Architecture document - Version 1

Zohour Abouakil
Sofia Boutahar
David Courtinot
Xiaowen Ji
Fabien Sauce

Reference : model-checking.archi
February 12^{th} 2015

Signatures

Project manager - Zohour Abouakil :
Quality responsible - David Courtinot :
Customers - David Doose - Julien Brunel :

Contents

Chapter I

Context

I.1 Objectives and motivations

As everyone knows, embedded systems are most often critical systems and must be as robust as possible to avoid critical failures which could have dramatic consequences. Hence, many researches are done in order to build tools that would help to ensure the good properties of an embedded system source code and compensate potential human failure. The model checking, which consists in asserting properties on a model thanks to graph search algorithms (for example), is one of those fields that can be applied to this matter. In this project, we are trying to build a model checker working on C++ code which takes the source code as an input and is transformed a few times in various abstract representations to end with a graph model that we are able to send to a model checker.

I.2 Definitions

I.2.1 AST - Abstract Syntax Tree

The AST is an abstract (and low-level) representation of the code. It is a tree data-structure which describes the code in a purely syntactic point of view. As an example, you can see below a simple C/C++ code and its AST representation. The AST is provided by the Clang API, which performs the first step of our transformation chain.

```
1  void fun(int &a) {
2      ++a;
3  }
4
5  int main(int argc, char* argv[]) {
6      int num = 10;
7      if (num > 5)
8          fun(num);
9
10     return 0;
11 }
```

```
TranslationUnitDecl 0x1030218d0 <<invalid sloc>>
|-TypedefDecl 0x103021e10 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x103021e70 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x103022230 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
|-FunctionDecl 0x103022410 <simple.cpp:242:1, line:244:1> fun 'void (int &)'
| |-ParmVarDecl 0x103022350 <line:242:10, col:15> a 'int &'
| `-CompoundStmt 0x103022500 <col:18, line:244:1>
|   `-UnaryOperator 0x1030224e0 <line:243:5, col:7> 'int' lvalue prefix '++'
|     `-DeclRefExpr 0x1030224b8 <col:7> 'int' lvalue ParmVar 0x103022350 'a' 'int &'
`-FunctionDecl 0x10306ffa0 <line:246:1, line:270:1> main 'int (int, char **)'
  |-ParmVarDecl 0x103022530 <line:246:10, col:14> argc 'int'
  |-ParmVarDecl 0x10306fe90 <col:20, col:31> argv 'char **':'char **'
  `-CompoundStmt 0x103070300 <line:247:1, line:270:1>
    |-DeclStmt 0x1030700d8 <line:265:5, col:17>
    | `-VarDecl 0x103070060 <col:5, col:15> num 'int'
    |   `-IntegerLiteral 0x1030700b8 <col:15> 'int' 10
    |-IfStmt 0x103070290 <line:266:5, line:267:16>
    | |-<<<NULL>>>
    | |-BinaryOperator 0x103070150 <line:266:9, col:15> '_Bool' '>'
    | | |-ImplicitCastExpr 0x103070138 <col:9> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0x1030700f0 <col:9> 'int' lvalue Var 0x103070060 'num' 'int'
    | | `-IntegerLiteral 0x103070118 <col:15> 'int' 5
    | |-CallExpr 0x103070260 <line:267:9, col:16> 'void'
    | | |-ImplicitCastExpr 0x103070248 <col:9> 'void (*)(int &)' <FunctionToPointerDecay>
    | | | `-DeclRefExpr 0x1030701f8 <col:9> 'void (int &)' lvalue Function 0x103022410 'fun' 'void (int &)'
    | | `-DeclRefExpr 0x1030701d0 <col:13> 'int' lvalue Var 0x103070060 'num' 'int'
    | `-<<<NULL>>>
    `-ReturnStmt 0x1030702e0 <line:269:5, col:12>
      `-IntegerLiteral 0x1030702c0 <col:12> 'int' 0
```

Figure I.1 - The AST corresponding to the above code

I.2.2 CFG - Control Flow Graph

The CFG is a graph representing all the possible execution paths (with some restrictions, for example we won't create several nodes for a single expression even if in reality an expression should be a graph). As an example, you can find below the CFG generated by a do while statement in an if statement.

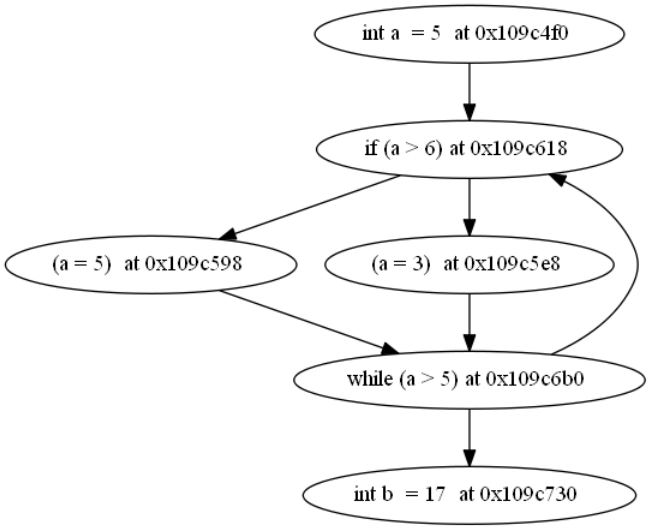

Figure I.2 - An example of CFG

I.2.3 CTL - Computation Tree Logic

CTL is a way of representing temporal logic expressions on a graph or a tree. For example, it can express properties such as *« All the paths starting from every node verifies the predicate p »*.

Chapter II

AST and CFG representations

After studying the Clang API, we came to the conclusion that the AST is a much more low-level representation of the program than the CFG. Indeed, the atom for a CFG is what is generally called a *statement* whereas the simplest instruction gives an AST representation composed of multiple nodes. We also found it difficult to handle the parsing and the linking of the graph nodes at the same time. Thus, we have chosen to transform the AST into a series of higher-level objects than the original nodes, which will be converted in nodes of the CFG.

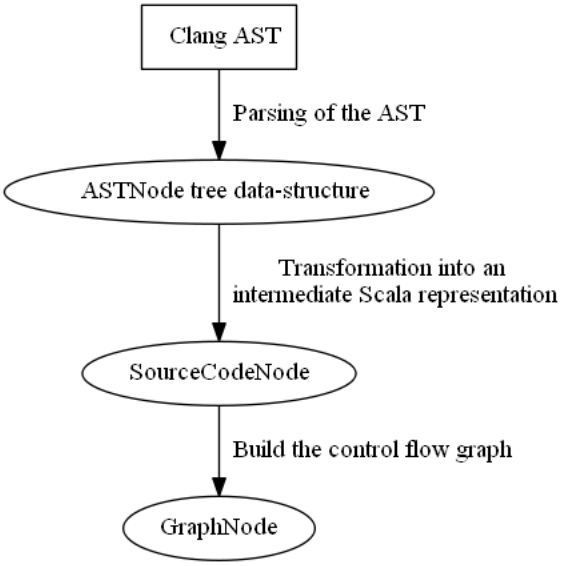

Figure II.1 - Transformation chain we are going to present

II.1 Intermediate representation of the Clang AST

II.1.1 Parsing the Clang AST file

At first, we have considered using XML parsing libraries to parse the XML version of the Clang AST. However, this type of output is no longer supported by the newest versions of the Clang compiler and all the existing tools provide partial support at best. Hence, we decided using the regular AST file and parse it line by line with a custom parser.

We have identified three main kinds of nodes in the AST. Each one is associated to a specific class which extends ASTNode :

- nodes consisting in an type name, an id, a code pointer pointing the relevant lines of the code and some metadata that depend on the type of the node. These are represented by the ConcreteASTNode class.
- < < <NULL> > > children, represented by the NullASTNode class.
- other kind of nodes, prior to class declaration for example. These are represented by OtherASTNode.

The file will be parsed and converted in a tree data-structure which nodes are of type ASTNode. The ASTNode objects will then be converted in Stmt or Decl accordingly to the class hierarchy we present in the next part.

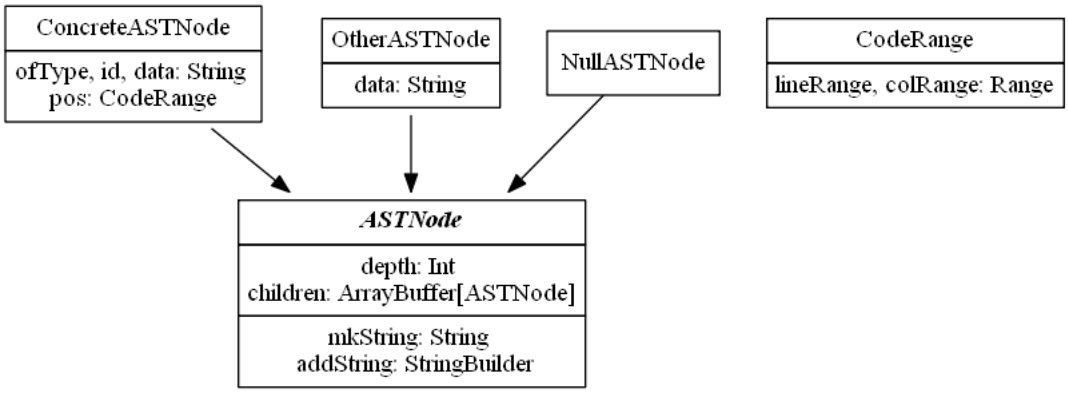

Figure II.2 - Class hierarchy for the output format of the ASTParser

II.1.2 From ASTNode to SourceCodeNode

The SourceCodeNode class represents a tree data-structure which is still close enough to the AST but with a higher abstraction, and some removed low-level information.

Decl class hierarchy

For the Decl part, which represents the different kinds of declarations in the code, we did not have too much trouble and just had to associate each high-level Clang Decl class to a Scala class extending our Decl class, as it is shown in the previous figure.

Stmt part

Stmt is a Clang asbtraction of a statement in a program (any expression, any flow-control structure). As it was not presented as a priority by the client, we have decided to skip the C++ object oriented part in order to focus exclusively on the imperative part. Inspired by the Clang API, we came up with the following class diagram.

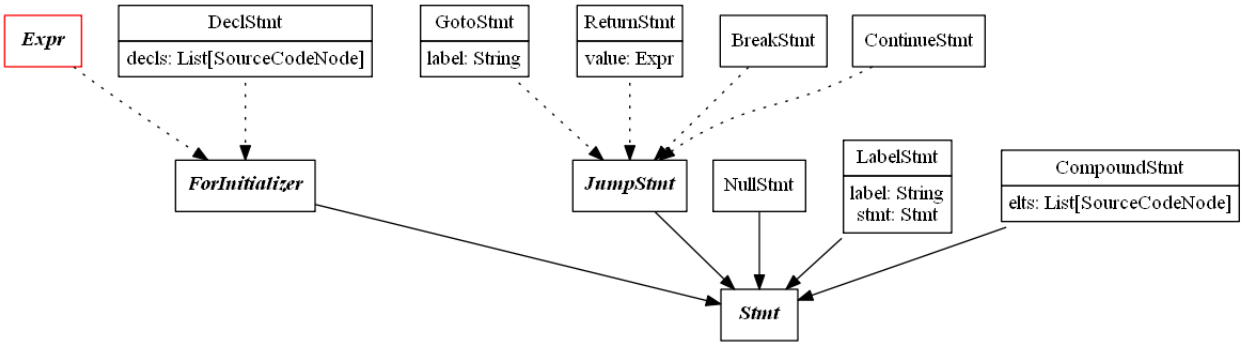

Figure II.3 - Representation of the most classes statements

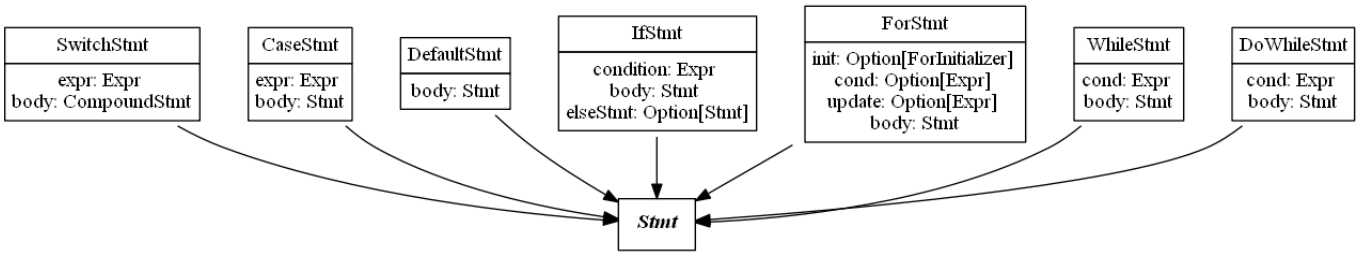

Figure II.4 - Flow-control structures

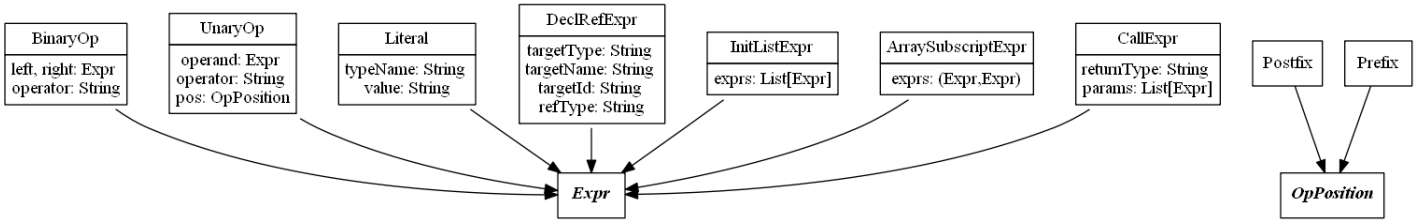

Figure II.5 - Detail of the Expr class hierarchy

Note that our model does not strictly represents a C++ code. For example, we do not prevent a *if* to contain the instruction *break* (among other inaccuracies ...). We felt that this kind of refinement would unnecessarily complicate the task without adding anything more to the CFG analysis. Since the code is already semantically checked by the Clang compiler and given our future needs, we thought it would be wiser to aim for a simple model.

Important notes

- To accurately represent the CFG of our input programs, we should take into account the fail-fast mechanism in the evaluation of boolean conjunctions/disjunctions. The importance of this mechanism for our project is illustrated in the figure below.
- However, since the evaluation's order of the expressions is not completely specified specified in C++ (unlike Java which evaluates from left to right), we will ignore that even if it will surely change the result for certain kind of treatments when some expressions contain side-effect sub-expressions (increment, assignments...).

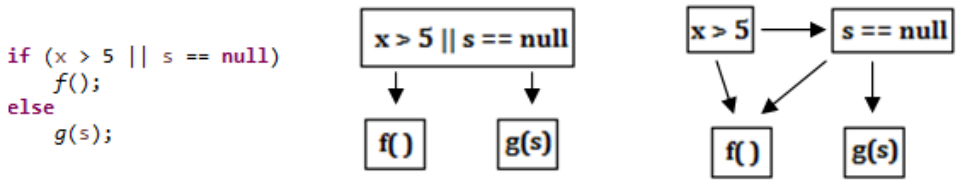

Figure II.6 - On the left the considerate code. On the center, we didn't take into account the fail-fast for the CFG, and finally on the right we consider the fail-fast

Just to see what we are losing by ignoring that kind of things is illustrated in the figure above : if we try to assert the property *« variable s is always initialized when g(s) is called »*, the first CFG will not allow us to conclude (or it will conclude **true**, erroneously) while the second allows us to state that there are executions, according to the value of x, where *g* is called with an uninitialized parameter (assuming that the left son is always a successful test and the right son is a failed test). It is not really a problem, as we could argue that it is fine to use fail-fast most of the times but that it is safer not to use it for a critical program.

Another important thing to mention is that at the beginning, we had a mix of regular classes with var/val members and case classes. Moreover, we were using «var» immutable lists (class List) for building incrementally some collections as the list of statements in a compound statement. However, it would imply to append at the end because otherwise, the list would have been to be reversed each time needed. Therefore, we have created a simple wrapper of ArrayBuffer, that hides all the ArrayBuffer definitions except + =, and otherwise behaves as an immutable List. Same thing was done with a wrapper of a mutable HashMap (for storing declaration by their name for example). This enables to use a more functional programming style while being efficient.

Finally, we chose to make all the classes as case-classes to enable the powerful Scala pattern-matching. Most algorithms are recursive, the parsing of the AST being the only exception.

II.2 SourceCodeNode to CFG

Considering that Stmt and Decl children classes were partly low-level elements of the code that are not important in the CFG, we decided to perform a last step of transformation from SourceCodeNode to ProgramNode, which is a simpler and higher level abstraction of the code. The ProgramNode objects will be the values of the nodes of the actual graph, represented by the class GraphNode[T].

GraphNode is actually a generic type, completely independent of all the classes we introduced so far. It basically represents any oriented unweighted graph. The conversion from SourceCodeNode to ProgramNode is handled on the fly while constructing the CFG (GraphNode[ProgramNode]), which consists in creating the links between the various nodes of the graph so that they enable to explore the various possible execution paths.

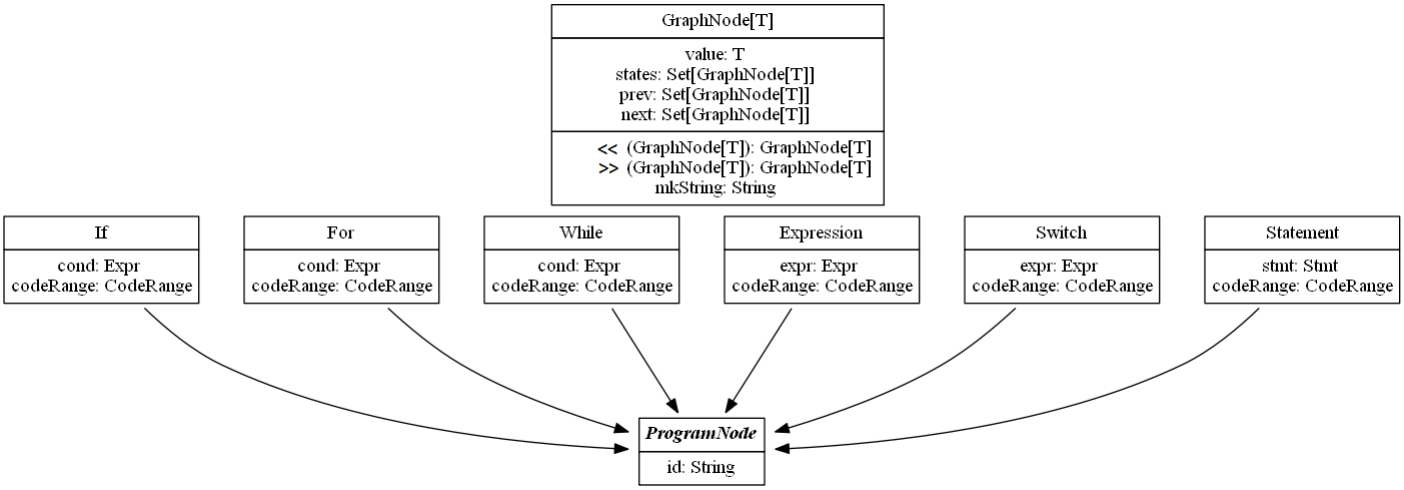

Figure II.7 - ProgramNode class hierarchy

Chapter III

Model checking

III.1 Representation of the environments

III.1.1

III.2 CTL expressions

III.2.1 Defining the CTL-V language

III.2.2 Evaluation of the predicates