Cohesive Project Report
Optimization of C Code for Critical Systems
Formal Methods Software Engineering
Master in Computer Engineering
University of Minho 2013/2014

Grupo

João Carlos Alves da Cruz

July 13, 2014

Contents

List of Figures

1 Introduction

The methodical and rigorous development of software systems is a complex task. Therefore the software engineer must be gifted with skills and techniques, that allows solving a problem in the best possible way. Software engineering for life critical systems is particularly complex and this is where Formal Methods emerge.

In today's industry, Formal Methods have been used for increasing the reliability of systems, especially in systems whose failure may result in death or serious injury to people. The software designed for these systems demands the use of a rigorous mathematical verification process. It is through these mathematical models that we are able to predict the behaviour of the programs before its implementation by reasoning and calculating the models.

It is in this sense and in the context of the Formal Methods in Software Engineering course, inserted in the Master in Computer Engineering of Minho University, that came up the possibility to do this Project has come up, together with Efacec , whose topic is "Optimization of C Code for Critical Systems".

The Software Systems development process defines the set of tasks and results that give rise to a software product. To explain the development of this project, this report is divided in three main steps : the *Specification*, the *Development* and the *Validation*.

The first step is Specification, where the goals of this project are explained, more precisely the features of the system and its restrictions. This step will be covered by the second, third and fourth sections of this report.

The second step is Development, which corresponds to the productive component, since the architectural conception until the coding language, justifying all the options and decisions that were made during this project. This step will be covered by the fifth, sixth and seventh sections of this report.

Finally the third step is Validation, where it is verified if what was actually developed corresponds to what was specified and works as intended. This step will be covered by the eighth section with a Case of study.

2 Project

2.1 Problem Description

Efacec offers a complete solution for railway signalling systems, including a software system responsible for the implementation of signalling rules, known as interlocking. Interlocking is one of the most critical elements within a signalling system, thus requiring a complex verification and validation process, in order to ensure that no hazard situations can be triggered by errors contained within this software system.

The development of interlocking system follows the model-driven development and it is made by using SCADE tool. At first, a model of the interlocking system is created. Then, this model is tested and verified to be correct regarding its functional requirements. At last, a C program is automatically generated from the provided model.

However, the generated C code is not optimized, which it is unacceptable when the target system has limited and expensive resources. The target systems are automatons with restrictions regarding memory and processing resources.

2.2 Project Description

The C code generated by KCG (the SCADE tool that generates C code) contains a lot of unnecessary variables, required only to map the generated code back to the model but not for the program logic. The removal of these variables is a key optimization, representing an important improvement concerning memory resources. Its implementation does not seem to be complex, nevertheless and since this is a critical system, it is required to ensure that the optimization process does not affect the behaviour of the program. In this context, Efacec proposes a project consisting of two main steps:

- **Step 1:** to implement an optimization technique which relies on the removal of unnecessary variables within the code generated by the KCG tool;
- **Step 2:** to verify that the C program resultant from the optimization process is functionally equivalent to the one not optimized.

2.3 Tasks

- To familiarize with SCADE and KCG tools;
- To understand the optimization technique concerning the removal of unnecessary variables;
- To create a tool that automatically applies the mentioned optimization technique, which should only require a C program as input;

- To provide a method that allows to verify if the optimization process does not interfere with the program functionality;
- To provide a case study where the previous tasks can be demonstrated;
- To write a project report.

3 SCADE Suite KCG C Code Generator

In this section the tool SCADE Suite KCG C Code Generated will be presented and briefly explored. This tool is used by Efacec to generate C code from models provided by them, as explained in the previous section.

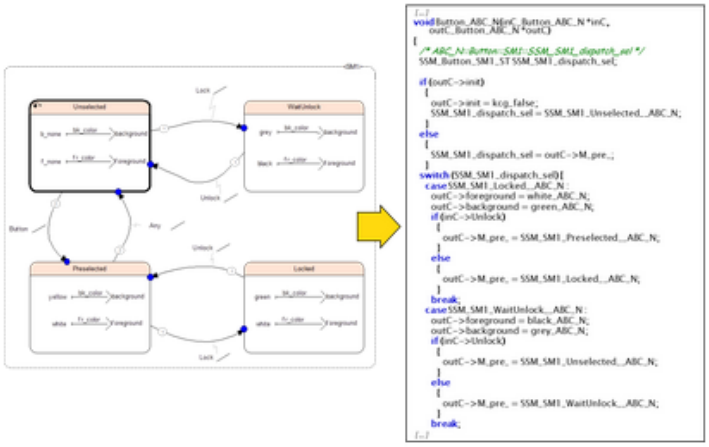

Figure 1: Connection made between models and code by KCG

SCADE Suite KCG, as the name indicates, is a C code generator, which has been certified for IEC 61508 at SIL 3, and for CENELEC EN 50128 at SIL 3/4. SIL is a measurement of safety system performance and stands for Safety Integrity Level, and according to railroad standards (EN50126, EN50128 and EN50129), SIL 4 is the maximum security level.

In the context of this project, it is important to become familiar with this code generator for the application development and as we shall further see to verify the transformation process required. So what is important to retain is that the C code provided by SCADE Suite KCG has the following important properties:

- Optimized code for all constructs;
- Static memory allocation;
- No pointer arithemetic;
- No recursion, bounded loops only;
- Bounded execution time.

4 Optimization Technique

In this section, the specific Optimization technique will be exhibited, according to the specifications provided by Efacec. As mentioned before, the transformation relies on the removal of unnecessary variables in the generated code, but the question is how to identify which variable is unnecessary and which one is not.

So, the program that will implement the optimization technique has to perform the desired behaviour :

1. The program receives two input files : one file with ".cpp" extension and another file with the same name but with ".h" extension;
2. In the *.cpp input file, identify any variables whose name start with "_L" and are assigned only once in the code;
3. Replace all variables identified in the previous step and remove the corresponding assignment instruction. The replaced code must be between parentheses;
4. Remove the declaration instruction, in the *.h file, for every variable that was replaced;

After studyin these specifications some code special cases should be carefully analysed, and according to Efacec the desired behaviour should be:

- In the following case, when the same variable is assigned in the *if statement* and in the *else statemet*, the variable cannot be removed because it is assigned more than one time, although the instructions are mutually exclusive.

```
if(....)
    {_Lxxxx= y; }
else
    {_Lxxxx= z;}
```

- The second case is the following situation, when there is the possibilty to perform a substitution inside another substitution:

```
_Lxxx1 = _Lxxx2;
_Lxxx3 = _Lxxx1;
_Lxxx4 = _Lxxx3;
```

And the expected result should be:

```
_Lxxx4 = ((_Lxxx2))
```

5 Explored Tools

5.1 Introduction

In order to find a viable tool that automatically applies the mentioned technique, a set of tools were assessed and tested. In this section, we present a simple definition of what each tool does and the advantages such tools would bring for this project in case they were used, pointing out the reasons why those tools were or were not useful for the project. The tools that have been explored are : Frama-C, CIL (C Intemediate Language), ANTLR Grammars for C code, libCLANG and finally the chosen tool Flex. In short terms, this section represents all the paths taken during the project and at the same time justifies the advantages and the disadvantages of each path.

5.2 Frama-C

Frama-C[1] is a suite of tools dedicated to the analysis of the source code of software written in C. It gathers several static analysis techniques in a single collaborative framework.

Frama-C is organized with a plug-in architecture. Plug-ins interact with each other through interfaces defined by the kernel. The three most famous plug-ins are: Value Analysis, Jessie and Wp.

The reasons why Frama-C was explored the first time are the Value Analysis plug-in, which could be useful to identify the variables we want to remove and the Slicing plugin, for Code Transformation. Besides Frama-C provides a set of good advantages such as:

- Static analysis of source code;
- Ready-made parser;
- Powerful semantic information like the notion of variables, function, types;

Unfortunately, Frama-C was not useful essentially for one reason: the plug-in Slicing, slices the code according to a user-provided criterion, but the optimization that is intended is very specific and the program cannot cope with that. In other words, there is no way to specify to Frama-C the kind of slice that is requested.

5.3 CIL

CIL[2] is an Infrastructure for C Program Analysis and Transformation and stands for C Intermediate Language. The main advantage of CIL is that it compiles all valid C programs into a few core constructs with a very clean semantics

[1] http://frama-c.com/index.html
[2] http://kerneis.github.io/cil/

and has a syntax-directed type system that makes it easy to analyse and manipulate C programs. Basically, CIL is a highly-structured "clean" subset of C.

The idea of using CIL, was the possibility of having the AST (Abstract Syntax Tree) representation of the code and with that the parser development would be much easier.

The downside of this tool, relatively to the required optimization, is compiling C to the CIL process. This process is made by a set of transformations that are applied to a C program to convert ir it to CIL, but some of these transformations will eliminate declarations, change the variables names, change the type definitions from structures, unions, and other modifications that were not required. Even if the transformation process succeds in the future, during the certification of the transformation problems may appear, because the comparison between the two programs, the one optimized and the one not optimized will be very difficult to make.

5.4 ANTLR Grammars for C Code

Another path explored during this project was the possibility of making an ANTLR Grammar for C Language. The idea appeared because of all the advantages that a C code Grammar has, such as : direct access to the AST of the C program, the transformation process and code generation would be very easy to implement with ANTLR and total liberty and control over the specific optimization process. This control is very important, especially in the certification process. Therefore, some ANTLR Grammars for C code were found[3] online and tests but with no satisfying results. Most of the grammar was unfinished in the way that it did not cover all the complete C language. Then, the idea of creating a new grammar for this project came up, but after some research the conclusion was that making a grammar for the complete C Language is a very complex process.

5.5 Libclang

Clang[4] is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages and it uses LLVM as its back end. Clang provides infrastructure to write tools that need syntactic and semantic information about a program.

LibClang is a C Interface to Clang and was tested because it provides a relatively small API which could be useful for parsing a source code into a Clang AST, loading an already-parsed AST and travesing the AST. The Clang AST is different from others ASTs produced by other compilers, it has a powerful

[3]http://www.antlr3.org/grammar/list.html
[4]http://clang.llvm.org/docs/index.html

semantic representation and the number of provided features for each AST Node (called CXCursor) is very useful.

Clangs AST nodes are modelled on a class hierarchy that does not have a common ancestor. Instead, there are multiple larger hierarchies for basic node types like Decl and Stmt. Many important AST nodes derive from Type, Decl, DeclContext or Stmt, with some classes deriving from both Decl and DeclContext. In the following picture a simple example of a Clang AST is presented.

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

Figure 2: Abstract Syntax Tree of Clang

The reasons why the CLANG was left behind are:

- The tool its very hard to install, with many potentially problematic dependencies which will led to problems when linking the program to the Interface;
- LibClang parser it is too smart and the types from KCG are not accepted;
- Some barriers were found with the generation of the Optimized Code;

Despite the disadvantages listed above, CLANG is very powerful and very useful in our transformation tool. The semantics it offers is immense and all API provided allows to do many things. With a bit more study and due to its enormous complexity the Clang could be used for this project and future work should be greatly facilitated.

5.6 Flex

Flex[5] is a fast lexical analyser generator. It is a tool for generating programs that perform pattern-matching on text.

The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of **regular expressions** and C code, called rules. Flex generates a C source file named, "lex.yy.c", which defines the function yylex(). The file "lex.yy.c" can be compiled and linked to produce an executable. When the executable is run, it analyses its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.

Let us see a very simple example:

```
%%

[0-9]+     { printf(" Flex recognized the %s number!",yytext); }

%%
```

In the example, only one rule is provided to Flex with a regular expression that says to find one or more (by the symbol "+") digit between 0 and 9, and a C code action that says to print out its value every time the indicated pattern appears. The *yytext* is a variable provided by Flex, which stores the tokens that were recognized.

The reasons why Flex was viable are:

- It is a tool that gives us more control on what we want to recognize on a file, in this case we can specify directly the kind of variables we want to analyse (␣L);
- The possibility to erform easy generation of code to an output file;
- Easy to install and use;
- Using Flex we lose some semantic information, but in this case it is not a big problem;
- We can benefit from the fact that the code we are going to optimize is a code generated by a tool;

[5] http://flex.sourceforge.net/

6 Application

6.1 Introduction

The Application consists of one User Interface, that was developed in Java, using the NetBeans IDE and is linked to a program that performs the optimization technique written in C. Therefore, in this section the program will be explained to detail in order to understand how *Flex* was used to perform the desired transformationd mentioned in the Optimization Technique section. The the Application Interface will be ilustrated.

6.2 Main Program

Firstly, for simplification purposes, the term valid variable refers to a variable that respects the substitution invariant, this is, the variable name starts with "_L" and is assigned only once in the input code. As mentioned before the main program receives as an input two files: one with the ".cpp" extension and another with ".h" extension.

The program uses a pipeline of four flex filters. To support this pipeline two hashtables are declared, one for the *valid variables* and one for the *invalid variables*. The name of the variables is used by the hashtable as the key to match the value assigned to that variable. In the invalid variables hashtable, the value of each variable is NULL because that is not necessary to store it.

1. **aalex** - the first flex file is responsible for reading the ".cpp" file, identifying assignment instructions and stroring the valid and the invalid variables.

   Between the *aalex* and *bblex* the program executes a substitution algorithm, on the valid variables hashtable. What the algorithm does is travel the valid variables hashtable and for each variable, it sees if in his value exists some valid variable name that could be substituted. While this is true, it performs the substitution directly in the mentioned hashtable, when it is not, it moves on for another valid variable, until it reaches the end. Next an example is presented as well the final result.

   This algorithm is required because from now on only one travel to the source code is needed to substitute the valid variable name for its final value. All this work can only be possible to do with hashtable because the order in which the variables appear on the code is irrelevant, such can be explained by the valid variable invariant: only once is assigned.

2. **bblex** - the function of this second flex file is to perform the substitution of the valid variables. This is done by reading the ".cpp" input file and

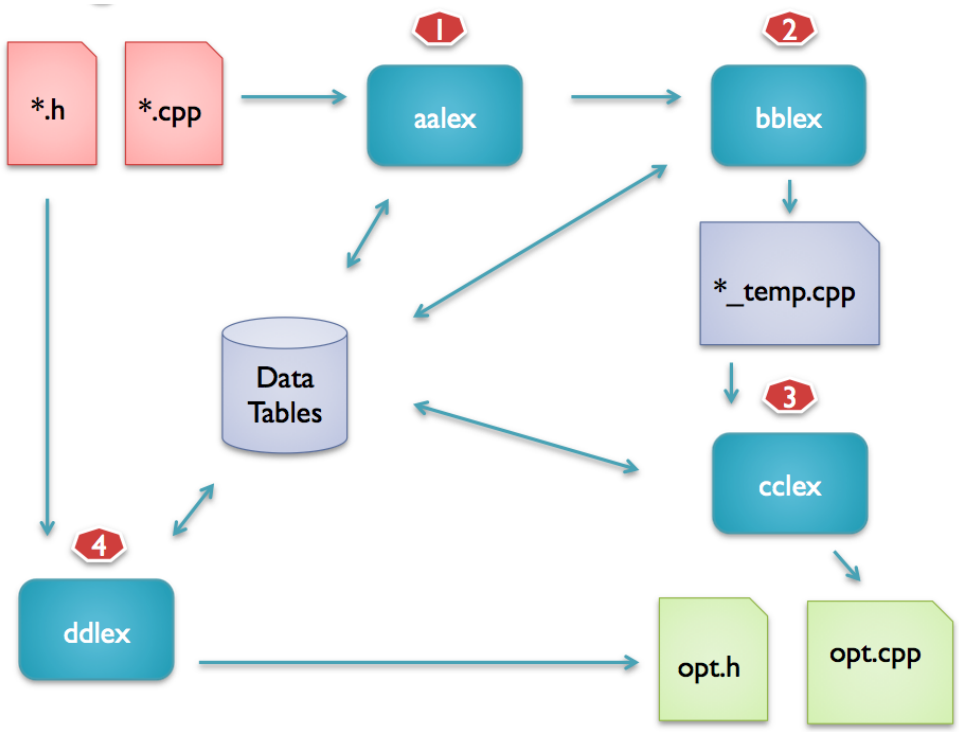

Figure 3: Program architecture with the flex pipeline

generating a "_tem.cpp" file which is exactly like the input but with all the final substitutions already made;

3. **cclex** - the third flex file is responsible for reading the "_tem.cpp" file generated before and for removing the assignment instructions which have the valid variable, by checking the valid variables hashtable;

4. **ddlex** - finally the last flex file, whose mission is to read the input file with the ".h" extension and to remove the declarations instructions of the valid variables substituted before.

To perform this pipeline of flex file a main C code file is presented.

6.3 Application Interface

The Application Interface for this project is very simple to understand and was built according to all specifications provided by Efacec.

The main panel shown is the next picture is composed by three main areas.

The first, is the Optimization Module area, where the user selects the optimization he intends to make.

Figure 4: Application Interface Main Panel

The second, is Input File area where the user selects the ".cpp" file he intends to optimize.

The last one, is the Action area where the user run the program.

Figure 5: Optimization Module selection

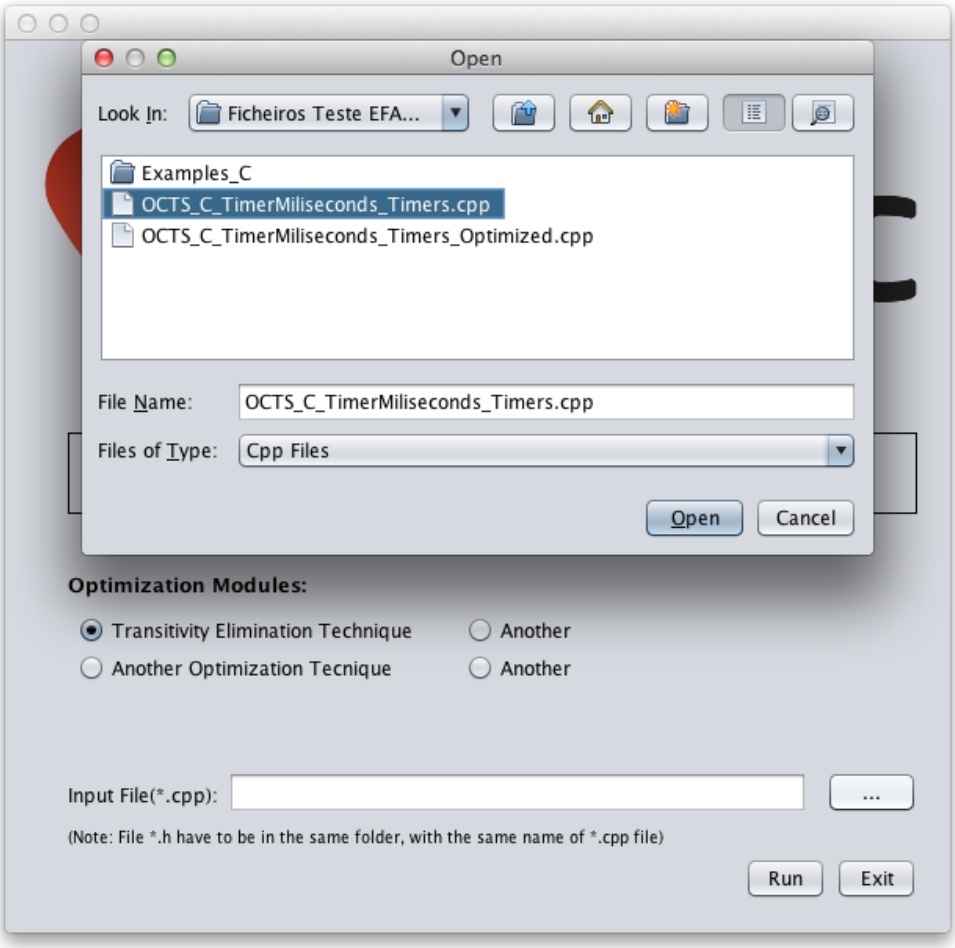

Figure 6: Input File selection

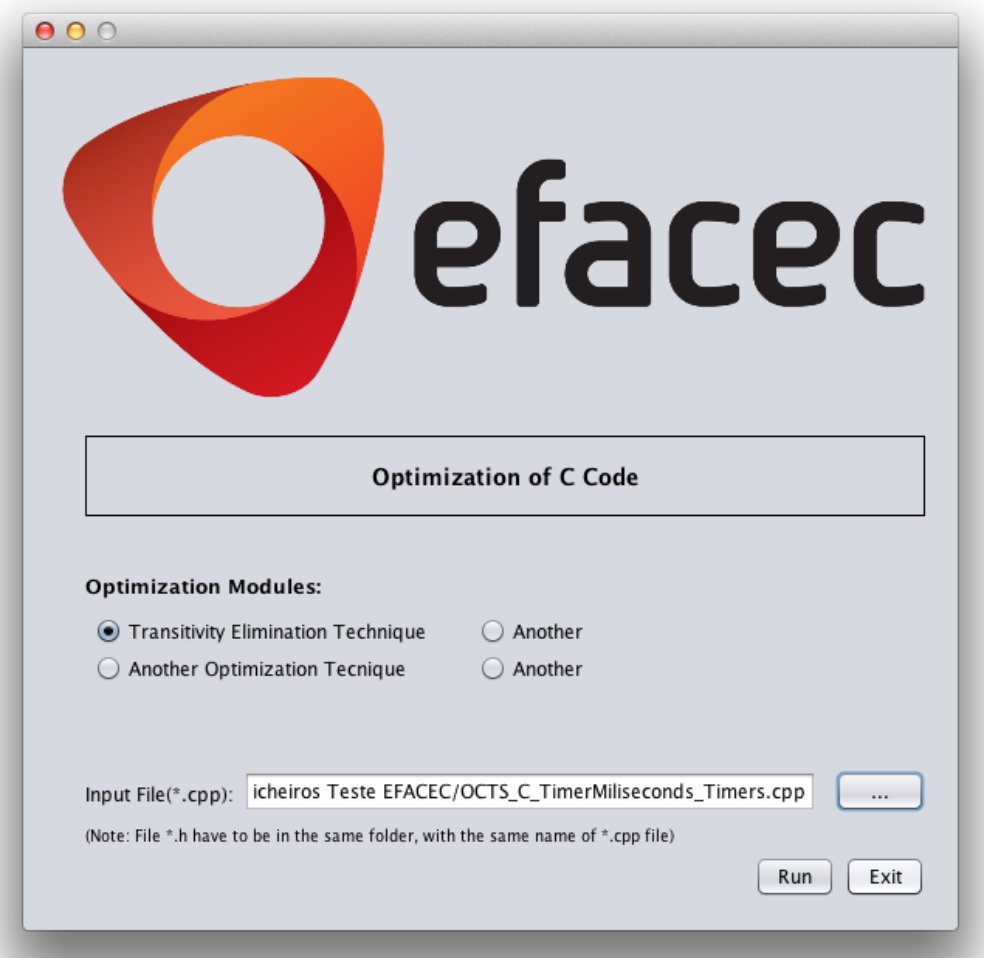

Figure 7: Run the program area

7 Verification Method

7.1 Introduction

Reached this stage, and after the transformation program was explained as well as the Interface for the Application, it is time to visit the Step 2 proposed of the project: to verify that the C program resultant from the optimization process is functionally equivalent to the one not optimized.

From this moment is where Formal Methods actually begin to make sense, more specifically a field in Formal Methods called **Software Formal Verification**. As result, we nedd to provide a method that allows to verify if the optimization process does not interfere with the program functionality and the method that was explored is called the Composition Technique.

Therefore, in this section we will explain this Composition Technique and present a simple example. This technique can be proved by using a Bounded Model Checker called **CBMC**[6].

7.2 Composition Technique

According to the notion of equality of programs, two programs are equal if for all initial states, the execution of both results in the same final state or none of them ends.

This composition technique is done by making a program that is the composition of two programs whose equivalence we are trying to prove, renaming all variables in one of the programs.

Let us consider the following example of a cycle:

```
for (k=i ; k<=j; k++) {
    b += k;
    a *= k;
}
```

Which can be refactored to this equivalent code, since there is no interaction between the **a** and **b** variables:

```
for (k=i; k<=j; k++) b += k;
for (k=i; k<=j; k++) a *= k;
```

In order to use the composition technique mentioned before, the folowing program can be done, and the order of the programs is irrelevant:

[6]http://www.cprover.org/cbmc/

```
for (k=i; k<=j; k++) b += k;
for (k=i; k<=j; k++) a *= k;


for (ks=is ; ks<=js; ks++) {
    bs += ks;
    as *= ks;
}
```

As the name space of both is disjoint, the sequential execution allows to think about the independent execution of each program but connecting the initial and the final variables values of both programs.Now to prove this equality we have to appeal to a verification tool such as CBMC.

CBMC is a Bounded Model Checker that allows verifying array bounds (buffer overflows), pointer safety, exceptions and **user-specified assertions**.The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.

All we have to do is say to CBMC that the initial state and the final state of the compound program must be equal. To do this comunication with CBMC in the source code we will use two built-in modeling primitives:

- **__CPROVER_assume (*expression*)** - this primitive restricts program traces to those satisfying the assumption;
- **assert(*expression*)** - this primitive specify the properties we want to check;

And from the example before, the compound code that is sent to CBMC is:

```
int a,b,as,bs,ks,k,i,is,j,js;
int main () {
    __CPROVER_assume(k==ks && i==is && j==js && a==as && b==bs);

    for(k=i ; k<=j ; k++) b += k;
    for(k=i ; k<=j ; k++) a *= k;

    for(ks=is ; ks<=js ; ks++) { bs += ks; as *= ks; }

    assert(k==ks && i==is && j==js && a==as && b==bs);
}
```

8 Case Study

8.1 Transformation on the .cpp file

In this section a Case of Study is presented, as a way to verify if what was actually developed corresponds to what was specified and works as intended.

Therefore, the example we will explore is a real test file provided by Efacec. First let us take a look to an excerpt of "caseofstudy.cpp" file, presented in the following picture.

```
29  if (outC->init) {
30    last_l_Miliseconds = 0;
31  }
32  else {
33    last_l_Miliseconds = outC->l_Miliseconds;
34  }
35  outC->_L151 = i_Reset;
36  outC->_L129 = 0;
37  outC->_L139 = i_ResetDelayed;
38  outC->_L142 = !outC->_L139;
39  outC->_L150 = i_Reset;
40  /* 1 */ OCTS_FallingEdge_Edge(outC->_L150, &outC->Context_1);
41  outC->_L137 = outC->Context_1.o_Out;
42  outC->_L140 = outC->_L142 & outC->_L137;
43  outC->_L135 = 0;
44  outC->_L145 = i_LastTimeCycle;
45  /* 2 */ OCTS_Sign_INT_Math(outC->_L145, &outC->Context_2);
46  outC->_L148 = outC->Context_2.o_Out;
47  outC->_L143 = i_LastTimeCycle;
48  outC->_L147 = 0;
49  if (outC->_L148) {
50    outC->_L144 = outC->_L143;
51  }
52  else {
53    outC->_L144 = outC->_L147;
54  }
55  outC->x_MyTime = outC->_L144;
56  outC->_L124 = outC->x_MyTime;
57  if (outC->_L140) {
58    outC->_L136 = outC->_L135;
59  }
60  else {
61    outC->_L136 = outC->_L124;
62  }
63  outC->_L112 = last_l_Miliseconds;
64  outC->_L90 = outC->_L136 + outC->_L112;
65  outC->x_CalculatedValue = outC->_L90;
66  outC->_L106 = outC->x_CalculatedValue;
67  /* 1 */ OCTS_Sign_INT_Math(outC->_L106, &outC->_1_Context_1);
68  outC->_L92 = outC->_1_Context_1.o_Out;
69  outC->_L108 = !outC->_L92;
70  outC->x_PositiveOverflow = outC->_L108;
71  outC->_L110 = outC->x_PositiveOverflow;
72  outC->_L113 = 2147483647;
73  outC->_L96 = outC->x_CalculatedValue;
74  if (outC->_L110) {
75    outC->_L103 = outC->_L113;
76  }
77  else {
78    outC->_L103 = outC->_L96;
79  }
80  if (outC->_L151) {
81    outC->_L101 = outC->_L129;
82  }
```

Figure 8: File "caseofstudy.cpp" before transformation

Although this excerpt not be completed, it is possible to see cleary that exists variables that obey to the invariant, so they can be substituted such as _L151, _L129 and more. But there is also some variables that did not obey to the invariant such as _L136 and _L144, so they cannot be substituted.

In the next picture is presented the "caseofstudy_optimized.cpp" file, after running the program. As we can see, the substitutions were made and the assignmet instructions were removed. It is also possible to see that the invalid variables were not substituted.

```
  if (outC->init) {
    last_l_Miliseconds = 0;
  }
  else {
    last_l_Miliseconds = outC->l_Miliseconds;
  }
  /* 1 */ OCTS_FallingEdge_Edge((i_Reset), &outC->Context_1);
  /* 2 */ OCTS_Sign_INT_Math((i_LastTimeCycle), &outC->Context_2);
  if ((outC->Context_2.o_Out)) {
    outC->_L144 = (i_LastTimeCycle);
  }
  else {
    outC->_L144 = (0);
  }
  outC->x_MyTime = outC->_L144;
  if ((((!(i_ResetDelayed)) & (outC->Context_1.o_Out))) {
    outC->_L136 = (0);
  }
  else {
    outC->_L136 = (outC->x_MyTime);
  }
  outC->x_CalculatedValue = (outC->_L136 + (last_l_Miliseconds));
  /* 1 */ OCTS_Sign_INT_Math((outC->x_CalculatedValue), &outC->_1_Context_1);
  outC->x_PositiveOverflow = (!(outC->_1_Context_1.o_Out));
  if ((outC->x_PositiveOverflow)) {
    outC->_L103 = (2147483647);
  }
  else {
    outC->_L103 = (outC->x_CalculatedValue);
  }
  if ((i_Reset)) {
    outC->_L101 = (0);
  }
  else {
    outC->_L101 = outC->_L103;
  }
  outC->x_Miliseconds = outC->_L101;
  outC->o_LimitReached = (((outC->x_Miliseconds)>= (i_Limit)) & (!(i_Reset)));
  outC->l_Miliseconds = (outC->x_Miliseconds);
  outC->init = kcg_false;
}
```

Figure 9: File "caseofstudy_optimized.cpp"

8.2 Transformation on the .h file

Next are presented the transformations that the file "caseofstudy.h" suffered. The desire transformation is the removal of the variables that were substituted before, so we can see that the invalid variables declaration instruction was not been removed in the "caseofstudy_optimized.h" file, as expected. We can look particularly for the _L136 and _L144 variables.

```
typedef struct {
  /* ----------------------------  outputs  ---------------------------- */
  kcg_bool /* Timers::C_TimerMiliseconds::o_LimitReached */ o_LimitReached;
  /* ------------------------  no local probes  ------------------------ */
  /* -------------------- initialization variables  -------------------- */
  kcg_bool init;
  /* ----------------------- local memories  ------------------------ */
  kcg_int /* Timers::C_TimerMiliseconds::l_Miliseconds */ l_Miliseconds;
  /* --------------------  sub nodes' contexts  --------------------- */
  OCTS_outC_Sign_INT_Math /* 1 */ _1_Context_1;
  OCTS_outC_Sign_INT_Math /* 2 */ Context_2;
  OCTS_outC_FallingEdge_Edge /* 1 */ Context_1;
  /* ------------------ no clocks of observable data ------------------- */
  /* --------------------- (-debug) no assertions  --------------------- */
  /* ------------------- (-debug) local variables --------------------- */
  kcg_bool /* Timers::C_TimerMiliseconds::x_PositiveOverflow */ x_PositiveOverflow;
  kcg_int /* Timers::C_TimerMiliseconds::x_CalculatedValue */ x_CalculatedValue;
  kcg_int /* Timers::C_TimerMiliseconds::x_MyTime */ x_MyTime;
  kcg_int /* Timers::C_TimerMiliseconds::x_Miliseconds */ x_Miliseconds;
  kcg_int /* Timers::C_TimerMiliseconds::_L113 */ _L113;
  kcg_int /* Timers::C_TimerMiliseconds::_L112 */ _L112;
  kcg_bool /* Timers::C_TimerMiliseconds::_L110 */ _L110;
  kcg_bool /* Timers::C_TimerMiliseconds::_L108 */ _L108;
  kcg_int /* Timers::C_TimerMiliseconds::_L106 */ _L106;
  kcg_int /* Timers::C_TimerMiliseconds::_L103 */ _L103;
  kcg_int /* Timers::C_TimerMiliseconds::_L101 */ _L101;
  kcg_int /* Timers::C_TimerMiliseconds::_L96 */ _L96;
  kcg_bool /* Timers::C_TimerMiliseconds::_L92 */ _L92;
  kcg_int /* Timers::C_TimerMiliseconds::_L90 */ _L90;
  kcg_int /* Timers::C_TimerMiliseconds::_L124 */ _L124;
  kcg_int /* Timers::C_TimerMiliseconds::_L129 */ _L129;
  kcg_bool /* Timers::C_TimerMiliseconds::_L137 */ _L137;
  kcg_int /* Timers::C_TimerMiliseconds::_L136 */ _L136;
  kcg_int /* Timers::C_TimerMiliseconds::_L135 */ _L135;
  kcg_bool /* Timers::C_TimerMiliseconds::_L139 */ _L139;
  kcg_bool /* Timers::C_TimerMiliseconds::_L140 */ _L140;
  kcg_bool /* Timers::C_TimerMiliseconds::_L142 */ _L142;
  kcg_int /* Timers::C_TimerMiliseconds::_L147 */ _L147;
  kcg_int /* Timers::C_TimerMiliseconds::_L145 */ _L145;
  kcg_int /* Timers::C_TimerMiliseconds::_L144 */ _L144;
  kcg_int /* Timers::C_TimerMiliseconds::_L143 */ _L143;
  kcg_bool /* Timers::C_TimerMiliseconds::_L148 */ _L148;
  kcg_bool /* Timers::C_TimerMiliseconds::_L150 */ _L150;
  kcg_bool /* Timers::C_TimerMiliseconds::_L151 */ _L151;
  kcg_bool /* Timers::C_TimerMiliseconds::_L157 */ _L157;
  kcg_int /* Timers::C_TimerMiliseconds::_L156 */ _L156;
  kcg_bool /* Timers::C_TimerMiliseconds::_L155 */ _L155;
  kcg_bool /* Timers::C_TimerMiliseconds::_L154 */ _L154;
  kcg_bool /* Timers::C_TimerMiliseconds::_L153 */ _L153;
  kcg_int /* Timers::C_TimerMiliseconds::_L152 */ _L152;
  kcg_int /* Timers::C_TimerMiliseconds::_L158 */ _L158;
} OCTS_outC_C_TimerMiliseconds_Timers;
```

Figure 10: File "caseofstudy.h" before transformation

```
typedef struct {
  /* ----------------------------  outputs  ---------------------------- */
  kcg_bool /* Timers::C_TimerMiliseconds::o_LimitReached */ o_LimitReached;
  /* ------------------------  no local probes  ------------------------ */
  /* -------------------- initialization variables  -------------------- */
  kcg_bool init;
  /* ----------------------- local memories  ------------------------ */
  kcg_int /* Timers::C_TimerMiliseconds::l_Miliseconds */ l_Miliseconds;
  /* --------------------  sub nodes' contexts  --------------------- */
  OCTS_outC_Sign_INT_Math /* 1 */ _1_Context_1;
  OCTS_outC_Sign_INT_Math /* 2 */ Context_2;
  OCTS_outC_FallingEdge_Edge /* 1 */ Context_1;
  /* ------------------ no clocks of observable data ------------------- */
  /* --------------------- (-debug) no assertions  --------------------- */
  /* ------------------- (-debug) local variables --------------------- */
  kcg_bool /* Timers::C_TimerMiliseconds::x_PositiveOverflow */ x_PositiveOverflow;
  kcg_int /* Timers::C_TimerMiliseconds::x_CalculatedValue */ x_CalculatedValue;
  kcg_int /* Timers::C_TimerMiliseconds::x_MyTime */ x_MyTime;
  kcg_int /* Timers::C_TimerMiliseconds::x_Miliseconds */ x_Miliseconds;
  kcg_int /* Timers::C_TimerMiliseconds::_L144 */ _L144;
  kcg_int /* Timers::C_TimerMiliseconds::_L136 */ _L136;
  kcg_int /* Timers::C_TimerMiliseconds::_L103 */ _L103;
  kcg_int /* Timers::C_TimerMiliseconds::_L101 */ _L101;
} OCTS_outC_C_TimerMiliseconds_Timers;
```

Figure 11: File "caseofstudy_optimized.cpp"

8.3 Case of Study Compound Code to CBMC

Now let us see the syntax of the coumpound program resulting the previous transformation. As mentioned before, the program have the compound code resulted by composing the code of "caseofstudy.cpp" file and the "caseofstudy_optimized.cpp" file. The code of the compound program is large , so we only will present the built-in primitives of CBMC.

Two programs are equal if for all initial states, the execution of both results in the same final state or none of them ends. Therefore, is that we will say to CBMC, the variable have the same value before execute the compound program (assume expression) and have to be the same value again after(assert expression).

As we can see in Figure 14 , the result from CBMC to the compound program from Case of Study is **VERIFICATION SUCCESSFUL**.

```
9
10  int main(){
11
12      __CPROVER_assume(
13        o_LimitReacheds == o_LimitReached &&
14          inits == init &&
15          l_Miliseconds == l_Miliseconds &&
16          _1_Context_1s == _1_Context_1 &&
17          Context_2s == Context_2 &&
18          Context_1s == Context_1 &&
19          x_PositiveOverflows == x_PositiveOverflow &&
20          x_CalculatedValues == x_CalculatedValue &&
21          x_MyTimes == x_MyTime &&
22          x_Miliseconds == x_Miliseconds &&
23          _L144s == _L144 &&
24          _L136s == _L136 &&
25          _L103s == _L103 &&
26          _L101s == _L101 &&
27        i_LastTimeCycle == i_LastTimeCycles &&
28        i_Reset == i_Resets &&
29        i_ResetDelayed == i_ResetDelayeds &&
30        i_Limit == i_Limits &&
31        last_l_Miliseconds == last_l_Miliseconds &&
32        Context_2o_Out == Context_2o_Outs &&
33        Context_1o_Out == Context_1o_Outs &&
34        _1_Context_1o_Out == _1_Context_1o_Outs);
```

Figure 12: Defining the initial state

```
153     assert(
154       o_LimitReacheds == o_LimitReached &&
155       inits == init &&
156       l_Miliseconds == l_Miliseconds &&
157       _1_Context_1s == _1_Context_1 &&
158       Context_2s == Context_2 &&
159       Context_1s == Context_1 &&
160       x_PositiveOverflows == x_PositiveOverflow &&
161       x_CalculatedValues == x_CalculatedValue &&
162       x_MyTimes == x_MyTime &&
163       x_Miliseconds == x_Miliseconds &&
164       _L144s == _L144 &&
165       _L136s == _L136 &&
166       _L103s == _L103 &&
167       _L101s == _L101 &&
168       i_LastTimeCycle == i_LastTimeCycles &&
169       i_Reset == i_Resets &&
170       i_ResetDelayed == i_ResetDelayeds &&
171       i_Limit == i_Limits &&
172       last_l_Miliseconds == last_l_Miliseconds &&
173       Context_2o_Out == Context_2o_Outs &&
174       Context_1o_Out == Context_1o_Outs &&
175       _1_Context_1o_Out == _1_Context_1o_Outs);
```

Figure 13: In the end the execution has the same final state

```
cbmc --bounds-check teste.c
file teste.c: Parsing
Converting
Type-checking teste
file teste.c line 44 function main: function `c::OCTS_FallingEdge_Edge' is not declared
file teste.c line 45 function main: function `c::OCTS_Sign_INT_Math' is not declared
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
**** WARNING: no body for function c::OCTS_FallingEdge_Edge
**** WARNING: no body for function c::OCTS_Sign_INT_Math
size of program expression: 221 steps
simple slicing removed 3 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.0 with simplifier
3155 variables, 9500 clauses
SAT checker: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 0.089s
VERIFICATION SUCCESSFUL
```

Figure 14: In the end the execution has the same final state

9 Conclusions

- The transformation is made according to all specifications provided by Efacec;
- The application interface its simple and functional, and already set for future work, with the possibility of inserting new optimization modules;
- For all the tests that were made the feedback from the application was positive;
- There are still some small edges missing filing, like for example the user choose the path for the output files, etc;
- All the tests that, using this technique, on real test files provided by Efacec, were made by manually built the composition program;
- The main goal for near future is add to our application the step 2 automatically, so that in the future, the application can provide not only the transformed code files but also the file with the compound code, already set to be proved by the CBMC;
- The application will be to perform the required transformation but also will be able to provide an important element for the certification of the transformation;
- We really believe that this technique can actually solve the certification problem because the compound code is an element generated in the process whose verification ensures the equivalence between the two programs;

10 Thanks

In this section just want to give a word of thanks for the help obtained, the availability, constant encouragement and all the suggestions and criticisms to Professor Jorge Sousa Pinto, to the Efacec external supervisors João Martins e Helder Azevedo and also to Professor José João Almeida.