UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

STATIC VERIFICATION OF DATA RACES IN OPENMP

Kátya Thaís Martins da Silva

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

STATIC VERIFICATION OF DATA RACES IN OPENMP

Kátya Thaís Martins da Silva

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Dimitris Mostrous

2014

Agradecimentos

Quero agradecer ao meu orientador Dr. Alastair Donaldson e ao meu co-orientador Dr. Jeroen Ketema, ambos da Imperial College of London, por aceitarem que trabalhasse com eles na minha tese e a todo o grupo Multicore por me ter acolhido e ajudado quando necessitei. Não posso deixar de agradecer também ao meu orientador da Faculdade de Ciências Dr. Dimitris Monstrous, que me ajudou sempre que necessitei e que foi essencial para a minha deslocação para Londres. Muito obrigada sem vocês nada disto teria sido possível.

Também quero agradecer à minha família por toda a forca e motivação que me deram. Aos meus pais, Maria e José da Silva, por todo o carinho, força e motivação nestes meses tao intensos de trabalho; a minha irmã, Patrícia que sempre me motivou e animou quando estava mais desanimada e ao meu irmão, Theo que apesar de pequeno esteve sempre ao meu lado quando precisei de uma gargalhada. Obrigada por tudo.

Por último, e não menos importante, quero agradecer ao meu namorado, Fábio, que apesar de estar longe sempre me deu muita força mesmo nos momentos em que estava mais stressada e rabugenta; e também quero agradecer aos meus colegas por estes anos todos de faculdade e a duas colegas em especial por me darem muita força quando precisei neste ultimo ano. Obrigada a todos.

Muito obrigada a todos que me acompanharam e me deram força nesta etapa da minha vida.

Á minha família.

Resumo

Com o uso de *linguagens paralelas* nascem alguns problemas como *data race* , uma argura para os programadores durante o seu trabalho. *Data race* são quando dois ou mais *threads* em um único processo acedem a mesma posição de memória ao mesmo tempo, sendo que um ou mais acessos são para a escrita/leitura, sem sincronização adequada.

A nossa ferramenta (*SVDR: Static Verification of Data race in OpenMP*) foi desenvolvida com o propósito de verificar a ausência de *data races* em *OpenMP* (uma linguagem de programação paralela). Empregamos a *verificação estática* com intenção de encontrar *data races* sem necessariamente executar o programa, possibilitando dessa forma ao utilizador, encontrar o problema antes de receber o resultado do mesmo. Para melhor compreensão de como funciona a ferramenta, é pertinente apresentar de forma concisa, as linguagens de programação aqui abordadas: *OpenMP* e *Boogie*.

OpenMP (Aberto Multi-Processing) 2.1 é um *API* usado na programação *(C, Fortran, C++)* em multiprocessadores com memória partilhada em multiplataformas e funciona em quase todos *Sistemas operativos e arquitecturas*.

Boogie [2, 17] é uma linguagem de verificação intermédia, projectada como uma camada sobre a qual é possível construir outros verificadores para outras línguaguens.

Uma vez aclaradas as definições, é mais fácil contextualizar e compreender como a ferramenta funciona.

Como já foi referido previamente, o objectivo da ferramenta é traduzir código *OpenMP* em código *Boogie* com o propósito de verificar se existe ou não *data race* no ficheiro de entrada. Assim sendo, é possível aferir que existem duas etapas para o processo de verificação: a primeira é a tradução de *OpenMP* em *Boogie*, e a segunda é a verificação de *data race* no ficheiro de entrada - que é do tipo *bpl*, ou seja, código *Boogie*.

A primeira fase não é tratado pela ferramenta, de modo que é necessária uma tradução manual do código *OpenMP* para o código *Boogie*.

A segunda fase é a ferramenta que foi implementada, *SVDR*. O *SVDR* funciona da seguinte forma: recebe como entrada um *ficheiro bpl* (um *OpenMP* traduzido em código *Boogie*) e retorna como resultado um *ficheiro bpl* com o algoritmo necessário para verificar se há ou não um *data race* no ficheiro de entrada.

O algoritmo criado é baseado no uso de *nao determinismo*, *invariantes* e *assertions*. Este funciona do seguinte modo, para verificar a existência de *data race* num ciclo *While*, o que se faz é guardar numa variável global, que designamos de *current_off*, o valor actual do índice do array (independentemente de se tratar de uma leitura ou uma escrita, o processo é o mesmo), e atribuir a outra variável global, que denominamos de *current*,o valor de true. Após guardados os referidos valores, o que se faz a seguir, é a incrementação do ciclo, atribuímos os valores do *current* e do *current_off* a duas novas variáveis globais *previous* e *previous_off*. Estas atribuições são feitas para evitar que se aceda duas vezes seguidas a mesma posição, porque isso poderia conduzir a *data race*.

Tanto as atribuições de *current* como de *previous*, são feitas dentro de *if não determinísticos*. *Não determinismo* é por definição "Uma propriedade da computação na qual pode ter um ou mais resultados", mas neste caso especifico significa que não é possível saber se entra ou não no *if*. A razão para o seu uso, é apenas a não necessidade, neste caso, de se guardar todos os valores do *currrent*, e mudar todos os valores do *previous* (até porque se tivermos em consideração todas as possibilidade de execução do programa, haverá sempre uma para cada tarefa).

Depois de guardados os valores é necessário validá-los de alguma forma, até porque senão, seriam apenas *if* num programa, ou seja, não teriam nenhum efeito especifico. A modo utilizado para verificar, é empregando *assercoes*, ou seja, criamos um *assert* na qual se afirma que caso o *previous* seja verdadeiro, então o *previous_off* será sempre diferente do índex actual do vector. Caso isto seja verdade, ou seja, caso se ocorra no programa a veracidade dessa afirmação, então não existem *data race*, caso contrário, existem. Mas para que esse *assert* seja sempre verdade, durante o programa, é indispensável a criação de uma *invariant* para suportar a sua veracidade. No nosso caso, as *invariantes* são testadas usando *Houdini* 2.2. São criadas bases de *invariantes* pela ferramenta, que posteriormente ao realizar-se a verificação do ficheiro com o algoritmo no *Verificador Boogie*, adiciona um atributo relativo ao *Houdini*. Por fim, para saber se existe ou não *data race* no código, utilizamos o *Verificador Boogie* para atestar o programa com o algoritmo.

Como conclusão, o que se pode dizer é que a ferramenta apresentou resultados bastante satisfatórios, e que em todos os exemplos apresentados e testados, o efeito foi o esperado. Apesar de algumas limitações, a ferramenta cumpre com o que foi descrito anteriormente *Verifica estaticamente a existência de data race na linguagem OpenMP*.

Palavras-chave: Boogie, Data race, OpenMP, Paralelismo, Verificaçao

Abstract

Increasingly, programmers use multi-core processors to develop code with multiple threads, i.e, parallel programs. There are some tools that support parallelism such as *Intel Parallel Lint* 2.4.1 or *Intel Thread checker* 2.4.2 and some parallel programming languages such as *OpenMP* 2.1.

With the use of *parallel languages* emerged some problems such as *data races* that no programmer likes to come across when working. *Data races* are when two or more threads in a single process access the same memory location concurrently and one or more of the accesses are for writing without proper synchronization.

Our tool (*SVDR: Static Verification of Data race in OpenMP*) was developed with the intention to verify *data race* freedom in *OpenMP* (a parallel programming language). We used *static verification* because we wanted to try to find *data races* without running the program because that way the user discovers the problem before getting results from the program.

The *SVDR* tool works as follows: it receives as input a *bpl file* (a file with the translated *OpenMP* code into *Boogie* code), then the tool performs executes the algorithm on the input, and gives as output the execution, i.e, the *Boogie code* with the algorithm necessary to verify if there is or not a *data race*.

Next the output of the *SVDR* is going to be used as input in the *Boogie Verifier* that will determine whether or not there exists a *data race* in the input. If there is a *data race* then the verifier will give an error, otherwise the *verifier* will verifies the code without any problem.

After several tests it was possible for us to verify that the tool works correctly in all tests that we ran. The examples that we ran were with nested loops, but just one of them was parallel, and also with simple loops, with reads and writes, and all of the examples were verified as expected by the *Boogie Verifier*.

The tool is useful to help the user to verify is exists any *data race* problem in code so that they can solve the problem if it exists.

Keywords: Boogie, Data races, OpenMP, Parallelism, Verification

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Increasingly, programmers use multi-core processors to develop code with multiple threads, i.e. parallel programs. There are some tools that support parallelism like *Intel Parallel Lint* 2.4.1 or *Intel Thread checker* 2.4.2 and some parallel programming language like **OpenMP** [23].

The problem with parallelism is that often accesses (read and/or write) simultaneously can lead to errors and bugs that no programmer likes to face. This bugs normally are *data races*, a very a sensitive issue that concerns the programmer.

To understand better what a *data race* is we give a brief definition that a *data race* is when two or more threads in a single process access the same memory location concurrently and one or more of the accesses are for writing without proper synchronization and they can lead to *non-determinism.* More details on *data races* present in Chapter 2.1.3.

The main reason that *data races* are such a big problem its because we can not always figure out why or where they occurred. Our tool tries to help the user by doing a static verification of the absence of *data races* in *OpenMP*.

The purpose of the tool is to transform the *OpenMP* code into **Boogie** and check/detect the races in *OpenMP*. Before explaining how the tool works it is important to give a short definition of what *OpenMP* 2.1.

OpenMP (Open Multi-Processing) 2.1 is an *API* used for programming *(C, Fortran, C++)* multi-processors with shared memory at multi-platforms on almost all *Operating Systems* and processor architectures.

Boogie [2] [17] is an intermediate verification language, designed as a layer on which it is possible build other verifiers for other languages.

Now that the definitions of *OpenMP* and *Boogie* are a bit clear we are going to explain in general how does the tool works and what it does.

The aim of the tool is to translate *OpenMP* code into *Boogie* code and verify if there exists *data races*, so we can divide our tool in two parts: first one is the translations of

OpenMP into *Boogie* and the second one is the verification of the absence of *data races* in the *Boogie* code, i.e., the input file. The first part was not handled by the tool, so the way to translate *Boogie* code from *OpenMP* is manually. In principle, this could be automated via the construction of a class that analyze the *OpenMP* language and translate it to *Boogie*. The second part is the tool that we implemented. The tool receives as input *Boogie* code and transforms it into code with the capacity to detect *data races* when checked by the *Boogie Verifier*. To make the tool work we use *invariants* and to test them we decided to use *Houdini* [2.2]. We explain in more detail how we used *Houdini* and *Invariants* in Chapters 2 and 5.

To make sure that the tool was working properly, we did some tests as described in Chapter 5 and we showed some of the results of these tests. That way it is easy to visualize what the tool does and how it behaves.

Although the tool does what it is supposed to do, like some other tool it has some limitations that need to be considered for it to work the best way possible.

We tried to build a tool that was efficient and good enough to handle solid examples, and that fulfills its purpose in the most advantageous manner for the user.

The rest of the report explains in detail what we have done and how we built the *static verifier of Data races in OpenMP (SVDR).*

1.2 Contributions

The main contributions of this work are:

- Static verification of the absence of *data races* in a specific programming language;
- Use of a sequential language to achieve *Race detection* for a parallel language;
- Algorithm for *data race* detection on a sequential language;

1.3 Document Structure

This report is divided into seven chapters.

- **Chapter 1** presents a vision of work that includes the introduction / motivation, objectives and structure of the document.
- **Chapter 2** is dedicated to the related work where we describe *OpenMP* [23], *Boogie* [17], *Static and Dynamic Analysis* and *Houdini*. In the two first approaches we discuss the respective definitions, the programming syntax, some examples and some definitions and examples of data race (this relate to the **OpenMP** [23] topic).

- **Chapter 3** is about the translation of *OpenMP* into *Boogie*. We explain how we translate from simple *OpenMP* code into *Boogie* code with the purpose of finding *data races* and we provide some examples of that type of translation.
- **Chapter 4** is the *Implementation* chapter. In this one we explain how we implemented the tool and what limitations has.
- **Chapter 5** this is the *Test* chapter. We show how we tested the tool and how it is supposed to be used.
- **Chapter 6** is dedicated to the tool results. We show some input and output results using the tool.
- **Chapter 7** is the *Conclusion and Discussion* chapter where we discuss what was the purpose of this work and what can be done in the Future.

Chapter 2

Related Work

2.1 OpenMP

2.1.1 What is OpenMP?

OpenMP (Open Multi-Processing) [10, 23, 29, 30] is an **API** used for programming *(C, Fortran, C++)* multi-processors with shared memory at multi-platforms on almost all Operating Systems and processor architectures. It is a simple, portable, scalable tool for developing parallel applications for platforms from simple PCs to supercomputers.

OpenMP is a multi-threaded application that starts with a single thread (the master thread). As the program runs and when the master thread finds some parallel block, it creates several threads. These threads will execute blocks of code in parallel. When the parallel block ends the threads join the master thread which continues until the program ends.

It is usual that each thread runs independently in each parallel region. Using **OpenMP** it's possible to achieve data parallelism (simultaneous execution of multiple cores that have the same function using a data set) and task parallelism (simultaneous execution of multiple cores that have different functions using the same or different data set).

The threads are distributed according to the processors and the performance of the machine among other factors. The **OpenMP** functions are all in the header file ("omp.h" in C / C++).

2.1.2 Features of OpenMP [10, 22, 29, 23]

The following structure is based on Wikipedia [28].

- Data sharing attribute clauses
  - *shared:* All the threads in a team have access to the variables in the *variable-list*, i.e, all variables in that list are shared by the threads.

```
#pragma omp parallel for shared(variable-list))
    block of code

```

  - *private:* The variables in the *variable-list* are private to each thread, meaning that each thread have a copy of the variables in the list.

```
#pragma omp parallel for private(variable-list))
    block of code

```

- Synchronization clauses
  - *critical:* A thread waits at the beginning of the critical region until no other thread is running in that critical region with the same name, i.e, only one thread at a time can execute the block of code.

    The name is optional and is used to identify the critical region.

```
    #pragma omp critical [(name)]
        block of code

```

  - *atomic:* A specific memory location is updated atomically (no other thread can access the location during the execution of the event). It is used to update the memory (write, or read-modify-write) of the variables and counters.

```
    #pragma omp atomic
        expression statement

```

  - *barrier:* Each thread waits until all the other threads have reached the barrier, where all of the threads synchronize. After the barrier each thread executes the statement in parallel.

```
#pragma omp barrier

```

- Scheduling clauses
  - *schedule (type, chunk):* The scheduling algorithm for the loop can be controlled. The most important types of scheduling are:
    - *static:* each thread decides what piece/chunk of the loop they will process.

```
#pragma omp for schedule(static)
   block of code

```

    - *dynamic:* there is no order in which the threads are assigned to the different items of the loop.

```
#pragma omp for schedule(dynamic)
   block of code

```

- Reduction
  - *reduction(operator | intrinsic : list):* Each thread has a local copy of the variable, but the values of local copies will be reduced on a global variable.
- Others
  - *omp for:* divides loop iterations between the threads.

```
#pragma omp for
   for-loop block of code

```

  - *sections:* defines which sections of code will run in parallel.
  - *master:* code block will be executed by the thread master only.

```
#pragma omp master
   block of code

```

Examples

- This is a simple **Hello World** [10, 22] piece of code with threads. It has been adapted from the tutorial [29]. The result will be the string and the number of the thread. The barrier directive causes threads encountering the barrier to wait until all the other threads in the same team (process) have encountered the barrier. In this case we use the barrier so that the last *print* will appear always last and not in the middle of the other *prints*. In this piece of code the *private* keyword specifies that *th_id* is private (each thread has their own copy of it) which ensures that the last line is printed only once [20].

```

int main (int argc, char *argv[]) {
  int th_id, nthreads;
  #pragma omp parallel private(th_id){

    // this gives the unique ID of a thread
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);
    #pragma omp barrier
    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }
  return 0;
}
```

- This example shows simple `for` loops containing print statements. It has been adapted from the tutorial [29]. It uses schedule static [22, 29] because each thread independently decides which chunk of the loop they will process, and dynamic means there is no predictable order in which the loop items are assigned to different threads [20].

  In this example the output result for the *static* will be the **"0,1,2,3,4,5,6,7,8,9."** and for the *dynamic* will be in a random order, depending on the order of the threads.

```
#pragma omp for schedule(static)
  int n;
  for(n=0; n<10; n++)
    printf(" %d", n);
  printf(".\n")


#pragma omp for schedule(dynamic)
  int n;
  for(n=0; n<10; n++)
    printf(" %d", n);
  printf(".\n");
```

- This example shows a simple loop where b is incremented with *a*. It is specified that *a* is private (each thread has its own copy of it) and that *b* is shared (each thread accesses the same variable). In this case the atomic [22, 29] variable (only one thread may access it simultaneously), would negate the advantages of parallelism [20].

```
int main(int argc, char *argv[]) {
  int a, b=0;
  #pragma omp parallel for private(a) shared(b)
  for(a=0; a<50; ++a)
    #pragma omp atomic
    b += a;
  return b;
}
```

- This example shows a simple parallel loop which stores some values in an array. Parallel [22, 29] for is used to divide loop iterations between the spawned threads [20].

```

int main(int argc, char *argv[]) {
  int i, a[10];
  #pragma omp parallel for
  for (i = 0; i < 10; i++)
    a[i] = 3 * i;
  return i ;
}
```

2.1.3 Data Races

What is a Data Race?

A data race [1] is when two or more threads in a single process access the same memory location concurrently and one or more of the accesses is for writing without proper synchronization (no lock controls the accesses to the memory).

When the conditions described above are satisfied and the order of accesses is non-deterministic, the result may be different each time the program is run.

It is important to have in mind this table of when a **data race** occurs:

		Data race?
Read	Read	No
Read	Write	Yes
Write	Read	Yes
Write	Write	Yes

Why data races are bad?

"Even though most data races are harmless, the harmful ones are at the heart of some of the worst concurrency bugs. Alas, spotting just the harmful data races in programs is like finding a needle in a haystack: 76% - 90% of the true data races reported by state-of-the- art race detectors turn out to be harmless." This paragraph is taken from [15] and it is a small demonstration of the danger of the data races and how bad they can be to a program.

The simplest way to create a race condition [1, 18] is to write to a shared variable in a parallel region. In this case, the final value of the variable depends on the order of the writes. In a parallel execution we cannot guarantee that this is always the last iteration (as in sequential execution).

Another cause of race conditions [10, 15] is the loop carried data dependency in parallel loop, which is the example of *OpenMP* where the loops are parallel. Usually this occurs when an array is indexed improperly. For example, if we have an array *"a"* and a loop counter *"x"* the value of *"a[x]"* is different in every loop iteration.

Example of a Data Race

- This is a simple example of the result of a matrix multiplication, in which we multiply two matrices and get a new one.

  The structure of the code was adapted from [10], has been optimized and the part that create the matrix was changed.

```
Matrix Multiplication

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]){
  int i, j, k;
  double sum;
  double a[5][5], /* matrix A to be multiplied */
  b[5][5], /* matrix B to be multiplied */
  c[5][5]; /* result matrix C */
  /*** Spawn a parallel region explicitly scoping all
   variables ***/

  #pragma omp parallel shared(a,b,c) private(i,j,k,sum)
  {
    printf("Initializing matrices...\n");
    /*** Initialize matrices ***/
    #pragma omp for schedule (static)
    for (i=0; i<5; i++)
      for (j=0; j<5; j++)
        a[i][j]= 1;

    #pragma omp for schedule (static)
    for (i=0; i<3; i++)
      for (j=0; j<3; j++)
        b[i][j]= 2;

    #pragma omp for schedule (static)
    for (i=0; i<5; i++)
      for (j=0; j<5; j++)
        c[i][j]= 0;

    #pragma omp for
    for (i=0; i<5; i++) {
      for(j=0; j<5; j++) {
        for (k=0; k<5; k++) {
          sum =sum+ a[i][k] * b[k][j];
        }
        c[i][j] = sum;
        sum=0;
      }
    }
  } /*** End of parallel region ***/
45    /*** Print results ***/
46    printf("*****************************************\n");
47    printf("Result Matrix:\n");
48    for (i=0; i<5; i++)
49    {
50      for (j=0; j<5; j++)
51        printf("%6.2f ", c[i][j]);
52      printf("\n");
53    }
54    printf("*****************************************\n");
55    printf ("Done.\n");
56  }
```

A data dependency exists if the same element of an array is written on one loop iteration and read on another. In the example before (**Matrix Multiplication**) if we change *line 15* in the code to **#pragma ompallel shared(a,b,c) private(i,j,k)** what would happen is a *data race* because the *variable sum* is not in *private* (the values of the result matrix will be different like in the figure 2.1).

The motive that sum needs to be private it is because each thread will have a local copy of *sum* and use it as a temporary variable.

```
Result Matrix:
6.00 6.00 6.00 0.00 0.00
6.00 6.00 6.00 0.00 0.00
6.00 6.00 6.00 0.00 0.00
10.0 6.00 6.00 0.00 0.00
6.00 6.00 6.00 0.00 0.00
******************************************************
Done.
******************************************************
Result Matrix:
6.00 6.00 6.00 0.00 0.00
6.00 6.00 6.00 0.00 0.00
6.00 6.00 6.00 0.00 0.00
6.00 6.00 6.00 0.00 0.00
6.00 6.00 6.00 0.00 0.00
******************************************************
Done.
```

Figure 2.1: Data race example

2.2 Boogie

2.2.1 What is Boogie?

Boogie [2, 17] is an intermediate verification language, designed as a layer on which we can build other verifiers for other languages. There are several verifiers that were built on this basis such as **VDC** and the **HAVOC** verifier for **C**, and verifiers for **Dafny**, **Chalice**, **Spec#** [19] and **GPUVerify** [3].

The previous version of the language was called BoogiePL [6] and now is known as Boogie (version 2).

The tool has the same name as the language (Boogie). It accepts programs written in the language as input, optionally deducts some invariants of the program and generates verification conditions that are passed to an **SMT** solver which by default is **Z3**.

Boogie's Architecture

The following section in based on the paper "Boogie: A modular reusable verifier for object-oriented programs" [2], and explains *Boogie's Architecture*.

1. Design-Time Feedback

   Boogie (along with the *Spec#* compiler) together with **Microsoft Visual Studio** [17] are integrated to provide feedback at design time in the form of red underlines that give emphasis not only to pre-condition violations but also to syntax errors.

2. Distinct Proof Obligation Generation and Verification Phases.

   The Boogie [2, 17] pipeline has intermediate representations based on **Boogie**, adapted to express proof obligations and assumptions in the language. **Boogie** plays a key role in the separation of generation of proof obligations of semantic encoding of the source program and proofs of such obligations. This separation has been instrumental in the simultaneous development of the verification methodology and object-oriented program verification technology core.

3. Abstract Interpretation and Verification Condition Generation.

   Boogie [2, 17] uses abstract interpretation to perform loop-invariant inference and to generate verification conditions that are passed to an automatic theorem prover. This combination allows Boogie to utilize both the precision of verification condition generation (that is necessarily lost in an abstraction) and the inductive invariant inference of abstract interpretation (that simply cannot be obtained with a concrete model).

2.2.2 Boogie Syntax

This section gives a brief overview of the basics features of Boogie 2 and is base on the manual/tutorial [17].

Boogie Declarations [2, 6, 17]

- *Type declarations* correspond to type constructors:
  - `type` *X*; Declares a type that represents something.
- *Symbolic constants* correspond to symbolic constants:
  - `cons` *x: X*; Says that x is a fixed value of type X.
- *Function declarations* induce mathematical functions:
  - `fuction` func(*X*) `return` (`int`); Declares a function that returns the result of *func* that uses a variable of type X.
- *Axiom declarations* used to assume functions and properties of constants.
  - `axiom` `func`(x)==y; says that `func` returns y for x.
- Global *variable declarations:*
  - `var` y: X; introduces a variable y that holds an element of type X.

- A *procedure declaration* has pre- and post-conditions that specify a set of conditions about what the procedure does. The modifies is to declare that the global variable z can be modified during the execution of the procedure.

```
procedure Y(n:X);
  modifies z;
  ensures z==n;
```

- *Implementation declaration:* gives a set of execution trace of the body of the code. The *implementation* is correct if the set is a subset of the *procedure*.

```
implementation Y(n:X){
  z:=n;
}
```

Boogie Statements [2, 6, 17]

- *Assume statement:*
  - assume *E*; Given an expression, *E*, that should hold, the checker will accept without verification that *E* is true. If the expression *E* is not true the program ends without giving any error message to the user. The result of the assume statement *E* is Boolean.
- *Assert statement:*
  - assert *E*; Given an expression, *E*, that should hold, the checker will try to prove that it is true. The result of the assert statement is Boolean.

  Comparing the assume with the assert we can conclude that the assert will always warn about an error in the logic of the program, but the assume is simply considered to hold by the verifier.
- *Goto statement:*
  - goto *A*,B,C; transfers control flow to one of the labels. The choice of label is arbitrary. The list of labels (has to be of minimum length 2) in the goto must be in the programming code.
- *Break statement:*
  - break ; transfers control flow to the point immediately following the end of the code.

- *If statement:*
  - there are two kinds of `if` statement: the boolean one and the non-deterministic one.
- *While statement:*
  - Boogie supports normal `while` loops and loop invariants as well.
- *Call statement:*
  - `call` *P()*; calls another procedure in this code.

Examples

In the following section we show some *Boogie* [17] examples.

- This example demonstrates how `assert` and `assume` work. In this first piece of code it happens to be the case that the values of *x* and *y* are equal, so the `assume` is true and no errors occur.

```
procedure F() returns (r: int)
{
  var x:int;
  var y:int;
  assume(x==y);
  x:=4;
  y:=4;
  if(x==y+1) {
    assert(false);
  }
}
```

- In the second example the values *x* and *y* are different so the `assume` is not true. What one would expect would be a error, because the `if` condition is true and `assert` false is always *false*. In this case what happens is that the `assume` affect the control flow of the program, making the conditions of the `if` false that way it never enter the body of the `if`.

```
procedure F() returns (r: int)
{
  var x:int;
  var y:int;

  x:=5;
  y:=4;
  assume(x==y);
  if(x==y+1)
  {
    assert(false);
  }
}
```

- Example of an axiom. The axiom declares that the constant will not be greater than 10. It is useful because as explained in the definition 2.2.2, the axiom assigns a property to a constant that will always be true.

```
const Z: int;
axiom 10>=Z;
procedure F() returns (r: int)
{
  var n:int;
  n:=0;
  while(n<Z){

    n:=n+1;
    r:=n+5;
  }
}
```

- Example of how `modifies` work (see procedure declaration: 2.2.2).

```
/*** Example of modifies ***/
var a,b:int;
procedure F(n: int) returns (r: int)
modifies a,b;
{
  b:=0;
  if (10 < n) {
    r := n - 10;
    b:=b+3;
  }
  else {
    a:=a+1;
  }
}
```

```
/*** This will give an error because b has been modified
   but not declared in modifies. ***/

var a,b:int;
procedure F(n: int) returns (r: int)
modifies a;
{
  b:=0;
  if (10 < n) {
    r := n - 10;
  b:=b+3;
  }
  else {
    a:=a+1;
  }
}
```

```
/*** This does not give an error because the variable a was
    not changed, i.e, it does not matter what variables are
    added to the modifies (these variables must have been
   declared in the procedure) because the program will not
   complain if they are not changed in the procedure. ***/
var a,b:int;
procedure F(n: int) returns (r: int)
modifies a,b;
{
  b:=0;
  if (10 < n) {
    r := n - 10;
  b:=b+3;
  }

  }
```

- This is a simple **Boogie** example of a `while` and `if` rewritten using `goto` and `assume`. The best way to understand this transformation is to think of the control flow graph. In this example the `goto` function works like an `if` and the `assume` works like the `if` condition.

```
/*** while and if ***/
procedure F() returns (){
  var n:int;
  n:=0;
  while(n<10){
    if(n==5){
    }
    n:=n+1;
  }
}
```

```
/*** goto and assume ***/
procedure F() returns (){
  var n:int;
  n:=0;
  goto Start;

  Start:
    goto A,exit;

  A:
    assume n<10;
    goto B,C;

  B:
    assume n==5;
    goto exit;

  C:
    assume n!=5;
    n:=n+1;
    goto A, exit;

  exit:


  }
```

- This example calculates the *sum* of two values. In **Boogie** there is no `for` loop so we use `while` (it has the same pattern as the **C** `while`).

```

/***Boogie program ***/
procedure F() returns (sum:int){

  var y,x:int;
  y:=0;
  x:=0;
  while(y<25){
    while(x<80){
      sum:=x+y;
        x:=x+1;
    }
    y:=y+1;
  }
}
```

- This example is just a `while` and an `if`.

```

/***Boogie program ***/
procedure F() returns (r: int){
  var n:int;
  n:=0;
  while(n<10){
    if(n<5){
      r:=3*n;
    }
    n:=n+1;
  }
}
```

2.3 Program Analysis

2.3.1 Static Analysis

Static analysis [26] [27] is a type of analysis of software (testing and evaluation of software) that examines the code without running the software. It is widely used in type systems in programming languages and security-critical software systems.

2.3.2 Dynamic Analysis

Dynamic analysis is the opposite of static analysis, i.e, it is testing and evaluation a software during run time.

2.3.3 Static Analysis versus Dynamic Analysis [11]

Dynamic [2.3.2] and *static* [2.3.1] analysis [11] are very useful in the detection of software problems that cause memory and threading errors. The two approaches complement each other because by themselves they can not find all the errors.

The major advantage of *dynamic analysis* [2.3.2] is to reveal vulnerabilities or problems too complex to be unravelled by static analysis. Despite its goal being debugging, dynamic analysis can also play a role in ensuring safety.

The main advantage of *static analysis* [2.3.1] is to examine the values of all variables and paths of the execution, even those that were not invoked during execution. Static analysis can reveal errors that may not manifest for a long period of time. This aspect is very important in software security.

2.4 Tools - Dynamic and Static Analysis for OpenMP

2.4.1 Intel Parallel Lint

Intel Parallel Lint is part of **Intel Parallel Composer** [12] that aims to detect problems in *OpenMP* through static analysis of source code. The results of *Parallel Lint* [13] are demonstrated, additionally, in the output of the compiler.

Parallel Lint processes the source file individually producing a pseudo object module (that way it avoids destroying the real object modules).

The results in Parallel Lint are produced in the linking step allowing it to find errors in procedures and file boundaries. This means that this tool must be invoke with the compiler and not in the linker directly.

Parallel lint and OpenMP.

The tool detects the following problems related to OpenMP:

- Creation of dynamically nested parallel regions;
- Improper use of variables;
- improper use of *threadprivate* variables.

Even if it complies with all specifications, the *OpenMP* program is not always correct. In such cases the *Parallel Lint* helps diagnose the following:

- Some types of deadlock;
- Some *data races* on loop iterations;
- Some side effects when synchronization is not appropriate.

2.4.2 Intel Thread checker

Intel Thread Checker is a tool with the purpose of observing the execution of a program, giving to the user the information about possible places where a problem can occur.

The problems that can occur are mostly related with the use of threads. An example of such problems is the incorrect use of the threading and synchronization *API* because the behavior can be correct in one instance and incorrect in general.

Another problem can be the lack of synchronization and this problem leads to *data races* in most of the cases.

This tool helps the users detect where the problems are so that they can solve them easily.

2.5 Houdini

Houdini [9], [16] is a simple, but scalable tool, that validates annotations that satisfy requirements of a specific program. It receives as input a set of annotations and returns as output a consistent subset of the input.

It is configurable to allow the user to enter *invariants)* using a simple pattern (i.e the arguments are not null); it is scalable and it is transparent since users can inspect the set of annotations that were derived.

2.5.1 How it works?

The input and candidate invariants are provided to Houdini [8]. After analyzing the invariants given as input, Houdini will result in a proper subset of invariants, that is, will give as output invariant satisfying the program.

The worst case in Houdini is when the output set is empty, i.e, none of the input candidate invariants is satisfied the program; and the best case is when all of the input candidate invariants are satisfied by the program, i.e, all the candidates are invariants of the program.

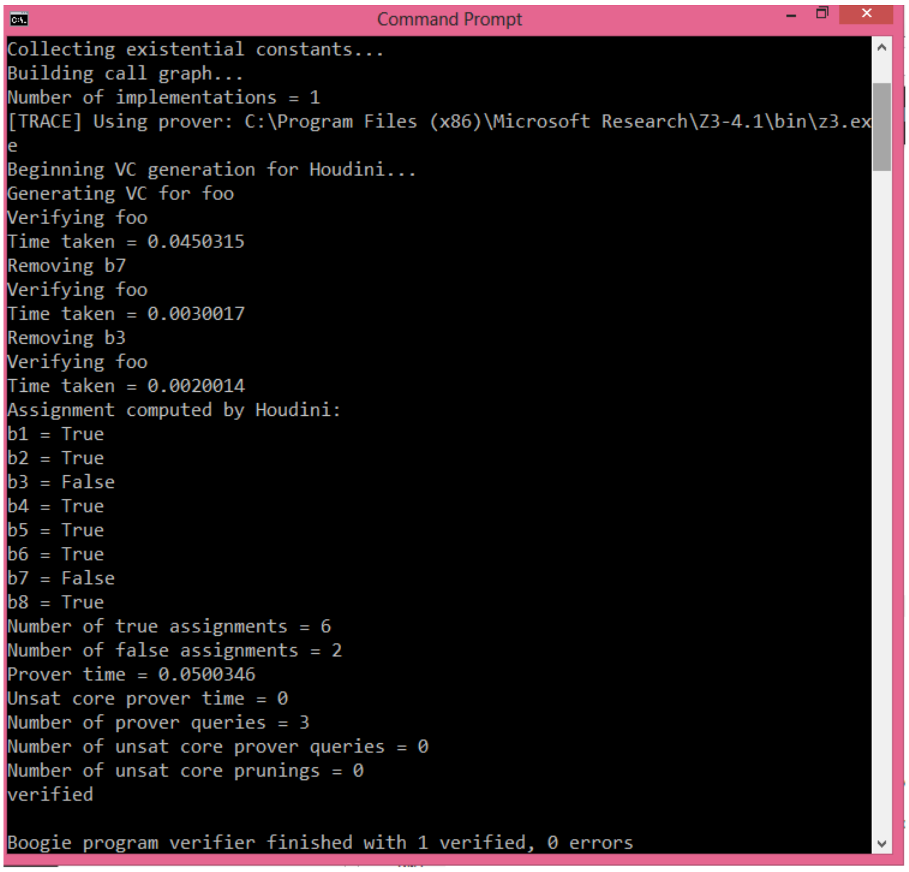
Example

This is just a simple example to demonstrate how *Houdini* works. In this case we add some *true* and *false invariants* by hand to see what would happen when we run with *Houdini*.

As it is possible to verify in the code below, eight invariants were inserted in the code which will be validated by Houdini. Each invariant is of the type *bx* ==> $expr$, and each *bx* is a constant with a special attribute: the {*:existential true*} that tells *Boogie* that this *invariants* must be treated in a special way, i.e, these invariants are used by *Houdini* for validation.

In Figure 2.2 it is possible to see that the *invariants* that are not *true* in the program have been eliminated one by one and the the program will be verified normally with the remaining *invariants*.

```
procedure foo() {

  var i, j : int;

  i := 0;
  j := 2;

  while(i < 100)
    invariant b1 ==> i <= 100;
    invariant b2 ==> i >= 0;
    invariant b3 ==> j <= 100;
    invariant b4 ==> j >= 0;
    invariant b5 ==> j >= 2;
    invariant b6 ==> j <= 202;
    invariant b7 ==> j == 2*i;
    invariant b8 ==> j == 2*i + 2;
  {
    i := i + 1;
    j := j + 2;
  }

  assert j == 2*i + 2;

}

const {:existential true} b1 : bool;
const {:existential true} b2 : bool;
const {:existential true} b3: bool;
const {:existential true} b4: bool;
const {:existential true} b5: bool;
const {:existential true} b6: bool;
const {:existential true} b7: bool;
const {:existential true} b8: bool;
```



```
Collecting existential constants...
Building call graph...
Number of implementations = 1
[TRACE] Using prover: C:\Program Files (x86)\Microsoft Research\Z3-4.1\bin\z3.ex
e
Beginning VC generation for Houdini...
Generating VC for foo
Verifying foo
Time taken = 0.0450315
Removing b7
Verifying foo
Time taken = 0.0030017
Removing b3
Verifying foo
Time taken = 0.0020014
Assignment computed by Houdini:
b1 = True
b2 = True
b3 = False
b4 = True
b5 = True
b6 = True
b7 = False
b8 = True
Number of true assignments = 6
Number of false assignments = 2
Prover time = 0.0500346
Unsat core prover time = 0
Number of prover queries = 3
Number of unsat core prover queries = 0
Number of unsat core prunings = 0
verified

Boogie program verifier finished with 1 verified, 0 errors
```

Figure 2.2: Result with Houdini

Chapter 3

Analysis

3.1 Translations of OpenMP code into Boogie code

In Chapter 2 we defined what **OpenMP** (section 2.1) and **Boogie** (section 2.2) are, showed some examples of **Boogie**. For any questions questions that may arise regarding to the syntax, simply check the sections: **OpenMP** (section 2.1.2) and **Boogie** (section 2.2.2).

This section is about the technical part of the problem, the translation of **OpenMP** programs into **Boogie** with the purpose of finding **data races** and we are going to show the translation step by step.

The first thing to do is to declare an empty *procedure*, given that these examples of *OpenMP* are based on *C* so the way we do it is equal to declaring a *function* in *C*:

```
void main() {

}
```

We translate the above to a `procedure` in *Boogie*:

```
procedure main() returns (){
}

```

After this step, we have to fill the contents of the procedure. Let us start with the **OpenMP** variable declarations:

```
void main() {
  int i;
  int j;
  int sum[20];
}
```

. It is important to note that there is no *array* type in **Boogie**.

For this reason we replace *arrays* by *maps*. The difference between the two is that *maps* do not have size limit and, hence, cannot overflow.

```
procedure main() returns (){

  var i:int;
  var j:int;
  var sum: [int] int; //initialization of a map in Boogie
}
```

Finally, we consider an *OpenMP* program with a loop :

```
void main() {
  int i;
  int j;
  int sum[20];
  j=0;

  for(i=0;i<10;i++){
    sum[i]=j;
    j++;
  }
}
```

And translate into **Boogie**

It is important to note that there is no `for` loop in *Boogie* so we use a `while` loop instead.

```
procedure main() returns (){
  var i:int;
  var j:int;
  var sum: [int] int;
  j:=0;
  i:=0;

  while(i<10){
    sum[i]:=j;
    j:=j+1;
    i:=i+1;
  }
}
```

The next step is to add parallelism to **OpenMP**, and translate that parallelism into **Boogie** somehow, since *Boogie* is a sequential language.

Before we give the details of the translation, it is important to explain how this translation is made and why it is made this way.

```
//OpenMP code
void main() {
  int i;
  int j;
  int sum[20];
  j=0;

  #pragma parallel for
  for(i=0;i<10;i++){
    sum[i]=j;
    j++;
  }
}
```

To get started it is necessary to realize that *Boogie* is a tool and language that computes sequentially and not in parallel as *OpenMP*. Being sequential, there is no reason to fear for *data races*, but in *OpenMP* it can happen. We have to somehow reflect this in *Boogie*.

It is important to keep in mind that data races occur on array elements and, hence, we need to track which accesses have occurred. Moreover as it is different loop iterations that are executed by different threads, we need to see if the accesses from different loop iterations are to the same array element.

The idea is to save locations which were accessed in the previous iterations and accesses that occur in the current iteration. This way it is possible to detect: that a past iteration accesses the same array elements as the present iteration (data race).

To save this information, we introduce two groups of variables *prev* and *cur*. Intuitively the *prev* group will help track the accesses from the *previous* iterations and the *cur* group will help track the accesses from the *current* iteration. Both of the groups consist of a `boolean` (*prev_write_sum* and *cur_write_sum* variable) and an `int` (*prev_off_write_sum* and *cur_off_write_sum*) in this specific case (in other examples the *sum* will be substituted by the name of the specifics arrays). We use the `int` to save *current* and *previous* offsets and the `boolean` to indicate if there exists a *current* and a *previous* one. In the case *prev_write_sum* is false, the value in *prev_off_write_sum* is meaningless, otherwise the value is significant. The same goes for the *cur_write_sum*, i.e., it can be concluded that the boolean variables are the most important in these groups.

Now that the variables are explained we will add them to the **Boogie** code:

```

//added variables related to the previous and the current
var prev_write_sum: bool;
var prev_off_write_sum: int;
var cur_write_sum: bool;
var cur_off_write_sum: int;

procedure main() returns ()
modifies prev_write_sum;
modifies prev_off_write_sum;
modifies cur_write_sum;
modifies cur_off_write_sum;
{
  var i:int;
  var j:int;
  var sum: [int] int;

  i:=0;
  j:=0;

  //initialize the prev_write_sum to false.
  prev_write_sum:=false;

  while(i<10){
    sum[i]:=j;
    j:=j+1;
    i:=i+1;
  }
}
```

We note that the variable *prev_write_sum* is initialized to `false` because at the beginning of the program there has not occurred any previous access. We now need to know when to update the variables. To do that we use an `if` condition, not the common one, but a non-deterministic one. Before we explain the rest of the idea it is important to explain what a *non-deterministic* `if` is. According to the definition in [7] *non-determinism* is "A property of a computation which may have more than one result", and in this specific case it means that there is no way to know if we go in the `if` or not. The reason that a *non-deterministic* `if` is used in this case is because there in no need to save all the *current* values and change the *previous* ones and because when considering all possible program executions one will track every possible assignment. The *non-deterministic* `if` are used in this way:

```
var prev_write_sum: bool;
var prev_off_write_sum: int;
var cur_write_sum: bool;
var cur_off_write_sum: int;

procedure main() returns ()
modifies prev_write_sum;
modifies prev_off_write_sum;
modifies cur_write_sum;
modifies cur_off_write_sum;
{
  var i:int;
  var j:int;
  var sum: [int] int;

  i:=0;
  j:=0;
  prev_write_sum:=false;

  while(i<10){
    cur_write_sum:=false;
    sum[i]:=j;

    //add the non-determinism if
    if(*){
      cur_write_sum:=true;
      cur_off_write_sum:=i;
    }
    j:=j+1;
    i:=i+1;

    //add the non-determinism if
    if(*){
      prev_off_write_sum:=cur_off_write_sum;
      prev_write_sum:=cur_write_sum;
    }
  }
}
```

The variable *cur* is initialized to `false` at the beginning of the iteration because for each iteration it needs to be `false` again, since we use *cur* to track the accesses from the current iteration only.

In each `if` the current or the previous variable is modified. In the first one, we set the value of *cur_write_sum* to `true` and save the position at which the array is accessed in *cur_off_write_sum*, and in the second one we just set the *previous* variables equal to the *current* variable, because at the end of the loop what was *current* will not be in the next iteration. Before the *non-deterministic* `if` we add an *assertion*. The reason why we introduce an *assertion*, is because we have to determine when the *prev_write_sum* is `true` then the *prev_off_write_sum* needs to be different from the *current*, if it is not then it means we are accessing the same position in the array as in a previous iteration and hence have a *data race*. And since we use an *assertion* it is necessary to add an *invariant* to validate the *assertion*. This is what the translation looks in the end.

```
var prev_write_sum: bool;
var prev_off_write_sum: int;
var cur_write_sum: bool;
var cur_off_write_sum: int;

procedure main() returns ()
modifies prev_write_sum;
modifies prev_off_write_sum;
modifies cur_write_sum;
modifies cur_off_write_sum;
{
  var i:int;
  var j:int;
  var sum: [int] int;

  i:=0;
  j:=0;
  prev_write_sum:=false;

  while(i<10)
    //add the invariant
    invariant(prev_write_sum==>prev_off_write_sum<i);
  {
    //Initialization of cur_off to false
    cur_write_sum:=false;

    sum[i]:=j;
    //add the assertion
    assert(prev_write_sum==>prev_off_write_sum!=i);

    //add the non-determinism if
    if(*){
      cur_write_sum:=true;
      cur_off_write_sum:=i;
    }
    j:=j+1;
    i:=i+1;

    //add the non-determinism if
    if(*){
      prev_off_write_sum:=cur_off_write_sum;
      prev_write_sum:=cur_write_sum;
    }
}
```

It is possible to optimize the translation by creating multiple *procedures*, we will use these in later examples. We create *procedures* for the *non-deterministic* `ifs` and for the `assert`. For the *main* procedure we analyze the auxiliary ones and it is necessary to insert {*:inline 1*} into each auxiliary procedure. This attribute will force *Boogie* to analyze the code of the other procedures, something he would not do naturally. This way the code is much more readable and easier to understand.

```
var prev_write_sum: bool;
var prev_off_write_sum: int;
var cur_write_sum: bool;
var cur_off_write_sum: int;

procedure main() returns ()
requires prev_off_write_sum >=0 && prev_off_write_sum<10;
requires cur_off_write_sum >=0 && cur_off_write_sum<10;
modifies prev_write_sum;
modifies prev_off_write_sum;
modifies cur_write_sum;
modifies cur_off_write_sum;
{
  var i:int;
  var j:int;
  var sum: [int] int;

  i:=0;
  j:=0;
  prev_write_sum:=false;

  while(i<10)
    //add the invariant
    invariant(prev_write_sum==>prev_off_write_sum<i);
  {
    //Initialization of cur_off_write_sum to false
    cur_write_sum:=false;
    sum[i]:=j;

    //add the assertion
    call Checker_Write_sum(i);

    //call the procedure of the non-determinist if
    //for the current
    call Log_Write_sum(i);

    j:=j+1;
    i:=i+1;

    //call the procedure of the non-determinist if
    //for the previous
    call Update_Prev_Write_sum();

  }
}

procedure {:inline 1} Checker_Write_sum(i:int)
{
  assert(prev_write_sum==>prev_off_write_sum!=i);
}

procedure {:inline 1} Log_Write_sum(i:int)
modifies cur_write_sum;
modifies cur_off_write_sum;
{
  if(*){
    cur_write_sum:=true;
    cur_off_write_sum:=i;
  }
}

procedure {:inline 1} Update_Prev_Write_sum()
modifies prev_off_write_sum;
modifies prev_write_sum;
modifies cur_off_write_sum;
modifies cur_write_sum;
{
  if(*){
    prev_off_write_sum:=cur_off_write_sum;
    prev_write_sum:=cur_write_sum;
  }
}
```

After analysing and doing a few translations it is clear that exists a procedure for all parallel for-loops (as is possible to observe in the examples) and here it is:

```
//OpenMP pseudo-code
main(){
#pragma parallel for
  for loop{
    read;
    write;
  }
}
```

```
//Boogie pseudo-code
main(){
  create variables(boolean prev and cur/
          int prev_off and cur-off)
  prev_write_sum:=false;
  prev_read_sum:=false;

  while(e)

    invariant(...);{
    cur_write_sum:=false;
    cur_read_sum:=false;

    write;
    call Checker_Write_sum(e);
    call Log_Write_sum(e);

    read;
    call Checker_Read_sum(e);
    call Log_Read_sum(e);

    increment the loop counter;

     call Update_Prev_Write_sum();
     call Update_Prev_Read_sum();
  }
}
```

The table below show us the translation used in the project, meaning that this is the translation of simple *Boogie* code into *Boogie* code with the procedure explain previously.

Stmt	Translate(Stmt)
x:=e;	x:=e;
x:=A[e];	x:=A[e];
	call Checker_Write_A(e);
	call Log_Write_A(e);
A[e]:=x;	A[e]:=x;
	call Checker_Read_A(e);
	call Log_Read_A(e);
Parallel	prev_read_A:=false;
while(e){	prev_write_A:=false;
S;	while(e){
}	cur_read_A:=false;
	cur_write_A:=false;
	translate(S);
	}
	call Update_Prev_Read_A();
	call Update_Prev_Write_A();
while(e){;	while(e){
S;	translate(e);
}	}
if(s){	if(s){;
S;	translate(S);
}	}
else{	else{
F;	translate(F);
}	}

It is important to mention that for each reading and/or writing is necessary to write the *non-deterministic* `if` condition, the `assert` and the *invariant*.

3.1.1 Examples

In the following examples it is possible to observe the transformation of simple programs (with and without data races) of **OpenMP** 2.1 code into Boogie 2.2.

It is important to recall that the *procedures* used in the first two examples occur in section 3.1.

Error Example: This example is similar to the one explained above, a simple write to an array, but with an error (the index of the array is always the same).

```
//OpenMP code
void main(int argc, char *argv[]) {
  int sum[100];
  int x = foo();
  #pragma omp parallel for
  for(i=0; i<10; i++){
    sum[0]=x;
  }
}
```

In this example we applied the procedure explained in the beginning of Section 3.1. We translate the *OpenMP* code into *Boogie* code and then we add the procedures referring to the *non-deterministic* `if` (lines 39 and 43). Each `if` is done in the same way as the ones explained above. In this case the position that we save in *cur_off_write_sum* (line 31) is *0* because that is the position accessed in the array. After we insert the `if` we add the *procedure* of the `assert` (line 36) and the `invariant` (line 15). Both of them are done using the same structure as the one explained in the beginning of Section 3.1.

```
 1 
 2 //Boogie Code
 3 var prev_write_sum: bool;
 4 var prev_off_write_sum: int;
 5 var cur_write_sum: bool;
 6 var cur_off_write_sum: int;
 7 
 8 procedure main() returns ()
 9 
10 requires prev_off_write_sum >=0 && prev_off_write_sum<10;
11 requires cur_off_write_sum >=0 && cur_off_write_sum<10;
12 modifies prev_write_sum;
13 modifies prev_off_write_sum;
14 modifies cur_write_sum;
15 modifies cur_off_write_sum;
16 {
17   var i:int;
18   var j:int;
19   var sum: [int] int;
20 
21   i:=0;
22   prev_write_sum:=false;
23 
24   while(i<10)
25   //add the invariant
26   invariant(prev_write_sum==>prev_off_write_sum==0);
```

```
  {

    //Initialization of cur_write_sum to false
    cur_write_sum:=false;
    call x:=foo();
    sum[0]:=x;

    //add the assertion
    call Checker_Write_sum(0);

    //add the non-deterministic if
    call Log_Write_sum(0);
    i:=i+1;

    //add the non-deterministic if
    call Update_Prev_Write_sum();

  }
}
```

In this example the **Boogie** verifier will fail in the `assert` because *prev_off_write_sum* will always be the same value (in this case *zero*), as it possible to observe in the *output*:

```
input(19,6): Error BP5001: This assertion might not hold
    Execution trace:
      input(11,6): anon0
      input(14,3): anon5_LoopHead
      input(17,9): anon5_LoopBody
Boogie program verifier finished with 0 verified, 1 error
```

Read Example: This example is a simple addition of some array values to a variable *k*, basically what it does is read some specific position in the array and add the value from that position to the variable *k*. In this case, no data race occurs, because as it is shown in table 2.1.3 of Section 2.1.3, just reads do not cause *data races*.

The way the translation is performed is equal to the one explained in the beginning of Section 3.1, but in this case the assertion can never fail because there are no writes but only reads, so is not necessary to put the `invariant` (line 24) and `assert` (line 29 and 37) in code.

After all that has been described, it is simple for the reader to note that in this example we also applied the procedure from 3.1 but in a more covert way (the `assert` is different from the procedure example in Section 3.1).

```
//openmp program
void main(int argc, char *argv[]) {
  int sum[11];
  int y;
  int i;
  int x=0;


  #pragma omp parallel for
  for(i=0; i<10; i++){
    x+=sum[i];
    x+=sum[i+1];
  }
}
```

```
//boogie program

var cur_read_sum:bool;
var prev_read_sum:bool;
var cur_off_read_sum:int;
var prev_off_read_sum:int;

procedure main() returns (r: int)

modifies prev_read_sum;
modifies cur_read_sum;
modifies prev_off_read_sum;
modifies cur_off_read_sum;
{
  var i:int;
  var x:int;
  var sum:[int] int;
  prev_read_sum:=false;
  i:=0;
  x:=0;
  while(i<10)
    invariant true;
  {
    cur_read_sum:=false;

    x:=x+sum[i];
    call Checker_Read_sum(i);

   //call the procedure of non-deterministic if related
   //with the current
    call Log_Read_sum(i);

    x:=x+sum[1+i];

    call Checker_Read_sum(i+1);

    //call the procedure of non-deterministic if related
    //with the current
    call Log_Read_sum(i+1);

    i:=i+1;

    //call the procedure of non-determinism if related
    //with the previous
    call  Update_Prev_Read_sum();
  }
}

procedure {:inline 1} Checker_Read_sum(i:int)
{
  assert(true);
}

procedure {:inline 1} Log_Read_sum(i:int)
modifies cur_read_sum;
modifies cur_off_read_sum;
{
  if(*){
    cur_read_sum:=true;
    cur_off_read_sum:=i;
  }
}

procedure {:inline 1} Update_Prev_Read_sum()
modifies prev_off_read_sum;
modifies prev_read_sum;
modifies cur_off_read_sum;
modifies cur_read_sum;
{
  if(*){
    prev_off_read_sum:=cur_off_read_sum;
    prev_read_sum:=cur_read_sum;
  }
}
```

Reads-Write Example: This is an example with two *reads* and a single *write*. In cases like this one it is necessary to create a set with *previous* and *current* for both *reading* and for *writing*, because a write followed by a read may cause *data races*, but a read followed by another read cannot. We call these *prev_off_read_sum*, *prev_off_write_sum*, *prev_read_sum*, *prev_write_sum*, *cur_off_read_sum*, *cur_off_write_sum* and *cur_read_sum and cur_write_sum* (lines 2-9).

In the case when we have *reading* and *writing* it is necessary to pay attention and prevent *reading* and *writing* to the same position, because a *read* and a *write* cause a *data race*. For that, what is done is the `assert` in *reading* is related with the *writing* (line 46 and line 50), and not with the *previous* read because two *reads* in a row do not cause a *data race*. The same happens in *writing*, but in this case the `assert` (lines 56-57) is related with the reading and the writing, because both two writes in a row or a *read* and a *write*, can cause *data races*.

Although it is becoming more complicated to see the procedure explained above, it is once again applied in this example but with a double dose, because a read and write must be treated differently.

It is important to remember that the *prev_read_sum* (line 27) and the *prev_write_sum* (line 28) are initialized outside the *loop* and the *cur_read_sum* (line 40) and *cur_write_sum* (line 41) inside the *loop*. This happens because we only need to save the *previous* ones and because they are going to change depending on the *current* ones.

The *current* needs to be reset because at the beginning of each iteration there is no *current* yet and we cannot assure that the program will enter in the *non-deterministic* `if` that will update the value and the state of the *currents* (line 46, 52 and 59).

In this specific case **Boogie** will verify the program because the **reads** will never have the same offsets. If one of the `assertions` would not hold then **Boogie** would not verify the program and who'd give an error message.

```
//OpenMP program
void main(int argc, char *argv[]) {
  int sum[30];
  int x;
  int k=0;
  int j=0;
  i=0;
  #pragma omp parallel for
  for(i=0; i<10; i++){
    k=sum[x+10];
    j=sum[x];
    sum[x]=k+j;
  }
}
```

```
//Boogie program
var cur_read_sum:bool;
var prev_read_sum:bool;
var cur_off_read_sum:int;
var prev_off_read_sum:int;
var cur_write_sum:bool;
var prev_write_sum:bool;
var cur_off_write_sum:int;
var prev_off_write_sum:int;

procedure F() returns (r: int)

modifies cur_read_sum;
modifies prev_read_sum;
modifies cur_off_read_sum;
modifies prev_off_read_sum;
modifies cur_write_sum;
modifies prev_write_sum;
modifies cur_off_write_sum;
modifies prev_off_write_sum;
{
  var i:int;
  var x:int;
  var y:int;
  var sum:[int] int;

  prev_read_sum:=false;
  prev_write_sum:=false;
  i:=0;
  x:=0;
  y:=0;

  while(i<10)
  invariant(prev_write_sum==>prev_off_write_sum <i);
  invariant(prev_read_sum==>prev_off_read_sum<i  ||
        prev_off_read_sum >= 10);
  {
    cur_read_sum:=false;
    cur_write_sum:=false;
    x:=sum[i+10];

    call Checker_Write_sum(i+10);

    call Log_Read_sum(i+10);

46    y:=sum[i];
47
48    call Checker_Write_sum(i);
49
50    call Log_Read_sum(i);
51
52    sum[i]:=x+y;
53
54    call Checker_Write_sum(i);
55    call Checker_Read_sum(i);
56
57    call Log_Write_sum(i);
58
59    i:=i+1;
60
61    call Update_Prev_Read_sum();
62    call Update_Prev_Write_sum();
63
64   }
65   }
```

Reads-Write with an error example: This example is similar to the one above, what differs is `Checker_Read_sum(i+1)` (line 33) that will not hold because the value that is stored in the *prev_off_read_sum* is *i+2* (line 25). As explained above in Section 3.1 the `if` are *non deterministic* so there is the possibility of the *i+1* (line 30) and *i+2* (line 25) being equal which will cause the `assert` to fail.

The code below shows only the differences with the previous code.

```
//OpenMP code
int main(int argc, char *argv[]) {
int sum[20];
int y;
int x;
int k=0;
int j=0;

#pragma omp parallel for
for(x=0; x<10; x++){
  j=sum[0];
  k=sum[x+2];
  sum[x+1]=j+k;}
}
```

```
//Boogie Program

  while(i<10)
    invariant(prev_write_sum ==>prev_off_write_sum<i+1);
    invariant(prev_read_sum ==>prev_off_read_sum <i+1 ||
          prev_off_read_sum >=2);
  {
    cur_read_sum:=false;
    cur_write_sum:=false;
    x:=sum[0];

    //call the assert procedure for the write
    call Checker_Write_sum(0);

    // read non-deterministic if
    call Log_Read_sum(0);

    y:=sum[i+2];

    //call the assert procedure for the write
    call Checker_Write_sum(i+2);

     // read non-deterministic if
    call Log_Read_sum(i+2);

    sum[i+1]:=x+y;

    //call the assert procedure for the write
    call Checker_Write_sum(i+1);

    //call the assert procedure for the read
    call Checker_Read_sum(i+1);

    // write non-deterministic if
    call Log_Write_sum(i+1);

    i:=i+1;

    // read non-deterministic if
    call Update_Prev_Read_sum();

    //write non-deterministic if
    call Update_Prev_Write_sum();
  }
```

Two whiles with one parallel for example: This example has two whiles with a `parallel for` in the inner-most `while` (line 25). This is just a *write* in the *array* so we will need just one block of variables *prev_write_sum, cur_write_sum, prev_off_write_sum and cur_off_write_sum* (lines 3-6). The tricky part is the `invariant` (lines 22-24 and 26-28) that is a bit different because of the position that we access in the *array* (line 30) is a combination of variables so it is necessary to use division and modulo. The procedure is easy to discover, because although this example has two whiles, only one of them is parallel so we just apply the *pattern* to the inner most `while` (lines 25-39).

In general it is just as if there was only one `while` because the first will not need much attention from the programmer, and that is why it becomes a simpler example to translate than the previous two.

```
 1 //OpenMP program
 2 void main(int argc, char *argv[]) {
 3   int sum[100];
 4   int y;
 5   int x;
 6   int j=4;
 7 
 8   for(y=0; y<4; y++){
 9     #pragma omp parallel for
10     for(x=0; x<4; x++)
11     {
12       sum[y+x*j]=x+y;
13     }
14   }
15 }
```

```
 1 //Boogie Program
 2 
 3 var cur_write_sum:bool;
 4 var prev_write_sum:bool;
 5 var cur_off_write_sum:int;
 6 var prev_off_write_sum:int;
 7 
 8 procedure main() returns ()
 9 
10 modifies prev_write_sum;
11 modifies cur_write_sum;
12 modifies prev_off_write_sum;
13 modifies cur_off_write_sum;
14 {
15   var y,x,j:int;
16   var sum:[int]int;
17    j:=4;
18    prev_write_sum:=false;
19    y:=0;
20    x:=0;
21    while(y<4)
22      invariant(prev_write_sum==>
23                  prev_off_write_sum div j < x);
24      invariant(prev_write_sum==>
25                  prev_off_write_sum mod j < y);
26    {
27      while(x<4)
28        invariant(prev_write_sum==>
29                    prev_off_write_sum div j < x);
30        invariant(prev_write_sum==>
31                    prev_off_write_sum mod j <= y);
32     {
33          cur_write_sum:=false;
34          sum[y+x*j]:=x+y;
35          //assert
36          call Checker_Write_sum(y+x*j);
37
38          //current non-deterministic if
39          call Log_Write_sum(y+x*j);
40
41          //previous non-deterministic if
42          call Update_Write_sum();
43          x:=x+1;
44      }
45      y:=y+1;
46    }
47 }
```

Two parallel While nested loops example: In this example we present two nested `while` loops and each `while` loop (line 30 and line 38) is parallel. The main idea is similar to the previous one but in this case we have to duplicate the procedure.

In the case where there are several parallel loops there is a need to differentiate loops. In this specific example there are only two nested loops, and we call it the inner and the outer. Each loop has 4 variables associated with it (`prev, cur, prev_off and cur_off`) (lines 2-9) with the inner and outer suffixes. The `invariants` (lines 31-24 and lines 39-42) and the `assert` (lines 48-49) are similar to the previous example, we just duplicate them because now we have inner and outer variables.

To sum up, when adding a loop in the examples, as well when adding a read and/or write, we also need to apply the procedure explained in the beginning of the Chapter 3.

```
//OpenMP code

void main(int argc, char *argv[]) {
  int sum[100];
  int y;
  int x;
  int j=4;

  #pragma omp parallel for
  for(y=0; y<4; y++){
    #pragma omp parallel for
    for(x=0; x<4; x++){
      sum[y+x*j]=x+y;
    }
  }
}
```

```
//Boogie Program
var cur_inner:bool;
var prev_inner:bool;
var cur_off_inner:int;
var prev_off_inner:int;
var cur_outer:bool;
var prev_outer:bool;
var cur_off_outer:int;
var prev_off_outer:int;

procedure main() returns (sum:[int]int)
modifies cur_inner;
modifies prev_inner;
modifies cur_outer;
modifies prev_outer;
modifies cur_off_inner;
modifies prev_off_inner;
modifies prev_off_outer;
modifies cur_off_outer;
{

  var y,x,j:int;
  j:=4;
  y:=0;
  x:=0;

  prev_inner:=false;
  prev_outer:=false;

    while(y<4)
      invariant(prev_inner==>(prev_off_inner div j)<x);
      invariant(prev_inner==>(prev_off_inner mod j)<y);
      invariant(prev_outer==>(prev_off_outer mod j)<y);
      invariant(prev_outer==>(prev_off_outer div j)<x);
   {
        cur_outer:=false;
        prev_inner:=false;
        while(x<4)
        invariant(prev_inner==>(prev_off_inner div j)<x);
        invariant(prev_inner==>(prev_off_inner mod j)<=y);
        invariant(prev_outer==>(prev_off_outer mod j)<=y);
        invariant(prev_outer==>(prev_off_outer div j)<x);
    {
          cur_inner:=false;
          sum[y+x*j]:=x+y;

          //assert for the inner and then outer loop
          call Checker_inner(y+x*j);
          call Checker_outer(y+x*j);

          //current non-deterministic if for the inner loop
          call Log_Current_inner(y+x*j);

          x:=x+1;

        //previous non-deterministic if for the inner loop
          call Log_Previous_inner ();
      }

      //current non-deterministic if for the outer loop
      call Log_Current_outer(prev_off_inner,prev_inner);

      y:=y+1;

      //previous non-deterministic if for the outer loop
      call Log_Previous_outer ();
    }
}

//assert for the inner loop
procedure {:inline 1} Checker_inner(i:int)
modifies prev_inner;
modifies prev_off_inner;
```

```
{
  assert(prev_inner==>prev_off_inner!=i);
}

//assert for the outer loop
procedure {:inline 1} Checker_outer(i:int)
modifies prev_outer;
modifies prev_off_outer;
{
  assert(prev_outer==>prev_off_outer!=i);
}

//current non-deterministic if for the inner loop
procedure {:inline 1} Log_Current_inner(i:int)
modifies cur_inner;
modifies cur_off_inner;
{

  if(*){
    cur_inner:=true;
    cur_off_inner:=i;
  }
}

//current non-deterministic if for the outer loop
procedure {:inline 1} Log_Current_outer(i:int, prev:bool)
modifies cur_outer;
modifies cur_off_outer;
{

    if(*){
    cur_outer:=prev;
    cur_off_outer:=i;
  }

}

//previous non-deterministic if for the inner loop
procedure {:inline 1} Log_Previous_inner()
modifies prev_off_inner;
modifies prev_inner;
modifies cur_off_inner;
modifies cur_inner;
{

  if(*){
    prev_off_inner:=cur_off_inner;
    prev_inner:=cur_inner;
  }
}

//previous non-deterministic if for the outer loop
procedure {:inline 1} Log_Previous_outer()
modifies prev_off_outer;
modifies prev_outer;
modifies cur_off_outer;
modifies cur_outer;
{

  if(*){
    prev_off_outer:=cur_off_outer;
    prev_outer:=cur_outer;
  }
}
```

Chapter 4

Implementation

As already mentioned previously the tool is divided into two parts: one is the translation of *OpenMP* to *Boogie* (which was not implemented, we only did it manually) and the other part is the transformation of the *Boogie* code into an extended *Boogie* code that checks if exist *data races* or not (this part is the tool that will be explained in this chapter).

In this chapter we will explain how the tool was implemented (we will explain what each method does), the structure used in the code, the implementation restrictions and how we obtain the invariant.

4.1 How it works

The implemented tool aims to translate simple *Boogie* derived from *OpenMP* code into *Boogie* code such that is possible to verify whether *data races* exist or not.

The *Boogie* code input is just a translation of *OpenMP* to *Boogie* (this translation is made by hand) and the output is that translate *Boogie* code with the procedure that the tool applies to the code to check for *data races*.

As you would expect the tool has some limitations and problems that will be discussed in the *Limitations and Problems* (Section 4.4.2).

After explaining how and what the tool does (4.2), it is important to explain the structure used for the construction of this tool.

First, the tool was built based on the *Boogie* code (the source code) that uses the *Structured statements* code representation. It was on this basis that we decided to use the same type of representation of code that will be explained below (using the following examples).

Figure 4.1 is a simple `while` and `if` code example. In order to understand how the structure works we will use as a basis the example of Figure 4.1. As it is possible to observe, the example has some letters associated with each line of code. To know what these letters represent we will give a brief explanation of how a Structured Statement works (Figure 4.2).

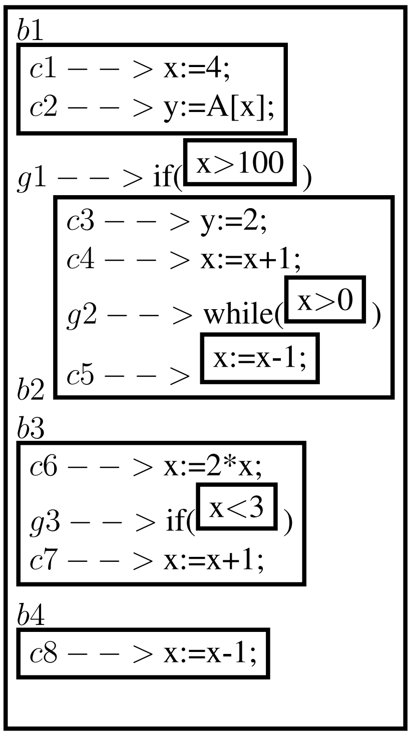


Figure 4.1: While and If Example

An *Implementation* that has *Structured Statement* contains *BigBlocks* (one *BigBlock* and a list of *BigBlocks*) which are blocks of code that contains *Commands* (we can see in Figures 4.1 and 4.2 the *BigBlocks*).

The *Commands* can be of *SimpleCommands* type and *CompondCommand* type. The *SimpleCommands* are simple commands, such as arrays reads/writes, initialization of variables and assigning values to the same, increment and decrement of variables, basically it is everything except the *commands* that use the program language reverse words. *Commands* with the reversed words, like `while`,`if`,`goto`,`invariant`, etc., are called *CompondCommands*.

In the *CompondCommands* we have `while`,`if`,`goto`,`invariant`, etc., and each of these commands can be *SimpleCommands* and *CompondCommands*. An example of this commands is the *WhileCmd*, which is shown in the Figures 4.1 and 4.2, these command has a *Guard* that is an expression and *Body* which is a *BigBlock*. Another example is the *IfCmd* that has a *Guard* which is again an expression, a *ElseCmd* that is a *BigBlock* and a *ThenCmd* which is also a *BigBlock*.

Now that we explained how it works, it is important to give a brief explanation about the Figures 4.1 and 4.2.

In this particular example (Figures 4.1 and 4.2) we have five *BigBlocks* (represented by *b1-b5*) that are *SimpleCmds*, *IfCmds* and *WhileCmds*. The *SimpleCmds* are represented by *c1-c8* being a total of 8 in this example; the *Guard* of the *WhileCmd* and *IfCmd* are represented in the example by *g1-g3*.

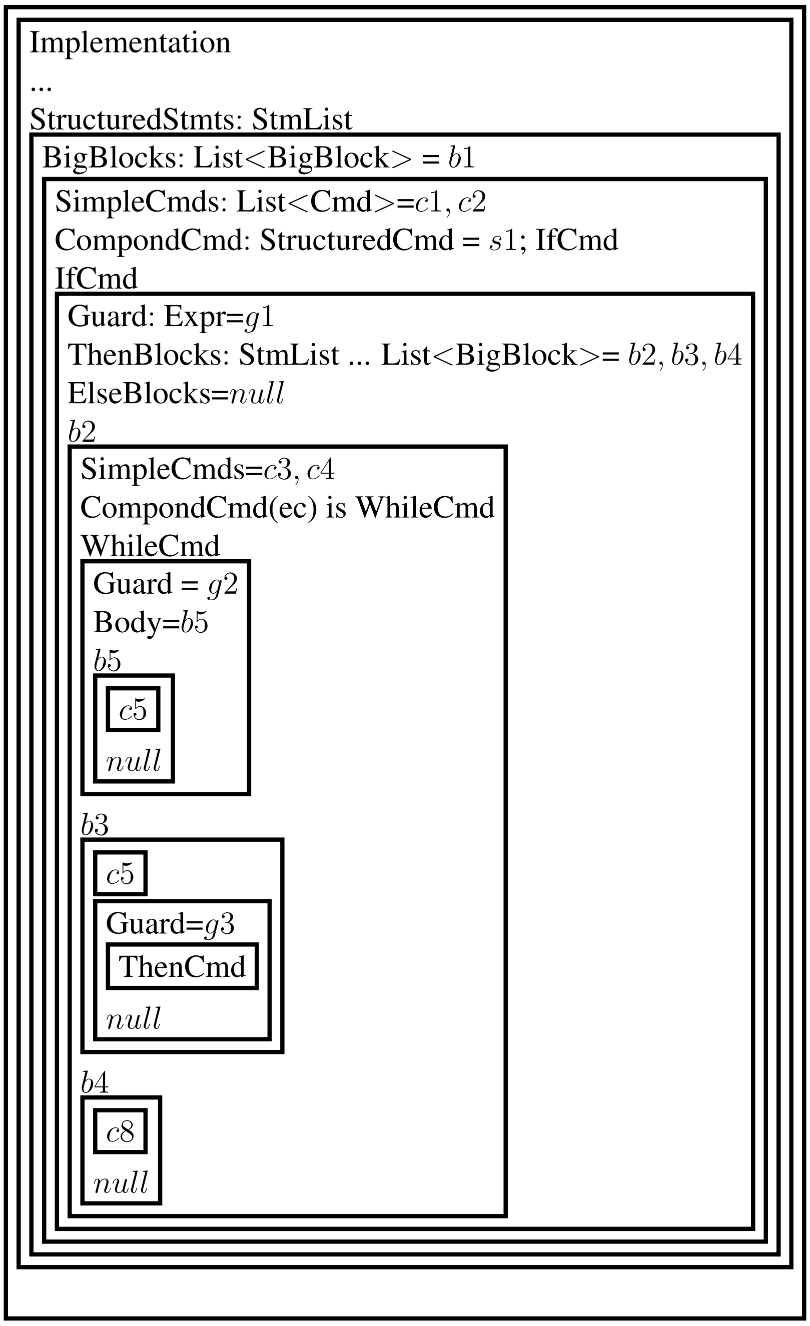


Figure 4.2: Structured Statement of Fig:4.1

In summary, it was based on this code Structure that the *data race* checker was implemented (Chapter 4.2).

4.2 Methods

The *SVDR* has five classes: *Program* that is the main class, *Add* is the class that has all the add methods, *Read_Write* has all the methods related to reads and writes and the finally class *Auxiliar* has the the remaining classes.

Thirteen principal methods were added to the classes explained above : *Principal*, *nameMethods*, *addUpdate*, *addLog*, *initializeVariable*, *addChecker*, *AddMethod*, *Read_Methods*, *Update_rw*, *Write_Methods*, *isParallel*, *Update_Previous*, *addInvariant*.

Figure 4.3 shows the relation between the classes and methods.

The **Principal** is the method where other methods are called, where the global variables are created and initialized and where the output is print to the screen.

The **nameMethods** saves the name of the arrays present in the code (just the parallel ones), then we use these names in the creation of methods (checker, log, update) and variables (current and previous). It has as argument an Implementation, which in this case will be the Implementation of the input file of the program.

The **addUpdate** creates the method that updates the previous variable in a non-deterministic way. As arguments it takes: a List of Declaration that is used to add the update method to the output; a string with the name of the method; and four global variables (prev, prev_off, cur, cur_off).

The **addChecker** creates the method that generates the assertion. As arguments this method has a List of Declaration that is used to add the assertion into the output, the name of the method and two global variables (prev and prev_off).

The **addLog** creates the method that initialize and update the current values. As arguments this method has a List of Declaration that is used to add the assertion into the output, the name of the method and two global variables (prev and prev_off).

The **AddMethods** inserts the call of the methods in the right position in the code. Basically what it does is analyze the code that is given as input, and place the calls to the methods in the right positions.

The **initializeVariables** is a method that initialize the *current* variables in the right position in the code.

The **isParallel** is a method that returns true if the the *While* is parallel and false otherwise. The way that it checks if is parallel or not is buy checking the *invariant* with the {*:parallelLoop*} attribute. That attribute is add in the input code buy the user to identify that the specific *While* is parallel.

The **Read_Methods** is a method that create the read *Call* methods and put them in the right position in the output.

The **Write_Methods** is a method that create the write *Call* methods and put them in the right position in the input code.

The **Update_rw** is a method that create the read/write update *Call* methods.

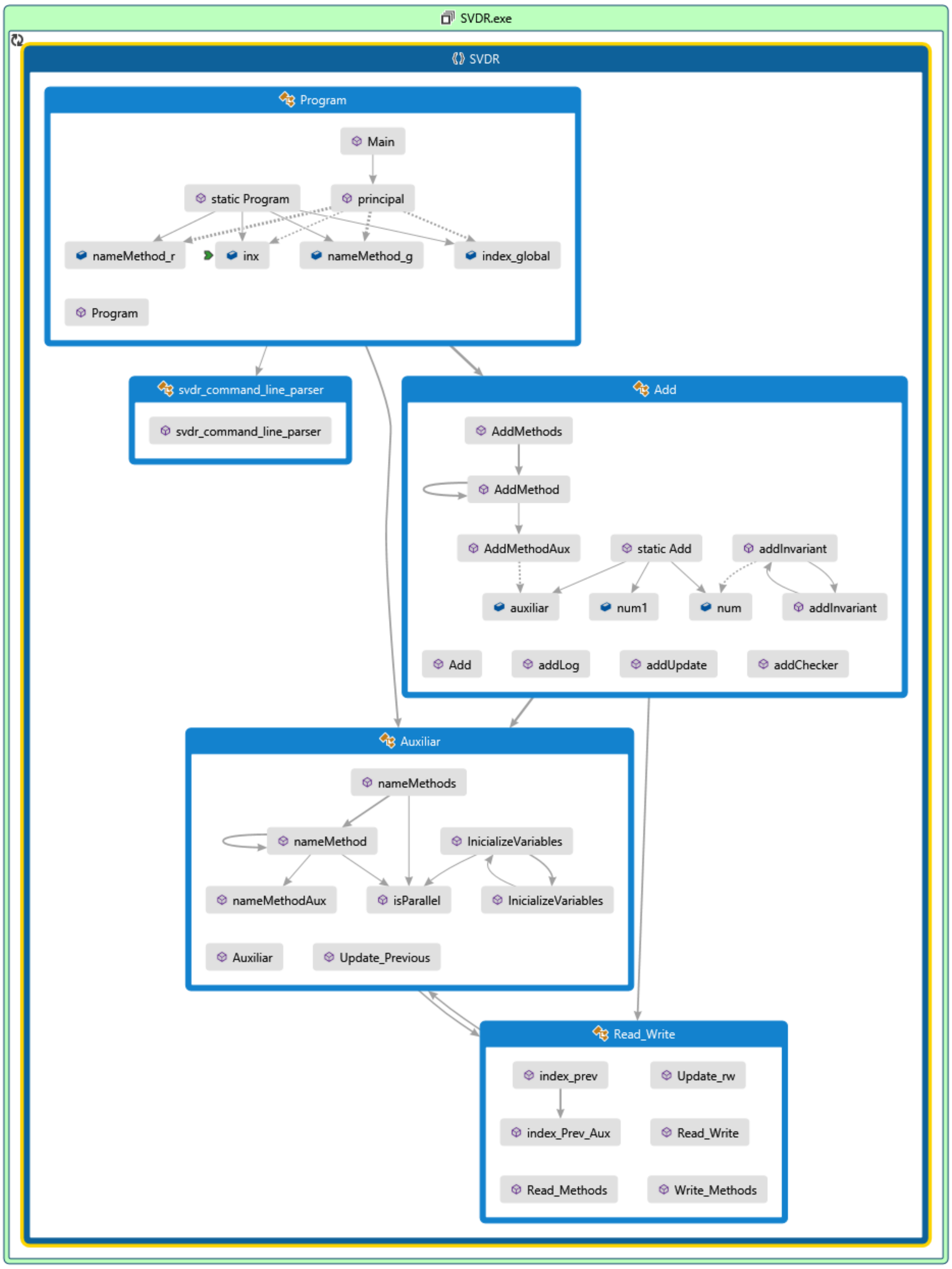


Figure 4.3: Class Diagram

The **Update_Previous** puts the Update methods in the right position in the output code.

The **addInvariant** is a method that creates the different invariants and associates them with Boolean constants. We do it that way so that *Houdini* can analyse them and delete those that do not apply to the input file.

4.3 Invariant Generation

The *Invariants* were the last thing that we took care of in the implementation of the tool.

We decide to used *Houdini* (Section 2.5) so that it was possible to eliminate the *false invariants* leaving the *true* ones to be used by the *Boogie Verifier*. We create a method that add the *invariants* in the input and creates the *Booleans* that are associated with them. This *Boolean* are the ones that return *false* or *true* as *Houdini* checks whether they hold or not.

It is important to explain that the *invariants*, in this case, are generated in the code. When the first translations were carried out (Chapter 2.2) it was possible to notice that in some of the translations the *invariants* was following the same procedure. These procedures are the ones used to create the *invariants* in our tool.

Despite working for the tested cases, these *invariants* are not valid in all examples.

The best that can happen is that all the *invariants* are *true* which is great, and the worst case is when none of the *invariants* hold, with means that it is not possible to verify the program.

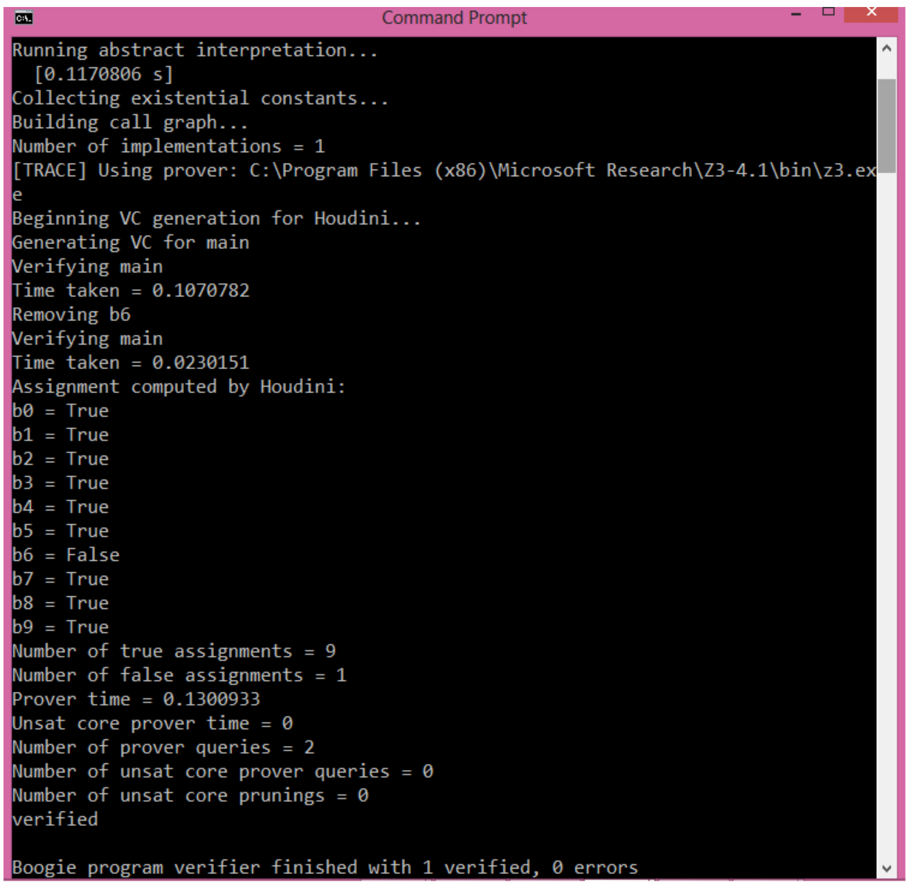
To illustrate what was been explained we are going to show an example of the tool working with *invariants* and the result of that in the *Boogie Verifier*.

In the example below we can check that 9 *invariants* are created, because the first loop is the *parallel* loop so any invariant associated with this loop has to be replicated in the other loops for the program to pass the *Boogie Verifier*.

```

//Boogie program verifier version 2.2.30705.1126, Copyright
    (c) 2003-2014, Microsoft.
var prev_read_sum: bool;

var cur_read_sum: bool;

var prev_off_read_sum: int;

var cur_off_read_sum: int;

procedure main() returns (r: int);
```

```
  modifies prev_off_read_sum, prev_read_sum,
   cur_off_read_sum, cur_read_sum;



implementation main() returns (r: int)
{
  var i: int;
  var x: int;
  var y: int;
  var b: int;
  var sum: [int]int;

    i := 0;
    prev_read_sum := false;
    x := 0;
    y := 0;
    b := 4;
    while (y < 4)
      invariant b0 ==> prev_read_sum ==> prev_off_read_sum
   < i + y * b;
      invariant b1 ==> prev_read_sum ==> prev_off_read_sum
   div  b <  y ;
      invariant b2 ==> prev_read_sum ==> prev_off_read_sum
   mod  b < i ;
      invariant b3 ==> prev_read_sum ==> prev_off_read_sum
   mod  b <= i ;
      invariant b4 ==> (prev_read_sum ==> prev_off_read_sum
    < i + y * b) || prev_off_read_sum >= 0;
    {
        while (i < 4)
          invariant {:parallelLoop} true;
          invariant b5 ==> prev_read_sum ==>
   prev_off_read_sum < i + y * b;
          invariant b6 ==> prev_read_sum ==>
   prev_off_read_sum div  b <  y ;
          invariant b7 ==> prev_read_sum ==>
   prev_off_read_sum mod  b < i ;
          invariant b8 ==> prev_read_sum ==>
   prev_off_read_sum mod  b <= i ;
          invariant b9 ==> (prev_read_sum ==>
   prev_off_read_sum < i + y * b) || prev_off_read_sum >=
   0;
        {
            cur_read_sum := false;
            x := sum[i + y * b];
            call Checker_Read_sum(i + y * b);
            call Log_Read_sum(i + y * b);
            i := i + 1;
            call Update_Prev_Read_sum();
        }

        y := y + 1;
    }
}

const {:existential true} b0: bool;

const {:existential true} b1: bool;

const {:existential true} b2: bool;

const {:existential true} b3: bool;

const {:existential true} b4: bool;

const {:existential true} b5: bool;

const {:existential true} b6: bool;

const {:existential true} b7: bool;

const {:existential true} b8: bool;

const {:existential true} b9: bool;
```

As it is possible to check in Figure 4.4 all the created *invariants* and the program is verified by *Boogie*, which means that there are no *data races* in this program. Some of there *invariants* will be eliminated by *Houdini* because they are not valid.



```
Running abstract interpretation...
  [0.1170806 s]
Collecting existential constants...
Building call graph...
Number of implementations = 1
[TRACE] Using prover: C:\Program Files (x86)\Microsoft Research\Z3-4.1\bin\z3.ex
e
Beginning VC generation for Houdini...
Generating VC for main
Verifying main
Time taken = 0.1070782
Removing b6
Verifying main
Time taken = 0.0230151
Assignment computed by Houdini:
b0 = True
b1 = True
b2 = True
b3 = True
b4 = True
b5 = True
b6 = False
b7 = True
b8 = True
b9 = True
Number of true assignments = 9
Number of false assignments = 1
Prover time = 0.1300933
Unsat core prover time = 0
Number of prover queries = 2
Number of unsat core prover queries = 0
Number of unsat core prunings = 0
verified

Boogie program verifier finished with 1 verified, 0 errors
```

Figure 4.4: Boogie Verification

4.4 Limitations and Problems

In this section we are going to show and explain the limitations and problems of our tool, and explain how we could solve some of these limitations and problems.

4.4.1 Limitations

During the entire implementation process of the tool we have come across situations where we would have to treat them or to assume that it would not be accepted by the tool, therefore the tool has some limitations.

We start with a simple one which is the fact that we do not allow two or more *parallel nested loops*. If the user enters an input with two or more *parallel nested loops* they will receive a *error message* as output saying that *"This operation is not possible."* The reason we did not take care of *parallel nested loops* was the fact that is complex and is not something that occurs in a lot of examples.

Another limitation on the *input* is the fact that it is not acceptable to have a complex expression with an array element (Example: *x: = x + sum [y];*), so in order to use a complex expression as mention above we must decompose as follows *k: = sum [y]; x: = x + k;*.

This limitation can be easily fixed by creating a method that translated this *x: = x + sum [y];* into *k: = sum [y]; x: = x + k;*.

The last limitation is related to the *invariants*. Despite being generated on the tool the procedure of them is written by us which means these exist some examples in which none of that *invariant* will be useful in the verification of the input which means that this input may no have *data race* but still will not be accepted by our tool.

These are the most glaring limitations that must be taken into account when using the tool for the result to be as smooth and satisfactory as possible.

4.4.2 Problems

One problem that has emerged during the tests of the code, was the use of the *unique* flag in *constants* in *Boogie*.

The flag *unique* states that each *constant* is different, i.e., there are not two equal *constants*. This flag is very useful when we want to create constants such as days of the week or months of the year because these values will always be different from each other.

Despite being very useful this flag is not much referred in the manual, although when we create a *constant* in *Boogie* it is set by default to *true*, so it is assumed that all *constants* are different from each other.

In our case what happened was that our *constants* are *Boolean*, so there are two possible values, *true* or *false*. When we create more than two *constants* what will happen is the equivalent of using an *assume false*, i.e., everything is possible, which is what happened with our code leading to all the tests failing to detect the errors and then all inputs were verified, even the ones that were wrong.

Fortunately the error was detected and corrected in time otherwise it would be a mistake with plenty of negative repercussions.

Chapter 5

Tests

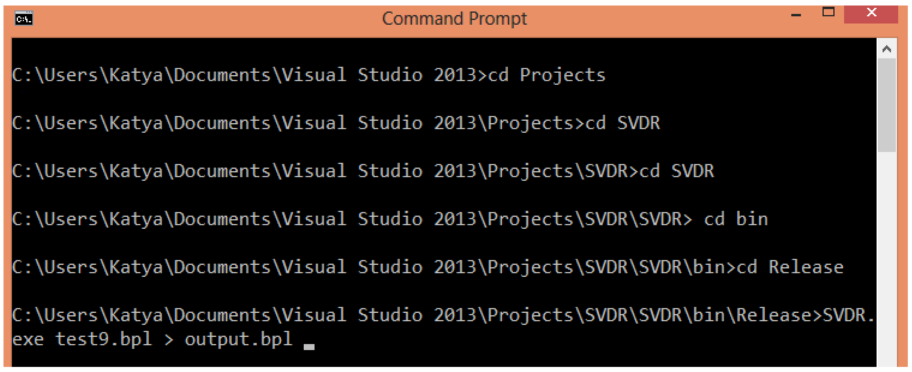
In this chapter, we will focus on different types of tests used to evaluate the tool and what they contributed to find problems and errors.

5.1 How to execute the Tests?

The tests are executed in two different phases: the first phase is just running the program with a test so that the procedure provided by the tool is added to the input, i.e, the first test is using the developed tool. The second test is to run the *Boogie* verifier using the result of the first test to see if there really exists or not a *data race* (if the result gives an error then the input has a *data race*, otherwise the input does not have a *data race*).

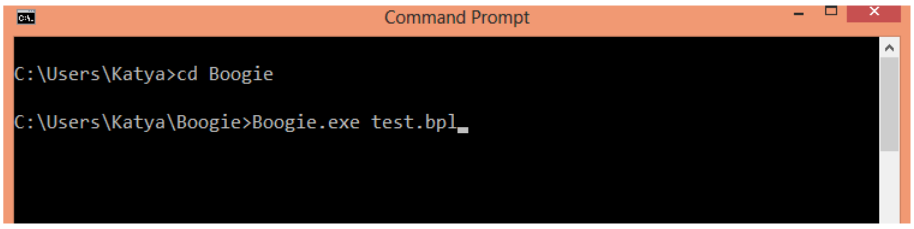
To run the first test phase, as shown in Figure 5.1, one must first download the package containing the executable, then open the command line. At the prompt we need to choose the directory where the executable is (using the *cd* command - change directory) and then just run the executable with the test (as seen in the last row of Figure 5.1).

Now, to run the second phase of the tests, as shown in Figure 5.2, it is necessary to have installed the *Boogie Verifier* (it is available for download in the *Microsoft* web page). After that we need to open the command line, choose the directory that has the *Boogie*



```
C:\Users\Katya\Documents\Visual Studio 2013>cd Projects

C:\Users\Katya\Documents\Visual Studio 2013\Projects>cd SVDR

C:\Users\Katya\Documents\Visual Studio 2013\Projects\SVDR>cd SVDR

C:\Users\Katya\Documents\Visual Studio 2013\Projects\SVDR\SVDR> cd bin

C:\Users\Katya\Documents\Visual Studio 2013\Projects\SVDR\SVDR\bin>cd Release

C:\Users\Katya\Documents\Visual Studio 2013\Projects\SVDR\SVDR\bin\Release>SVDR.
exe test9.bpl > output.bpl
```

Figure 5.1: How to run the first phase



```
C:\Users\Katya>cd Boogie

C:\Users\Katya\Boogie>Boogie.exe test.bpl_
```

Figure 5.2: How to run the second phase

executable and runs using the output of the first phase of testing. If the result gives an error then we have a data race, otherwise there is no data race in the file.

5.2 Tool Tests

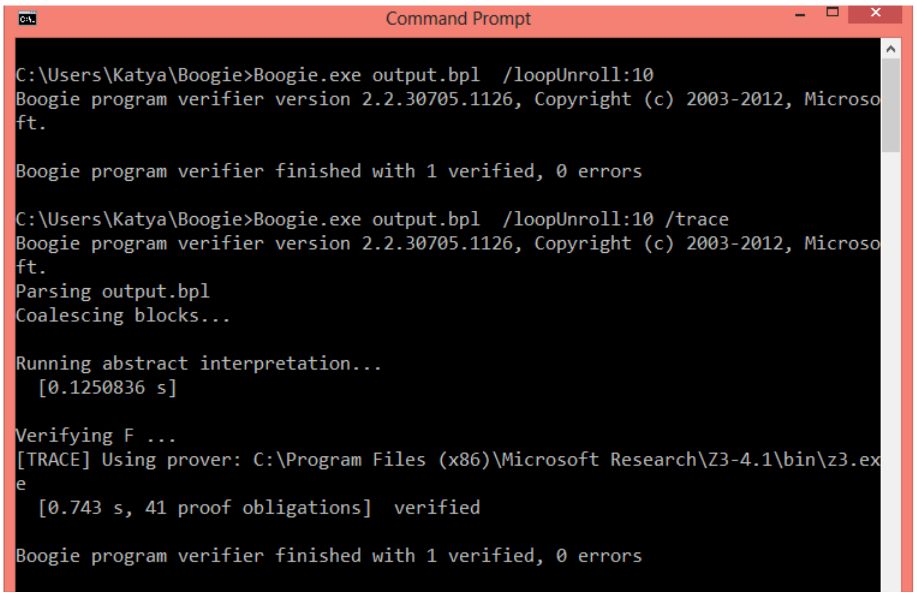
The first phase of the tests is the one that confirms if the tool does what it is supposed to do.

The tests used were the same as those shown in *Chapter* 3. We decided to do it that way because it was easy to compare if everything was alright.

We tested all the examples in *Chapter* 3 and here is some example of the result of a test using the tool:

```
var prev_write_sum: bool;

var cur_write_sum: bool;

var prev_off_write_sum: int;

var cur_off_write_sum: int;

procedure main();
  modifies prev_off_write_sum, prev_write_sum,
   cur_off_write_sum, cur_write_sum;
implementation main()
{
  var i: int;
  var j: int;
  var sum: [int]int;

    prev_write_sum := false;
    i := 0;
    j := 0;
    while (i < 10)
    invariant {:parallelLoop} true;
    {
        cur_write_sum:=false;
        sum[0] := j;
        call Checker_Write_sum(0);
        call Log_Write_sum(0);
        j := j + 1;
        i := i + 1;
        call Update_Write_sum();
    }
}



procedure {:inline 1} Checker_Write_sum(i: int);

```

```


implementation {:inline 1} Checker_Write_sum(i: int)
{
  bigblock:
    assert prev_write_sum ==> prev_off_write_sum != i;
}



procedure {:inline 1} Log_Write_sum(i: int);
  modifies cur_write_sum, cur_off_write_sum;



implementation {:inline 1} Log_Write_sum(i: int)
{
  bigblock:
    if (*)
    {
      bigblock1:
        cur_write_sum := true;
        cur_off_write_sum := i;
    }
}



procedure {:inline 1} Update_Prev_Write_sum();
  modifies cur_write_sum, cur_off_write_sum, prev_write_sum
   , prev_off_write_sum;




implementation {:inline 1} Update_Prev_Write_sum()
{
  bigblock:
    if (*)
    {
      bigblock1:
        prev_write_sum := cur_write_sum;
        prev_off_write_sum := cur_off_write_sum;
    }
}
```



Command Prompt

```
C:\Users\Katya\Boogie>Boogie.exe output.bpl  /loopUnroll:10
Boogie program verifier version 2.2.30705.1126, Copyright (c) 2003-2012, Microso
ft.

Boogie program verifier finished with 1 verified, 0 errors

C:\Users\Katya\Boogie>Boogie.exe output.bpl  /loopUnroll:10 /trace
Boogie program verifier version 2.2.30705.1126, Copyright (c) 2003-2012, Microso
ft.
Parsing output.bpl
Coalescing blocks...

Running abstract interpretation...
  [0.1250836 s]

Verifying F ...
[TRACE] Using prover: C:\Program Files (x86)\Microsoft Research\Z3-4.1\bin\z3.ex
e
  [0.743 s, 41 proof obligations]  verified

Boogie program verifier finished with 1 verified, 0 errors
```

Figure 5.3: Run loopUnroll

If we compare the example above with the one in *Chapter* 3 (example 1 in Section 3.1.1), it is possible to realize that the tool gives the same result as the example, which means that it is correct. The only difference between the two examples is the *invariant parallelLoop true;*. We have to introduce this *invariant* to identify which *while loop* was parallel.

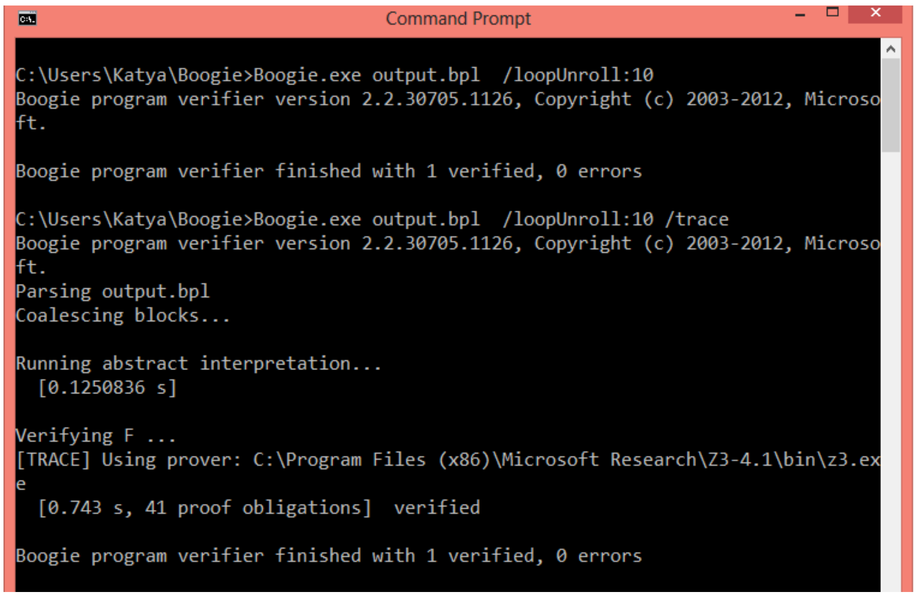
5.3 Verification Test

The following tests were executed to verify if the output produced by the tool passes the Boogie verifier or not (in some cases it will not pass because a real data race exists).

5.3.1 Verification Tests using *loopUnroll*

The tests that we did using the *loopUnroll* where done in a stage of the project were the *invariant* was not implemented. Its main objective was to verify if the code input would have *data races* or not. Basically what the *loopUnroll* does is to unroll the body of the *loop* X times and by doing that we can verify if the output is correct or not and if the *Boogie* verifier checks if the *output* verifies or not.

To test using the *loopUnroll* we need to follow the same procedures as in Figure 5.2, i.e., to have installed the *Boogie Verifier* (it is available for download in the *Microsoft* web page), then open the command line, choose the directory that has the *Boogie* executable and run it using the output of the first phase of testing and add *loopUnroll: X* as an attribute (X is the number of times that we want it to execute); see Figure 5.3.



```
C:\Users\Katya\Boogie>Boogie.exe output.bpl  /loopUnroll:10
Boogie program verifier version 2.2.30705.1126, Copyright (c) 2003-2012, Microso
ft.

Boogie program verifier finished with 1 verified, 0 errors

C:\Users\Katya\Boogie>Boogie.exe output.bpl  /loopUnroll:10 /trace
Boogie program verifier version 2.2.30705.1126, Copyright (c) 2003-2012, Microso
ft.
Parsing output.bpl
Coalescing blocks...

Running abstract interpretation...
  [0.1250836 s]

Verifying F ...
[TRACE] Using prover: C:\Program Files (x86)\Microsoft Research\Z3-4.1\bin\z3.ex
e
  [0.743 s, 41 proof obligations]  verified

Boogie program verifier finished with 1 verified, 0 errors
```

Figure 5.4: Run the code with invariant

5.3.2 Tests with invariants

An important part in the project are the generated invariants and for this reason they have been extensively tested and have prove to be very accurate.

Invariants introduced by hand

At one point of the project, in order to test whether the rest of the implementation is correct it was necessary to introduce *invariants*. Since that had not yet been deal with in the code, what was done was enter them manually in the input of the *Boogie Verifier*.

That way it was possible to detect errors in auxiliary classes, the positions of the variables, etc.

Later in the project we implemented the *invariants* automatically, as explained below.

Invariant method

To test the invariants it was necessary to add some attributes to the method used in Figure 5.2, as can be seen in Figure 5.4.

Basically the process is the same, run the executable *Boogie* with the input file, and in this case it is necessary to add one more attribute (*contractInfer*) that is used for the *Houdini* check of the invariant, then we can add two more attributes (*trace* and *printAssignment*) just to print the steps of the evaluation of *Houdini* on the screen.

After *Houdini* establish if the *invariants* are correct for the example, *Boogie Verifier* checks if the input holds or not.

Chapter 6

Results

In this chapter we are going to show an example, to explain how we did the implementation of the *SVDR tool* and how long it that to compile.

6.1 SVDR Tool

To implement the *SVDR tool* we used *Visual Studio 2013* on a 64-bit computer with 4 gigabytes of memory and a disk of 500 gigabytes. The operating system was *Windows 8* and to run the examples we used the command line of *Windows 8*. At the begging the tool was test in a different operating system, *Ubuntu 12.10*, and it worked just fine.

This tool can be run in a 32 and 64-bit computer with both *Ubuntu* and *Windows*.ns

Concerning the time we have to wait for the tool to show the result we do not have to worry about that much because the tool is almost simultaneous.

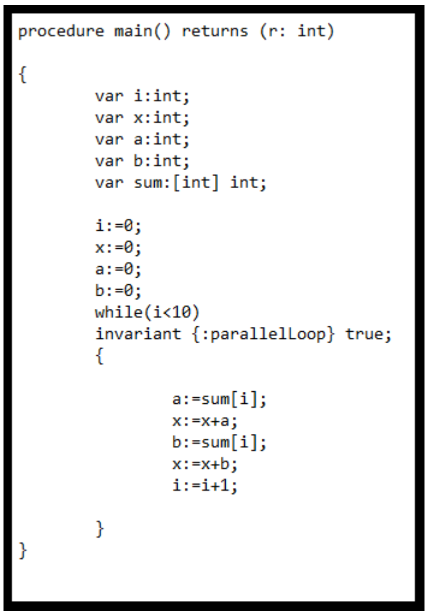
When we use *Boogie Verifier* we can check on the command line how long it took the program to present the results. This time increases with the number of invariants but that increase is not so significant. When we have several nested loops and therefore many invariants the time increases significantly and can take some time to show the result in the command line.

Despite this, the tool works properly and without noticeable delay for all the tests that we demonstrated in the report, i.e., itnsalways gives the expected result.

6.2 Example

The example is a very simple example of two reads. As explained two consecutive reads do not cause *data races* so we already now that this example does no have any problem.

Figure 6.1 shows the input used in the tool and below (in the code) is showed the output of the tool.



```
procedure main() returns (r: int)

{
        var i:int;
        var x:int;
        var a:int;
        var b:int;
        var sum:[int] int;

        i:=0;
        x:=0;
        a:=0;
        b:=0;
        while(i<10)
        invariant {:parallelLoop} true;
        {

                a:=sum[i];
                x:=x+a;
                b:=sum[i];
                x:=x+b;
                i:=i+1;

        }
}
```

Figure 6.1: Example Test 1

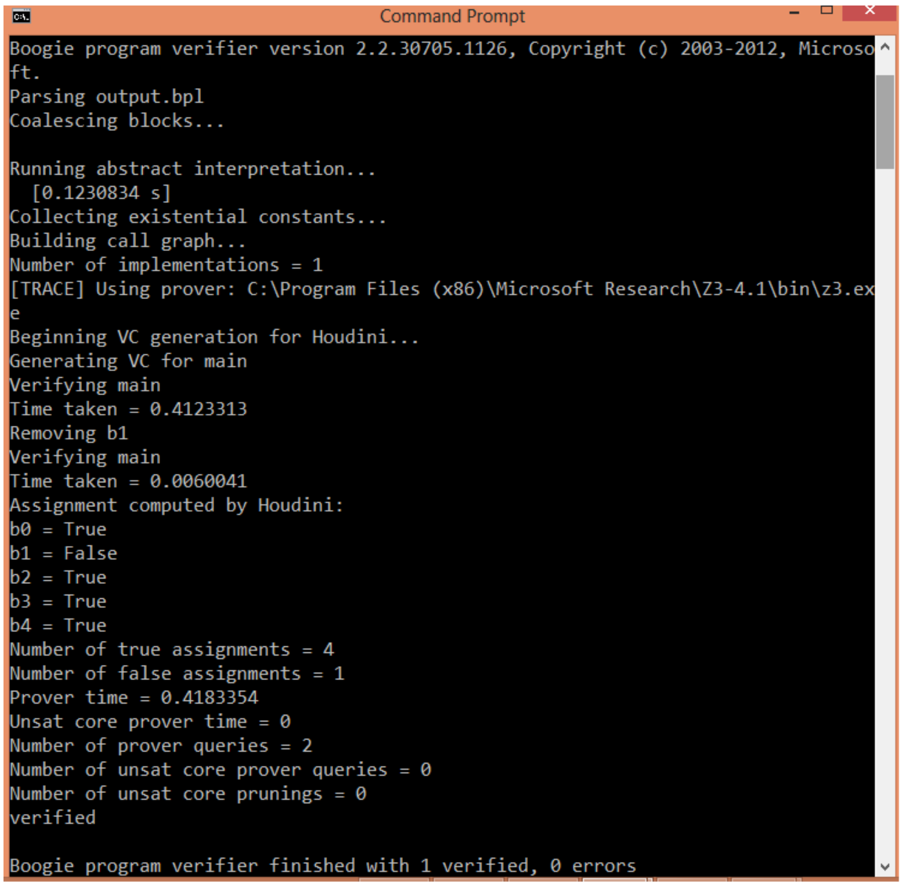
As it is possible to observe in the result produce by the tool several *invariants* were created. Beyond the *invariants*, the auxiliary methods relating to the *Checker*, *Log* and *Update_Previous* were created correctly and the call were introduced where it was supposed to.

The *Boogie Verifier* checks the output of the tool and the result is in Figure 6.2. As expected the output was verified and the *invariant* that was not true was removed by *Houdini*.

After running the input (Figure 6.1) the tool was possible to verify that this specific input is *data race* free.

```

//Boogie program verifier version 2.2.30705.1126, Copyright
    (c) 2003-2014, //Microsoft.
var prev_read_sum: bool;

var cur_read_sum: bool;

var prev_off_read_sum: int;

var cur_off_read_sum: int;

procedure main() returns (r: int);
  modifies prev_off_read_sum, prev_read_sum,
   cur_off_read_sum, cur_read_sum;

implementation main() returns (r: int)
{
  var i: int;
  var x: int;
  var a: int;
  var b: int;
  var sum: [int]int;

    i := 0;
    prev_read_sum := false;
    x := 0;
    a := 0;
    b := 0;
    while (i < 10)
      invariant {:parallelLoop} true;
      invariant b0 ==> prev_read_sum ==>
            prev_off_read_sum < i;
      invariant b1 ==> prev_read_sum ==>
            prev_off_read_sum div 1 < 1;
      invariant b2 ==> prev_read_sum ==>
            prev_off_read_sum mod 1 < 1;
      invariant b3 ==> prev_read_sum ==>
            prev_off_read_sum mod 1 <= 1;
      invariant b4 ==> (prev_read_sum ==>
            prev_off_read_sum < i|| prev_off_read_sum >= 0)
   ;
    {
        cur_read_sum := false;
        a := sum[i];
        call Checker_Read_sum(i);
```

```
            call Log_Read_sum(i);
            x := x + a;
            b := sum[i];
            call Checker_Read_sum(i);
            call Log_Read_sum(i);
            x := x + b;
            i := i + 1;
            call Update_Prev_Read_sum();
      }
}
const {:existential true} b0: bool;

const {:existential true} b1: bool;

const {:existential true} b2: bool;

const {:existential true} b3: bool;

const {:existential true} b4: bool;
```



```
Boogie program verifier version 2.2.30705.1126, Copyright (c) 2003-2012, Microso
ft.
Parsing output.bpl
Coalescing blocks...

Running abstract interpretation...
  [0.1230834 s]
Collecting existential constants...
Building call graph...
Number of implementations = 1
[TRACE] Using prover: C:\Program Files (x86)\Microsoft Research\Z3-4.1\bin\z3.ex
e
Beginning VC generation for Houdini...
Generating VC for main
Verifying main
Time taken = 0.4123313
Removing b1
Verifying main
Time taken = 0.0060041
Assignment computed by Houdini:
b0 = True
b1 = False
b2 = True
b3 = True
b4 = True
Number of true assignments = 4
Number of false assignments = 1
Prover time = 0.4183354
Unsat core prover time = 0
Number of prover queries = 2
Number of unsat core prover queries = 0
Number of unsat core prunings = 0
verified

Boogie program verifier finished with 1 verified, 0 errors
```

Figure 6.2: Verification of the Test 1

Chapter 7

Conclusion

The purpose of this work was to implement a tool that would be able to perform *Static Verification of Data Race (SVDR)* in a programming language, in this case *OpenMP*. This project went through two phases: the first was the translation of *OpenMP* into *Boogie* and the second was *static verification* of the *Boogie* translation.

Our tool, *SVDR*, implements the second phase: the *static verification of data race in the translated Boogie code*. The tool receives as input a *Boogie* file and applied the procedure to that file, and then return the file with the procedure as output. Then the file will be verified by the *Boogie Verifier* and returns positive if there is not a *data race* in the code.

SVDR was implemented in *Visual Studio 2013* and at the beginning it was developed directly in *Boogie* source code. After many tests were performed using the tool, that the tool works properly and the results were quite positive.

Since the main objectives of the project were completed successfully we can say that this opens the door for future longer-term projects on verification of multi-core programs.

7.1 Knowledge acquired

During this period in which I elaborated my thesis I acquired a lots of new knowledge. At the beginning of the thesis I had to study papers on *GPUVerify*, *OpenMP* and *Boogie* to better understand the tools and languages.

GPUVerify is one of the most important project of the working group in which I was inserted and was quite useful reading and researching about it because there is a part of the project that does a similar translation and verification as my tool.

OpenMP and *Boogie* were the languages and tools that I used in my project so before I started working with then I read a lot of papers [3], [4], [5]. It was quite a long process, to start the real project but it was worth it.

Other knowledge that I gained was about the *Linux* terminal. That was useful in the beginning of the project.

To implement the project I had to use a C#, programming language that had never used, and also had to understand the source code of *Boogie*.

In summary, all year I acquired a lot of new knowledge that will be quite useful in the future.

7.2 Future Work

There are some limitations in our tools, which were explained in detail in Chapter 4, which could easily be resolved in future work.

One would be the translation of *OpenMP* into *Boogie* which is currently performed by hand. In practical and theoretical terms it would not be very difficult to create a tool that automatically performs this translation, which would be quite advantageous for the user. ns Another issue, is that the tools produces a limited number of invariants, which may not be enough for complex program, but this can be resolved in the future. Furthermore, in future work we could extend the project to allow the use of *goto* for example and we could also take care of *x:= x + sum [y],* by creating a new method in the *SVDR* tool.

Something very important to make the tool better would be to add nested parallel loops (see Chapter 4) because it is a limitation of the tool difficult to accomplish.

Finally, in a long term project it would be possible to analyze programs written in different programming languages and apply the same algorithm to verify the absence of *data races*.

Bibliography

[1] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 69–78, 2006.

[2] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[3] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: A verifier for gpu kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 113–132, New York, NY, USA, 2012. ACM.

[4] Nathan Chong, Alastair F. Donaldson, Paul H.J. Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier invariants: A shared state abstraction for the analysis of data-dependent gpu kernels. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 605–622, New York, NY, USA, 2013. ACM.

[5] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. Interleaving and lock-step semantics for analysis and verification of gpu kernels. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 270–289, Berlin, Heidelberg, 2013. Springer-Verlag.

[6] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Microsoft Corporation, 2005.

[7] Dictionary.com. Nondeterminism, 2014. http://dictionary.reference.com/browse/nondeterminism.

[8] Software Reliability Alastair Donaldson. Loop Invariant Generation Using Houdini, 2014. http://www.doc.ic.ac.uk/ afd/teaching/SoftwareReliability/.

[9] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java, 2000.

[10] High Performance Computing. OpenMP tutorial, 2013. https://computing.llnl.gov/tutorials/openMP/.

[11] Intel. Dynamic Analysis vs. Static Analysis, 2014. http://software.intel.com/sites/products/documentation/doclib/iss/2013/inspector/lin/ug_docs/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm.

[12] Intel. Intel Parallel Studio, 2014. https://software.intel.com/enus/intelparallelstudio-xe.

[13] Intel. Parallel Lint Overview, 2014. http://software.intel.com/sites/ products/documentation/ studio/composer/enus/2011Update/compiler_c/bldaps_cls /common/bldaps_svover.htm.

[14] Intel. Problem Detection and Analysis with Parallel Lint, 2014. http://software.intel.com/sites/products/documentation/ studio/composer/enus/2011Update/compiler_c/bldaps_cls/ common/bldaps_svparallellint.htm#bldaps_svparallellint.

[15] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, 2012.

[16] Shuvendu K. Lahiri and Julien Vanegue. Explain Houdini: Making Houdini Inference Transparent. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 309–323, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] K. Rustan M. Leino. This is Boogie 2. 2008. Informal document.

[18] Yuan Lin. Static nonconcurrency analysis of openmp programs. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, IWOMP'05/IWOMP'06, pages 36–50, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Microsoft Research. Spec#, 2013. http://research.microsoft.com/enus /projects/specsharp/.

[20] Microsoft Research. Rise4fun : Boogie, 2014. http://rise4fun.com/Boogie/.

[21] Microsoft Developer Network. OpenMP in Visual C++, 2014. http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx.

[22] University of Porto: Parallel and Distributed Programming. Introducao ao OpenMP, 2014. http://www.dcc.fc.up.pt/ fds/aulas/PPD/0708/intro_openmp-1x2.pdf.

[23] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming, 2014. http://openmp.org/wp/about-openmp/.

[24] Charles Severance and Kevin Dowd. Understanding Parallelism - Loop-Carried Dependencies, 2014. http://cnx.org/content/m32782/latest/.

[25] Ian Sommerville. Software Engineering. Addison-Wesley; 9 edition, 2010.

[26] Techopedia. Static verification, 2014. www.techopedia.com/definition/13696/static-verification.

[27] Veracode. Static analysis, 2014. http://www.veracode.com/products/ static-analysis-sast/static-analysis-tool.

[28] Wikipedia. OpenMP, 2014. http://en.wikipedia.org/wiki/OpenMP.

[29] Joel Yliluoma. Guide into OpenMP: Easy multithreading programming for C++, 2013. http://bisqwit.iki.fi/story/howto/openmp/.

[30] Fang Yu, Shun-Ching Yang, Farn Wang, Guan-Cheng Chen, and Che-Chang Chan. Symbolic consistency checking of OpenMp parallel programs. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 139–148, New York, NY, USA, 2012. ACM.