Optimization of C code for Critical Systems

"Technology that moves the world "

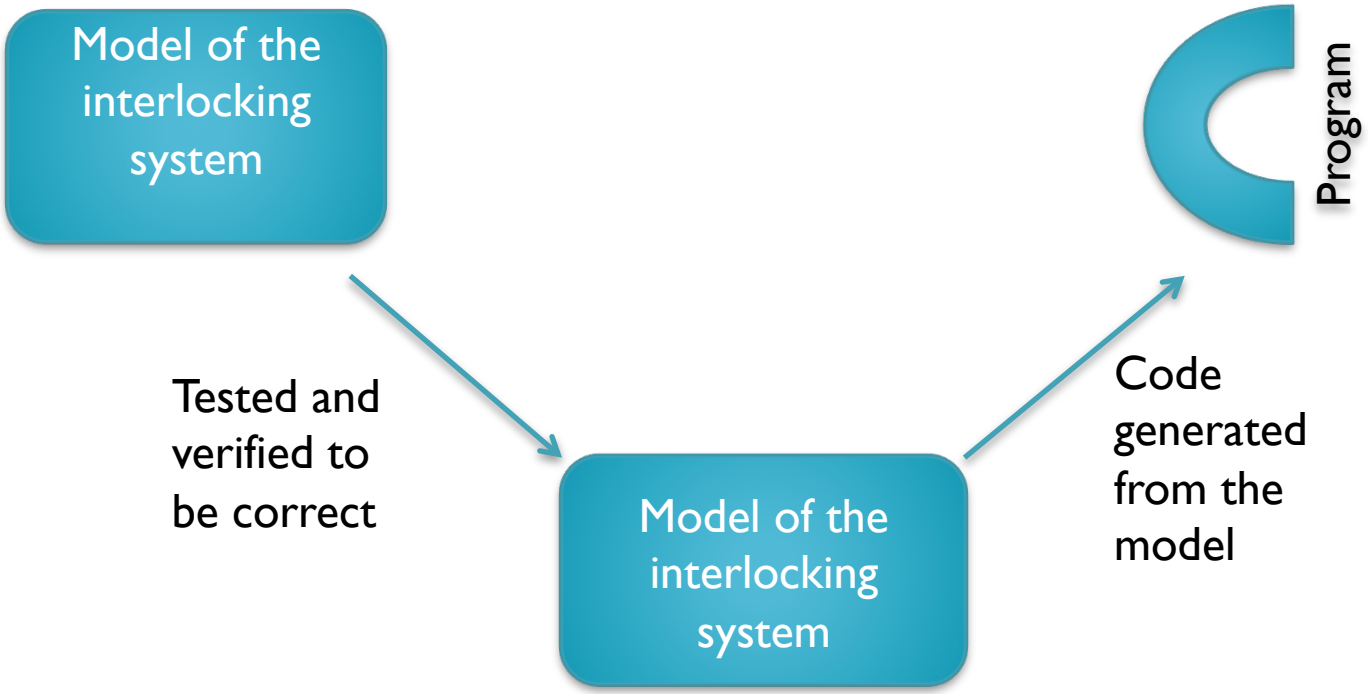
Hélder Afonso – João Cruz

University of Minho
Master in Computing Engineering
Formal Methods in Software Engineering
Project Presentation : Final Milestone

Problem Introduction

- **Efacec** offer a complete solution for railway signaling systems, including a software system responsible for the implementation of signaling rules named **interlocking**

- Interlocking system requires a **validation** and **verification** process



Project Description

- The C code generated contains a lot of unnecessary variables, not required for the program **logic**
- The removal of these variables is an essential **optimization**.
- The optimization technique is very simple : eliminate transitivity between unnecessary variables

```
type a, b, c;
/…
b = c;
/…
a = b;
/…
```

```
type a, c;
/…
a = c;
/…
```

- But this is a critical system…

Project Steps

Step 1: Implement an optimization technique which relies on the removal of needless variables within the code generated by the KCG tool. How?

Step 2 : Verify that the C program resultant from the optimization process is functionally equivalent to the one not optimized. How? Formal Methods?

Project Steps

Step 1: Implement an optimization technique which relies on the removal of needless variables within the code generated by the KCG tool.

1. Design an application interface
2. Create a program that implements the optimization technique, with the following desired behavior:

- In the *.cpp input file, identify any variables whose name start with "_L" and are assigned only once;
- Replace all variables identified in the first step and remove the corresponding assignment instruction;
- Remove the declaration instruction, in the *.h file, for every variable that was replaced;

Explored Tools

In order to find a viable tool, we have assessed and tested the following :

- Static analysis of source code
- Powerful semantic information
- Benefit from ready-made parser
- Not useful because of the code generation and the very specific optimization that is pretended

CIL (C Intermediate Language)

- Facilitate program analysis and transformation
- Problem with CIL specific language manipulation
- Transformation from C to CIL change instructions
- Generation of the C code optimized still a problem

Explored Tools

ANTLR Grammars for C Language

- Building a grammar for the complete C Language is very complex;
- It would be very useful because we have direct access to the AST;
- The Optimized Code generation it would be very easy;

CLANG : libCLang

- Provides infrastructures to write tools that need syntactic information about the program
- Provides a very good parser with a rich AST and node semantics
- Types from input files not accepted by libCLang
- Unfortunately the generation of code is a problem, and the tool its very hard to install, with many potentially problematic dependencies

Explored Tools

Flex

Flex is a fast lexical analyzer generator. It´s a tool for generating programs that perform pattern-matching on text.

- 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 analyze (_L)
- Perform easy generation of code to a output file
- Easy to install and use
- Using Flex we lose some semantic information, but in this case it´s not a big problem
- We can take profit that the code we are gone to optimize it's a code generated by a tool

Lexical Analyzer Flex

How does it work?

- Flex can be seen as a filter, or a tokenizer;
- It works based on rules, where each rule is a pair of regular expression and C code action.
- Flex generates as output a C source file witch can be compiled and linked with the flex runtime library to produce an executable.
- When the executable is run, it analyzes its input for occurrences of the regular expressions. When it find one, executes the action C code.

Simple example :

```
%%
[0-9]+                  { printf( " It´s the number %s ", yytext); }
%%
```

Back to the project…

- To implement our optimization technique, we use a pipeline of four flex filters as follows:

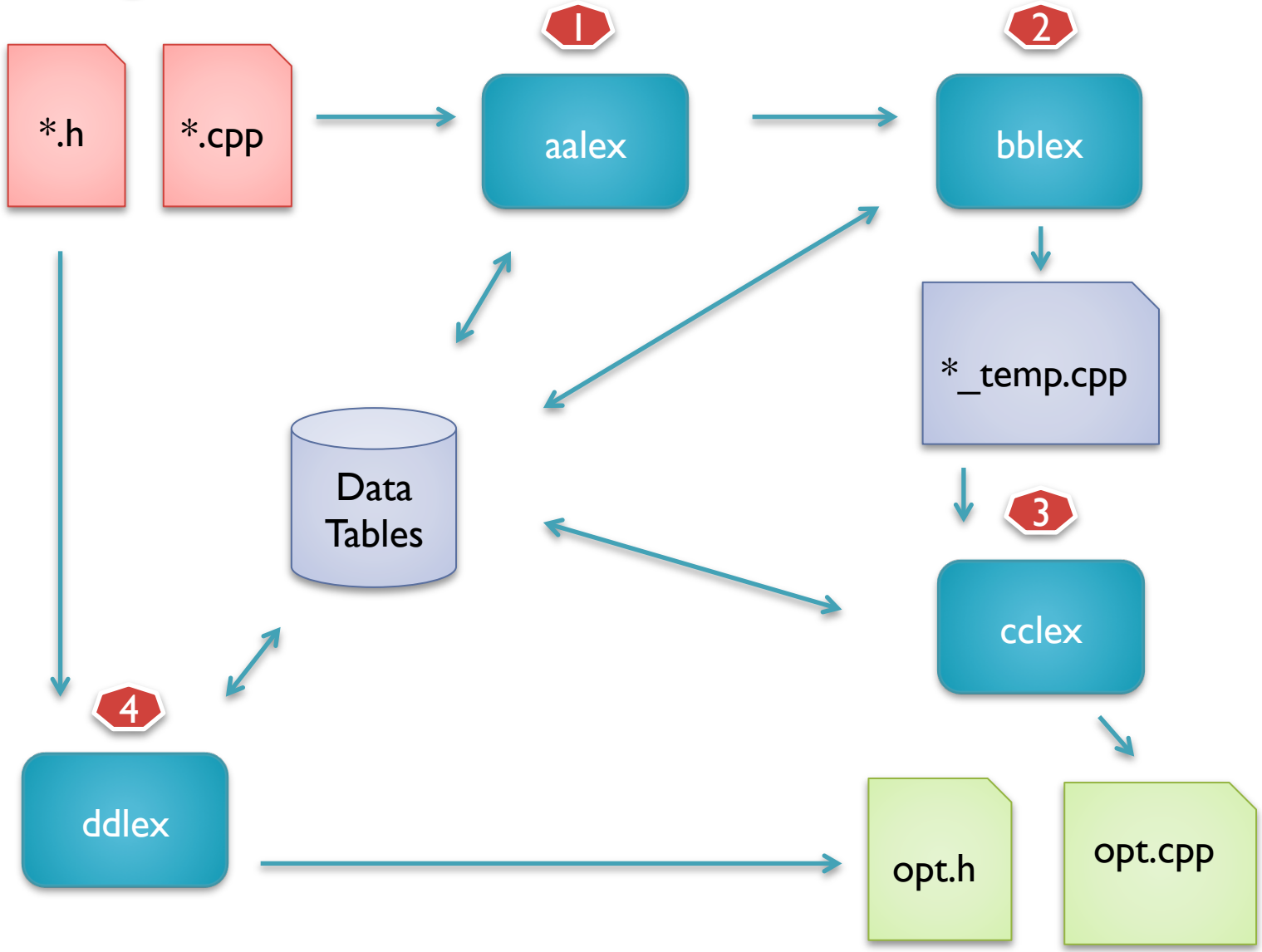
1. **aalex:** first flex file, responsible for identify assignment instructions and for storing the variable and its value, if they obey to the invariant;

2. **bblex:** its function is to perform the substitution of the valid variables (variables that obey to the invariant) and generate a *_temp.cpp file

3. **cclex:** responsible for reading the *_temp.cpp file generated before and removing the assignment instructions which have the valid variable;

4. **ddlex:** receives the input file *.h and removes the declaration instructions of the valid variables

Program Architecture



Project Steps

Step 2 : Verify that the C program resultant from the optimization process is functionally equivalent to the one not optimized.

- Bounded Model Checker **CBMC**
- **Composition Technique:**

According with the notion of equality of programs, two programs are equals if for all initial states, the execution of both result in the same final state.

This composition technique its 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.

As the name space of both are 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.

Composition Technique

So we can do something like this :

```
int main(){

 // Initial Program received in the input file

// Optimized Program after the execution of the application

assert( … );
}
```

On the assert content we have to say that each variable value in the first program must be equal to the renamed correspondent variable value in the second program.
Now to prove this equality we have to appeal to a verification tool like CBMC.

Example

We can see the following example of a cycle :

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

Which can be refactored to this equivalent code:

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

Example

So the compound code could be something like this :

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

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

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

}
```

Conclusions

In the case of the first step of this project relative to the transformation process:

- The transformation is made according to all specifications provided by Efacec;
- The application interface its simple and functional, and already set for future work;
- 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;

Conclusions

- 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 is add this "manually work" to our application, 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 able to 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;

Optimization of C code for Critical Systems

"Technology that moves the world "

Helder Afonso – João Cruz

University of Minho
Master in Computing Engineering
Formal Methods in Software Engineering
Project Presentation : Final Milestone