in the folder. After obtaining the results, it would work exactly as the first approach.

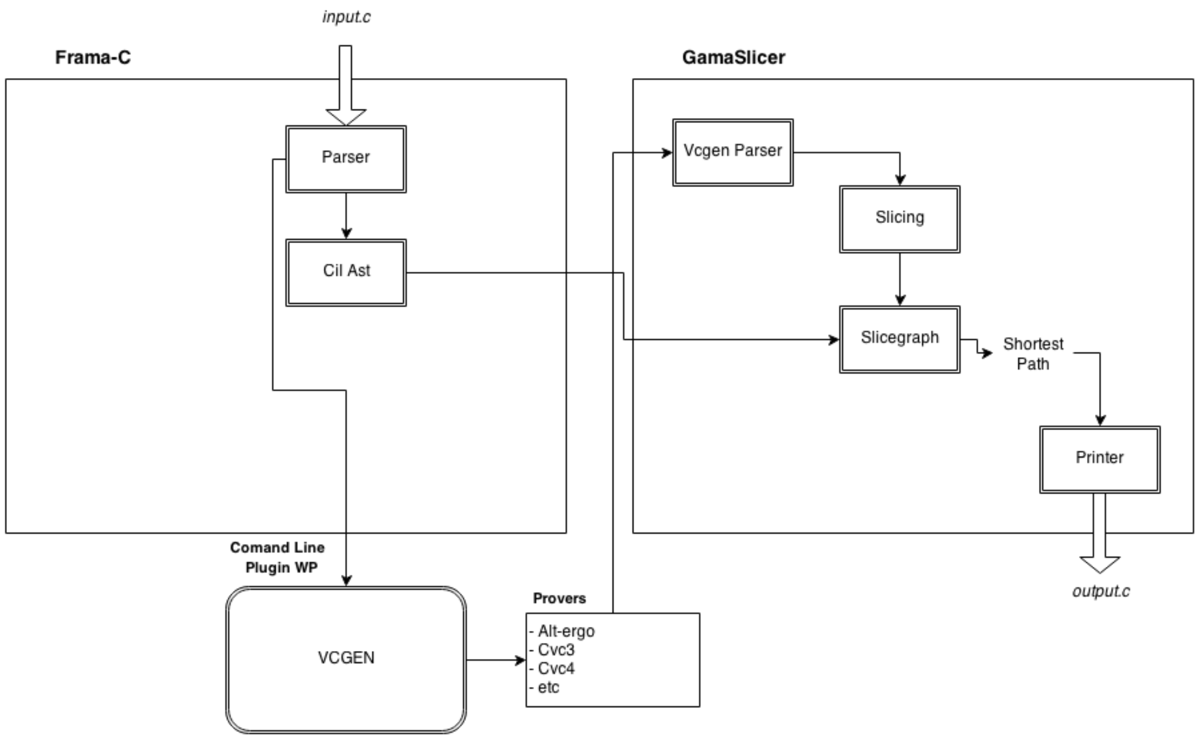


Figure 14: GamaSlicer second approach

This second approach also failed, due to the fact that WP plugin was not able to generate a proof obligation per statement and only the final proof obligation of the program [4]. As GamaSlicer requires weakest precondition calculus to apply assertion-based slicing. This approach also had the same drawback of the WP plugin not generating strongest postconditions, which are required to do precondition-based slicing and specification-based slicing.

4.1.3 *Final Approach*

With both the initial and second approach failing a new approach was needed, in figure 15 . As stated before the problem was always with the WP plugin and with the lack of way to generate strongest postconditions. It was decided that a new module in the GamaSlicer plugin would be implemented. This new module would receive Frama-C Cil_Ast and generate both weakest precondition and strongest postcondition proof obligation per statement and be called Vcgen.

After generating the proof obligations, their validity would be proven. To do this a new module using Why3 [5] API. First the formulas are converted to Why3 Terms [6] and then Why3 connects to the provers of choice to assert the validity of the proof obligation. With the validity of the proof

4 The proof obligation that proves or disproves the precondition or postcondition of the given program.
5 Why3 is a platform for deductive program verification. Why3 homepage: http://why3.lri.fr/
6 Why3 Term is a Why3 library available trough Why3 API that is used to create and build logic terms which then can be used to build logic formulas.

obligations found and reported to the GamaSlicer Slicing module this approach is equal to the two initial approaches.

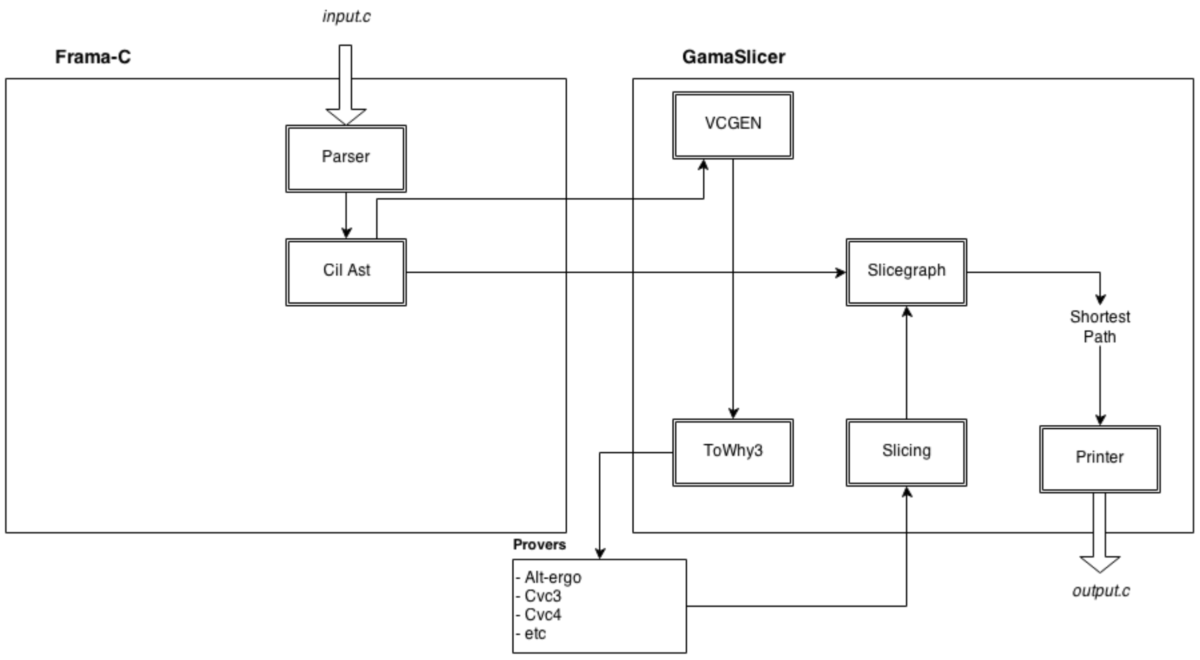


Figure 15: GamaSlicer final approach

This was the approach that was implemented with success, although with some drawbacks. Reusing Frama-C modules and plugins, had the advantages of time and performance. Also creating a Vcgen for both WP and SP calculus with the imposed time constraints, some cuts had to be done, implementing only simple C types.

4.2 A FRAMA-C PLUGIN

As stated before GamaSlicer is a Frama-C Plugin. One can use Frama-C with several entry points. By a plugin registering in entry point, Frama-C will recognize and execute that plugin when called to. The several different entry points are shown in Figure 16:

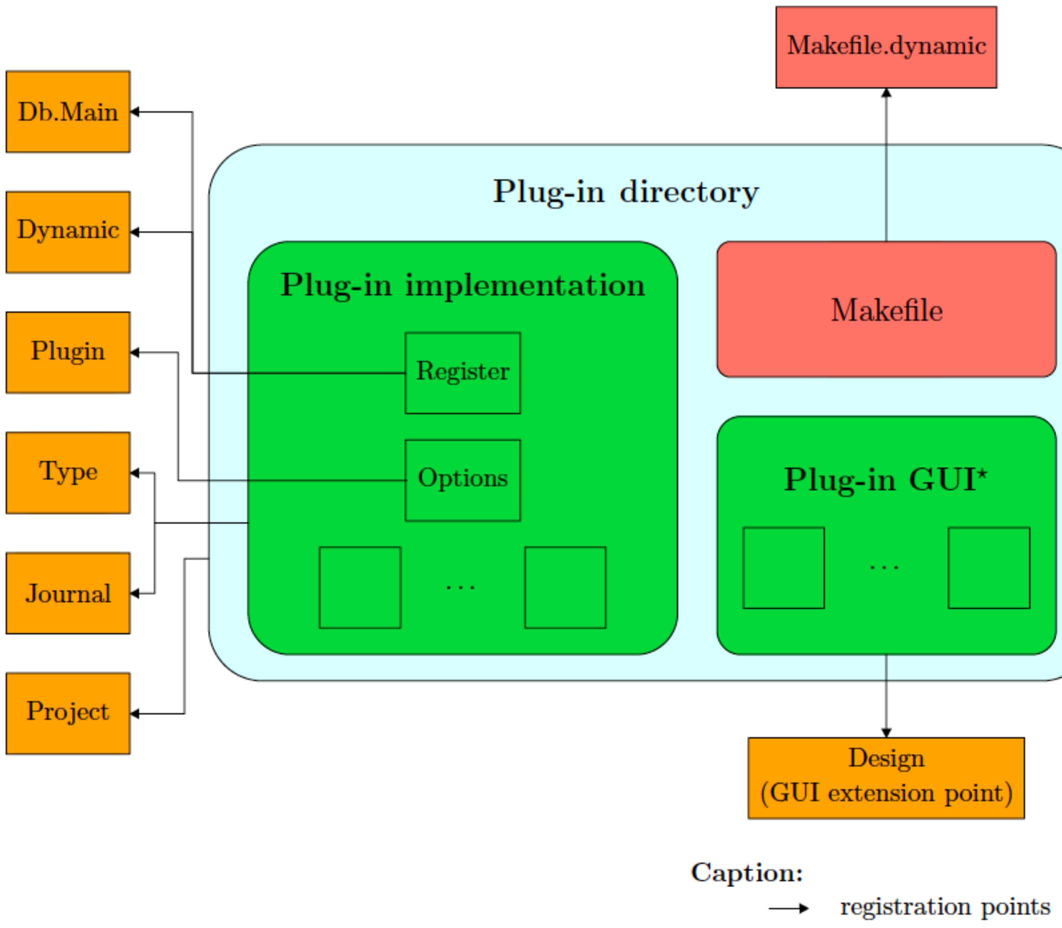


Figure 16: Plugin intregation overview

The two main registration points reproduce the two ways that a Frama-C plugin can be used:

- Use a simple script that extends Frama-c entry point by using **Db.Main.extend** in a Ocaml script;
- Build a plugin by using plugin Modules as script or by using **Makefile.dynamic**.

As shown in Figure 16 one can also register a plugin GUI if necessary.

A simple example of the first option[7]is:

```
let run () =
let chan = open_out " hello.out" in
Printf.fprintf chan "Hello , world !\n";
close_out chan
let () = Db.Main. extend run
```

Listing 4.1: OCaml hello_world.ml

Listing 4.1, is a simple script that writes a print message, and registers the function run as an entry-point for the script. When executed Frama-C will call it if the script is loaded. The script can be loaded and compiled with **frama-c -load-script hello_world.ml**, which creates a executable **hello.out**.

The second option is to register a script as a plug-in. A illustrated in Listing 4.2:

```
let help_msg = " output a warm welcome message to the user "
module Self = Plugin.Register
(struct
let name = " hello world "
let shortname = " hello "
lethelp = help_msg
end)
let run () =
let chan = open_out " hello .out" in
Printf.fprintf chan "Hello , world !\n";
close_out chan
let () = Db.Main. extend run
```

Listing 4.2: OCaml Registered hello_world.ml

Registering a plugin is achieved by using the functor **Plugin.Register**. This functor takes as arguments three options:

- **name**, is a non empty string with the full name of the module;
- **shortname**, is a small string with a shortname of the module, normally used as prefix;
- **help**, is a string with the help and description of the module.

GamaSlicer uses the second option but uses it with a makefile that inherits from Frama-C **Makefile.dynamic**, and loads several different modules that use Frama-C modules.

Using the option **-load-script** is ideal for small experiments, but when a plugin becames larger and more complex with several files it is a good ideia to install it correctly by using a makefile. An example with the hello_world makefile:

7 Taken from Frama-C Developer Manual

```
FRAMAC_SHARE :=$(shell frama -c. byte -print -path )
FRAMAC_LIBDIR :=$(shell frama -c. byte -print -libpath )
PLUGIN_NAME = Hello
PLUGIN_CMO = hello_world
include $(FRAMAC_SHARE)/ Makefile.dynamic
```

Listing 4.3: hello_world Makefile

As shown in Listing 4.3, one must set some variables before including the generic **Makefile.dynamic**.

To run the plugin, one must first do **make** to compile it, and then load and execute the plugin using **frama-c -load-module ./Hello**.

If everything is correct, it is possible to install the plugin by running **make install**. After that the plugin is loaded everytime Frama-C is lanched.

For more information about how to build a Frama-C plugin it is recommend to read the Frama-C Developer Manual (available from their website) and Cuoq et al. (2012).

4.2.1 *Used Frama-C Modules*

Several different Frama-C modules where used in GamaSlicer, bellow follows a list of the used Frama-C modules and the reason for their use:

- **Ast** - To compute and retrive the Cil Abstract Syntax Tree;
- **Db** - To register the enter point of the plugin;
- **Cil_types** - To retrive Cil_types so GamaSlicer recognizes them.
- **Cfg** - To add Control Flow Graph to the Cil Abstract Syntax Tree;
- **Ast_info** - To get the names of the different functions in Cil_AST, and to find the default behaviors of ACSL clauses in the Cil_AST;
- **Globals** - To fold over the functions of the Cil_AST;
- **Kernel_function** - To retrieve the definition of a function;
- **Annotations** - To retrieve the function logic specification;
- **Cil** - To find the default behavior of a function and to create ghost statements[8];
- **Logic_const** - To create new logic constants when creating proof obligations;

8 Ghost Statements are not part of the actual code, but they are created as support to the generated graph. In this case, as start and end nodes of the slicegraph

- **Logic_utils** - To convert a C expression to a logic term;
- **Visitor** - To use Frama-C visit mechanism of the Cil_type Named Predicates;
- **Printer** - To print the several different Cil_types.

Another minor modules where used to do minor tasks, mostly sub-modules of the modules above.

4.3 IMPLEMENTATION

In this section the final approach will be explained in more detail. Then, a walkthrough of GamaSlicer different modules and how they integrate between themselves and Frama-C will be explained.

As stated before, GamaSlicer was implemented using OCaml, due to the fact that Frama-C requires it's plugins to be written in OCaml. Two other major libraries were also used, Why3 and OCamlgraph[9].

Why3 was used, because from its support for different provers, making GamaSlicer independent of a single prover. The provers must still be implemented and currently GamaSlicer supports several different provers but due to Why3 more provers can be added easily if needed.

Ocamlgraph was used, to implement the slicegraph, mainly due to its ease to use graph data structures, for being able to define your own data structure for the graph , it's performance and also providing several classic operations and algorithms over graphs.

4.3.1 *Modules*

GamaSlicer is divided in eight modules, which follow the Frama-C naming convention. These modules are: Gs_options, Towhy3, Vcgen, Provers, Slicing, Slicegraph, Gs_printer and Gs_register.

Gs_options

In this module, the plugin is registered with Frama-C, this is done by registering the name, shortname, help, etc. By registering the plugin, Frama-C will treat GamaSlicer as a plugin and not as a script. Also in this module the command line options are defined. As default, GamaSlicer applies Postcondtion-based slicing, but if called with option *-slice-type*, a different behavior can be set:

- *"post"* - GamaSlicer will apply Postcondtion-based slicing;
- *"prec"* - GamaSlicer will apply Precondition-based slicing;
- *"spec"* - GamaSlicer will apply Specification-based slicing.

9 OCamlgraph is a graph library for Objective Caml. OCamlgraph homepage: http://ocamlgraph.lri.fr/

Towhy3

Towhy3 is a module that converts Frama-C predicates into Why3 Term, using Why3 API [10] in a recursive algorithm. Currently only *Int* types and theories are supported due to the fact that module Vcgen only handles *Int* types. Also in this module, formulas are bounded with a first order quantifier depending on the type of slicing being invoked.

Vcgen

As the module states, in this module proof obligations, for each statement, are calculated using weakest precondition calculus and strongest postcondition calculus.

The algorithm used to implement the weakest precondition calculus and strongest postcondition calculus are presented bellow. Both algorithms receive a logic Hoare triple $\{\phi\}P\{\varphi\}$, Hoare (1969), where φ represents the postcondition and ϕ the precondition of a given program. P is a program, which translates to a sequence of statements, which is denoted by **;**. The algorithms listed use first order symbols, as normal also $\varphi[e/x]$ means that all ocurrences of x in formula φ are replaced by e.

Weakest precondition calculus algorithm:

$$\begin{aligned}
&wprec(\textbf{skip}, \varphi) = \varphi\\
&wprec(x := e, \varphi) = \varphi[e/x]\\
&wprec(C; S; \varphi) = wprec(C, wprec(S, \varphi))\\
&wprec(if\ e\ then\ S_t\ else\ S_f, \varphi) = (e \rightarrow wprec(S_t, \varphi)) \wedge (\neg e \rightarrow wprec(S_f, \varphi))\\
&wprec(while\ b\ do\ \{\theta\}\ S, \varphi) = \theta
\end{aligned}$$

Strongest postcondition calculus algorithm:

$$\begin{aligned}
&spost(\textbf{skip}, \phi) = \phi\\
&spost(x := e, \phi) = \exists .v\ \phi[v/x] \wedge x = e[v/x]\\
&spost(C; S; \phi) = spost(S, spost(C, \phi))\\
&spost(if\ e\ then\ S_t\ else\ S_f, \phi) = spost(S_t, b \wedge \phi) \vee spost(S_f, \neg b \wedge \phi)\\
&spost(while\ b\ do\ \{\theta\}\ S, \phi) = \theta \wedge \neg\phi
\end{aligned}$$

Using both algorithms above, the Vcgen module gets, from Frama-C Abstract Sintax Tree, the list of statements of the program and then applies a recursive weakest precondition calculus and a strongest postcondition calculus to the statements list. It creates the formulas using Frama-C Logic_utils module. Two proof obligation lists are returned, one corresponding to WP calculus and other to SP. Both lists are then stored in an hash table with the key being the function name and the value a tuple containing the two lists produced. Due to the complexity of generating proof obligations with pointers, and due to time constrains only *int* types were implemented.

10 Why3 API homepage: http://why3.lri.fr/API-0.85/

Provers

In this module, Provers are implemented using Why3 API. Provers must be correctly installed in the system and configured via command line using: *why3 configure -detect*. To configure a prover installed in the system one must provide its name and version. At the time being, GamaSlicer supports the following provers:

Alt-ergo, **Cvc4**, **Cvc3**, **yices**, **z3**, **e-prover**.

Yices, z3 and e-prover are not totally supported and require adjustments depending of the system and its options.

When receiving a formula to assert its validity, all provers must first build a task with the theories used in the formula and the formula itself. This task is then invoked in the specified prover. When the prover ends the computation then it reports the result back to to Why3 which can be of the following types:

- **Valid**, the task is valid;
- **Invalid**, the task is invalid;
- **Timeout**, the prover exceeds the time or memory limit;
- **Unknown**, the prover can't determine if the task is valid;
- **Failure**, the prover was unable to read its input task;
- **HighFailure**, an error occurred while trying to invoke the prover.

The provers also report the time in seconds they took to give an answer.

Slicing

Slicing Module is responsible for the execution of assertion based slicing algorithm. As stated previously, assertion based slicing is a term that encompass precondition-based slicing, postcondition-based slicing and specification based slicing.

Postcondtion-based slicing algorithm, receives an ordered list of statements and the corresponding weakest precondition proof obligations . In a reduce(fold) algorithm, the proof obligations from each statement are joined in a logic implication (creating a new formula) to lower proof obligations of the ordered list of statements and proof obligations until the list is empty. The new folded list, called slice results, contains the implications between statements, then they are dispatched to the prover where their validity is asserted.

Precondition-based slicing algorithm, also receives an ordered list of statements, but now instead of the proof obligations calculated by the weakest precondition calculus, they were calculated by strongest postcondition calculus. The algorithm to compute the slice results is the same as postcondition-based slicing.

Specification-based slicing algorithm, receives two ordered lists of statements, one with the proof obligations generated by weakest precondition calculus and other the other with the proof obligations produced by the strongest postcondition calculus. Also using a reduce algorithm, for each statement, its corresponding strongest postcondition proof obligation is joined in a logic implication with weakest preconditions proof obligations of lower statements. The new folded list, as in the algorithms above is also dispatched to the prover to assert the implications validity.

Slicegraph

Slicegraph module is the builder of the final Slicegraph, implemented using Ocamlgraph. A slicegraph is a Control Flow Graph which can have additional edges between statements representing the several different slices. Each Node has type Statement from Frama-C Cil_types, and the edges have type *int* and by default have weight 1.

Initially as the diagram in figure 15 shows, this module receives from Cil_AST module, the program AST, which is translated into a control flow graph. Two particularly important additional statements are added: the Start statement, which is connected with an edge to the first statement of the control flow graph, and End statement which is connected to the last statement of the control flow graph (both using Frama-C Ghost statement as implementation). These two statements are added to keep the graph consistency; for example if the first statement of a program is sliced, one must have always an initial statement that will never be sliced, and correspondingly if the last statement is sliced one must always have a End statement that will never be sliced. Also the Start statement is associated with the program precondition and the End statement is associated with the program postcondition.

After creating the initial Slicegraph, this module will be called again after the Slicing module computes the list of slice results. First, all results that don't have a valid answer are filtered, then using an imperative algorithm on the filtered list, depending on the type of slice, new edges are added.

If invoked with Postcondition-based slicing, when it receives a slice result between two statements and considering a slice result is a implication between two proof obligations with an corresponding statement , new edges are added from the predecessors of the left statement of the implication to the right statement of the implication.

When invoked with Precondition-based slicing, new edges are added from the left statement of the implication to the successors of the right statement of the implication.

Lastly when using Specification-based slicing, an edge is added between the left statement of the implication and the right statement of the implication.

After adding the new slice edges, several new paths of the program will be present. One could choose randomly which path to keep, but the best option is to choose the shortest path since we want the optimal slice of the program. To do this, a shortest path algorithm is applied to the Slicegraph, from the Start statement to the End statement (another reason to add these two ghost statements).

A problem when using shortest path is when it deals with conditional statements. It will always choose a branch instead of two. A solution, prior to run the shortest path algorithm, is:

1. Mark all conditional nodes;
2. Iterate over all conditional nodes with the following algorithm;
    a) Run shortest path on both branchs of the conditional statement;
    b) Store the path results and weights from both branches;
    c) Create a new edge with weight that equals the sum of the weight of the branches plus one, from the conditional statement to the first statement after the conditional blocks;
    d) Delete both branchs of the conditional statements.

After running the shortest path, Slicegraph module then maps the path by when encountering a conditional statement retrieving the shortest path of both branches, and adding both paths in place to the final path. Then converts it a format that can be printed (due to conditional branchs, if the path is printed without transformation it would not result in a valid C program).

Gs_printer

This module, as the name states, contains all printing functions that use the structures from all the other modules. This module is invoked in the main module.

Gs_register

This is the main module. Here Frama-C modules are invoked, including parser, Ast, and Ast_info. Also in here all the other modules are invoked according to the execution flow defined in the diagram of Figure 15, finally it is in charge of reporting the sliced output program or an error (in case one has concurred).

4.4 TESTING GAMASLICER

In this section we will test GamaSlicer with different input C programs and with the three assertion based slice techniques, postcondition-based slicing, precondition-based slicing and specification-based slicing.

4.4.1 *GamaSlicer Poscondtion-based slicing*

In this subsection we will test the plugin with postcondtion-based slicing. To invoke the plugin with postcondition-based slicing, the option is **-slice-type ”post”**.

The first input is a simple example with postcondition $x \geq 0$ and can be seen in Listing 4.4:

```
/*@ ensures x >= 0;
*/
```

```
void f(int x){
  x = x-150;
  x = x+100;
  x = x+100;
}
```

Listing 4.4: post1.c

The Weakest preconditons calculated by GameSlicer for each line are:

- WP_Line1 : $x \geq -50$
- WP_Line2 : $x \geq -200$
- WP_Line3 : $x \geq -100$

We can see that the first two instructions will be sliced because WP_Line1$\rightarrow$ WP_Line3 , according to the slicegraph algorithm stated in subsection 4.3.1. The output of GamaSlicer with Listing 4.4 is:

```
/*@ ghost ; */
x += 100;
/*@ ghost ; */
```

Listing 4.5: Output from post1.c

As expected both the first and last statement were sliced. The ghost statements are the Start and End statement as stated in previous section.

The second example also has the same postcondition $x \geq 0$, but with different statements as shown in Listing 4.6:

```
/*@ ensures x >= 0;
*/

void f(int x){
  x = x+100;
    x = x+50;
  x = x-100;
}
```

Listing 4.6: post2.c

GamaSlicer should slice every statement after the first one because the first statement satisfies the postcondition. As show in Listing 4.7 , GamaSlicer does precisely that:

```
/*@ ghost ; */
x += 100;
/*@ ghost ; */
```

Listing 4.7: Output from post2.c

To test GamaSlicer with more complex program, an example of taxes calculation was used as shown in Listing 4.8:

```
/*@ requires age >= 18;
  @ ensures personal >= 5750;
*/

void taxesCalculation(int age, int income, int personal, int t){

  if(age >= 75){ personal = 5980; }
  else if(age >= 65){ personal = 5720; }
     else { personal = 4335; }

  if((age >= 65) && (income > 16800))
  {
    t = personal - ((income -16800)/2);
    if (t > 4335){ personal = t + 2000; }
    else { personal = 4335; }
  }
}
```

Listing 4.8: taxes_calculation.c

The program in Listing 4.8 has precondition $age \geq 18$ and postcondition $personal \geq 5750$. Using chained conditional statements calculates the personal taxes depending on the age of the individual.

The output from GamaSlicer is shown in Listing 4.9

```
/*@ ghost ; */
if (age >= 75)
{
personal = 5980;
}else{
if (age >= 65)
}
if (
age >= 65)
{
if (income > 16800)
}else{
}
/*@ ghost ; */
```

Listing 4.9: Output from taxes_calculation.c using Poscondition-based slicing

All non conditional statements were sliced except *personal $\bar{5}$980* as it was expected since the postcondition is $\geq$ 5750. One will note that much of conditional statements are not sliced, but its branches are empty. That happens when an conditional is chained to another conditional and its branches were also sliced and because slicing a conditional statement is breach of control flow and if done changes the control flow of the program. On a side note, one can see CIL in action since it decomposed the conditional statement with the $\wedge$ in two conditional statements chained to one to another each one with expression composing the separated Logic *and*.

4.4.2 *GamaSlicer Precondition-based slicing*

In this subsection we will test the plugin with precondition-based slicing. To invoke the plugin with precondition-based slicing, the option is **-slice-type "prec"**.

In Listing 4.10, it is shown a simple example with precondition $x \geq 0$:

```
/*@ requires x >= 0;
*/

void f(int x){
  x = x+100;
  x = x-200;
  x = x+200;
}
```

Listing 4.10: pre1.c

Since the last two statements do not violate the final strongest precondition both can be removed, as it is shown in Listing 4.11 :

```

/*@ ghost ; */
x += 100;
/*@ ghost ; */
```

Listing 4.11: Output from pre1.c

The second input, in Listing 4.12 is a more complex one that demonstrates a conditional statement:

```
/*@ requires x >= 0;
*/
```

```
void f(int x){

  if(x>0){
  x = x+100;
  x = x-200;
  x = x+200;
  }else{
  x = x-150;
    x = x-100;
  x = x+100;
  }
}
```

Listing 4.12: pre2.c

It also has the precondition $x \geq 0$ and since the expression in the conditional is the same as the precondition the second branch should be sliced. The block inside the branch should be sliced as the first input (shown in Listing 4.10). The output from GamaSlicer is shown in 4.13:

```

/*@ ghost ; */
if (x > 0)
{
x += 100;
}else{
x -= 150;
}
/*@ ghost ; */
```

Listing 4.13: Output from pre2.c

As expected the first branch was sliced like the first example (Listing 4.11). All the statements in the second branch should have been sliced, but the first statement was not. This is due to the prover not being able assert the validity of the implication of the conditional statement strongest postcondition with the statement strongest postcondition, although the formula is proven valid by a human. Using different provers or with different options to assert the validity of the implication will slice all the statements in the second branch.

Using the previous example of taxes calculation, shown in Listing 4.8, the output from GamaSlicer using Precondition-based slicing is shown in Listing 4.14

```

/*@ ghost ; */
/*@ ghost ; */
```

Listing 4.14: Output from taxes_calculation.c using Precondition-based slicing

Remembering that precondition of Listing 4.8 is $age \geq 18$, everything is sliced. This is expected because the precondition was too strong and nothing in the program is relevant to affect that precondition.

4.4.3 *GamaSlicer Specification-based slicing*

In this last subsection we will test the plugin with Specification-based slicing. To invoke the plugin with Specification-based slicing, the option is **-slice-type ”spec”**.

The first example of specification-based slicing like the previous subsections is a simple program with precondition being the most weak possible (*true*) and postcondition being $x \geq 100$, as shown in Listing 4.15:

```
/*@ requires \true;
  @ ensures x >= 100;
*/

void f(int x){
  x = x*x;
  x = x+100;
  x = x+50;

}
```

Listing 4.15: spec1.c

Since the precondition is the most weak of them all, only after the second statement there should be any slice. The output from GamaSlicer is shown in Listing 4.16

```

/*@ ghost ; */
x *= x;
x += 100;
/*@ ghost ; */
```

Listing 4.16: Output from spec1.c

As expected GamaSlicer only sliced the last statement since it didn’t affect the conditions.

The second example, shown in Listing 4.17 of specification-based slicing is a program with a conditional statement, with precondition $y \geq 10$ and postcondition $x \geq 0$.

```
/*@ requires y > 10;
  @ ensures x >= 0;
*/

void f(int x,int y){
  if(y>0){
  x = 100;
  x = x+50;
  x = x-100;
  }else{
  x = x-150;
    x = x-100;
  x = x+100;
  }
}
```

Listing 4.17: spec2.c

In this example we can see clearly the advantage of using specification-based slicing as shown in Listing 4.18

```
/*@ ghost ; */
if (y > 0)
{
x = 100;
}else{
x -= 150;
}
/*@ ghost ; */
```

Listing 4.18: Output from spec2.c

Due to precondition being $y \geq 10$ one can assume that the second branch of the conditional statement should be sliced, but like the output in Listing 4.13 the prover is not able to assert the validity of the formula. The first branch was sliced due to postcondition $x \geq 0$ since after the first statement in the branch this property is already respected.

As with other types of assertion based-slicing the taxes calculation program(shown in Listing 4.8) was used test specification-based slicing of GamaSlicer:

```
/*@ ghost ; */
if (age >= 75)
{
personal = 5980;
}else{
if (age >= 65)
}
if (
age >= 65)
{
```

```
if (income > 16800)
}else{
}
/*@ ghost ; */
```

Listing 4.19: Output from taxes_calculation.c using Specification-based slicing

The output is the same as Listing 4.9 when using Postcondition-slicing. This is due to preconditon $age \geq 18$, because age is only used in conditional statements and the value of the variable age is never altered.

5 CONCLUSION

This document starts with an overview of Slicing techniques. Then a more detailed explanation of the several slicing techniques called Assertion-based slicing (Postconditon-based slicing, Precondition-based slicing and Specification-based slicing) is given.

The intermediate language, CIL was exaplained was well as its importance to Frama-C. After, Frama-C and its various plugins were explored. Due to lack of documentation, it was hard to find what kind of slicing is implemented. The slicing plugins do not use semantic information but they bypass this weakness by running the value analysis plugin first. Slicing plugins are all very similar which is strange. Frama-C developers seem to use slicing as mean to an end and not slicing as main feature; that can explain the diversity of slicing plugins.

Frama-C was explored and a plugin that showcases assertion-based slicing techniques was developed with success. The strengths of assertion-based slicing were shown (being able to slice a program by its contract) and its limitations (being dependent on provers and their performance) were also shown.

In Chapter 4 an extensive description of the developed plugin was presented. To develop a Frama-C plugin it was necessary to learn OCaml language and Frama-C philosophy and architecture. There were three different approaches to the architecture of the plugin, the first two failed due to the difficulty in using Frama-C WP plugin API, because that plugin was not developed with the idea of being used in Slicing (for example not being able to provide a proof obligation per statement). That fact delayed this thesis timeframe, because a Vcgen was developed which was not initially expected. Although the first two approaches failed, the last one was successfully implemented and, as shown by the examples, GamaSlicer was able to slice several different programs using the program contracts as slicing criteria. Postcondtion-based slicing, Precondition-based slicing and Specification-based slicing were all implemented with success and their utility in slicing programs was shown. GamaSlicer is a positive contribution to the Frama-C family of plugins, and shows that Frama-C can actually be used to perform different kinds of C code analysis. Frama-C developers should improve its documentation and specially its API, which will improve the good reputation that already enjoys and increase the community of C analyzers around Frama-C. Due to the failure of the two first approaches, there was not enough time to extend the Vcgen module of GamaSlicer to support dynamic memory, and so GamaSlicer was not applied to large C unix kernal libraries.

As future work, GamaSlicer Vcgen module can be extended to support dynamic memory. Also a Graphic User Interface shall be developed to shown more easily the slicegraph and its different slices.

BIBLIOGRAPHY

José Bernardo Barros, Daniela Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. *Formal Aspects of Computing*, 24(2):217–248, 2012. ISSN 0934-5043. doi: 10.1007/s00165-011-0196-1. URL http://dx.doi.org/10.1007/s00165-011-0196-1.

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.

Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Combining static analysis and test generation for c program debugging. In Gordon Fraser and Angelo Gargantini, editors, *Tests and Proofs*, volume 6143 of *Lecture Notes in Computer Science*, pages 94–100. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13976-5. doi: 10.1007/978-3-642-13977-2_9. URL http://dx.doi.org/10.1007/978-3-642-13977-2_9.

Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1284–1291, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2245276.2231980. URL http://doi.acm.org/10.1145/2245276.2231980.

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

JosephJ. Comuzzi and JohnsonM. Hart. Program slicing using weakest preconditions. In Marie-Claude Gaudel and James Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 557–575. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-60973-5. doi: 10.1007/3-540-60973-3_107. URL http://dx.doi.org/10.1007/3-540-60973-3_107.

Daniela Cruz. *Verification, slicing, and visualization of programs with contracts*. PhD thesis, Universidade do Minho, 2011. URL http://repositorium.sdum.uminho.pt/handle/1822/19646.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33825-0. doi: 10.1007/978-3-642-33826-7_16. URL http://dx.doi.org/10.1007/978-3-642-33826-7_16.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

Liu Hua-Xiao, Jin Ying, Chi Xue-Hang, Li Junrong, Li Yu-Shuang, and Xu Yong. A new dynamic program slicing algorithm based on abstract machine. In *Proceedings of the 2013 International Conference on Computational and Information Sciences*, ICCIS '13, pages 738–741, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5004-6. doi: 10.1109/ICCIS.2013.199. URL http://dx.doi.org/10.1109/ICCIS.2013.199.

Paritosh Jain and Nitish Garg. A novel approach for slicing of object oriented programs. *SIGSOFT Softw. Eng. Notes*, 38(4):1–4, July 2013. ISSN 0163-5948. doi: 10.1145/2492248.2492266. URL http://doi.acm.org/10.1145/2492248.2492266.

J. Jiang, X. Zhou, , and D.J. Robson. Program slicing for c - the problems in implementation. In *Proceedings of Conference on Software Maintenance*, pages 182–190. IEEE CSPress, 1991.

B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(88)90054-3.

Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

GeorgeC. Necula, Scott McPeak, ShreeP. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In R.Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43369-9. doi: 10.1007/3-540-45937-5_16. URL http://dx.doi.org/10.1007/3-540-45937-5_16.

Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187674. URL http://doi.acm.org/10.1145/2187671.2187674.

F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. URL citeseer.ist.psu.edu/tip95survey.html.

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.

Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL `http://dl.acm.org/citation.cfm?id=800078.802557`.