Design review - Iteration 1 - Version 2

Zohour Abouakil
Sofia Boutahar
David Courtinot
Xiaowen Ji
Fabien Sauce

Reference : model-checking.design
January 30th 2015

Signatures

Quality responsible :
Clients :

Contents

Partie I

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 programme 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 or supernodes containing a subgraph.

I.1 Intermediate representation of the Clang AST

I.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.

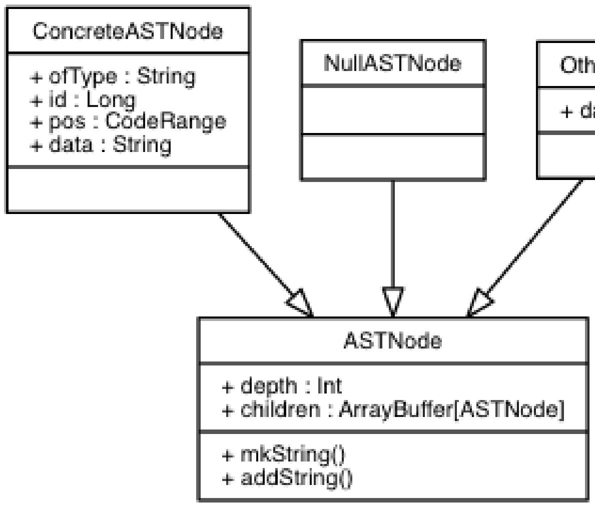


Figure I.1 - Inheritance relationships between the classes used to represent the AST

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.

I.1.2 From ASTNode to ProgramNode

Decl class hierarchy

For the Decl part, 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.

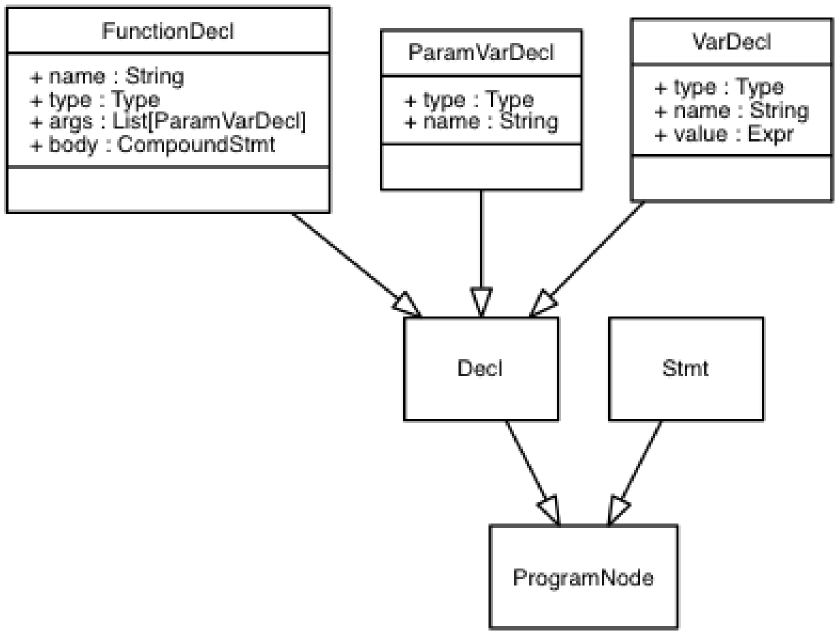


Figure I.2 - Class diagram of the imperative part of Decl

Stmt part

For this first iteration, 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 :

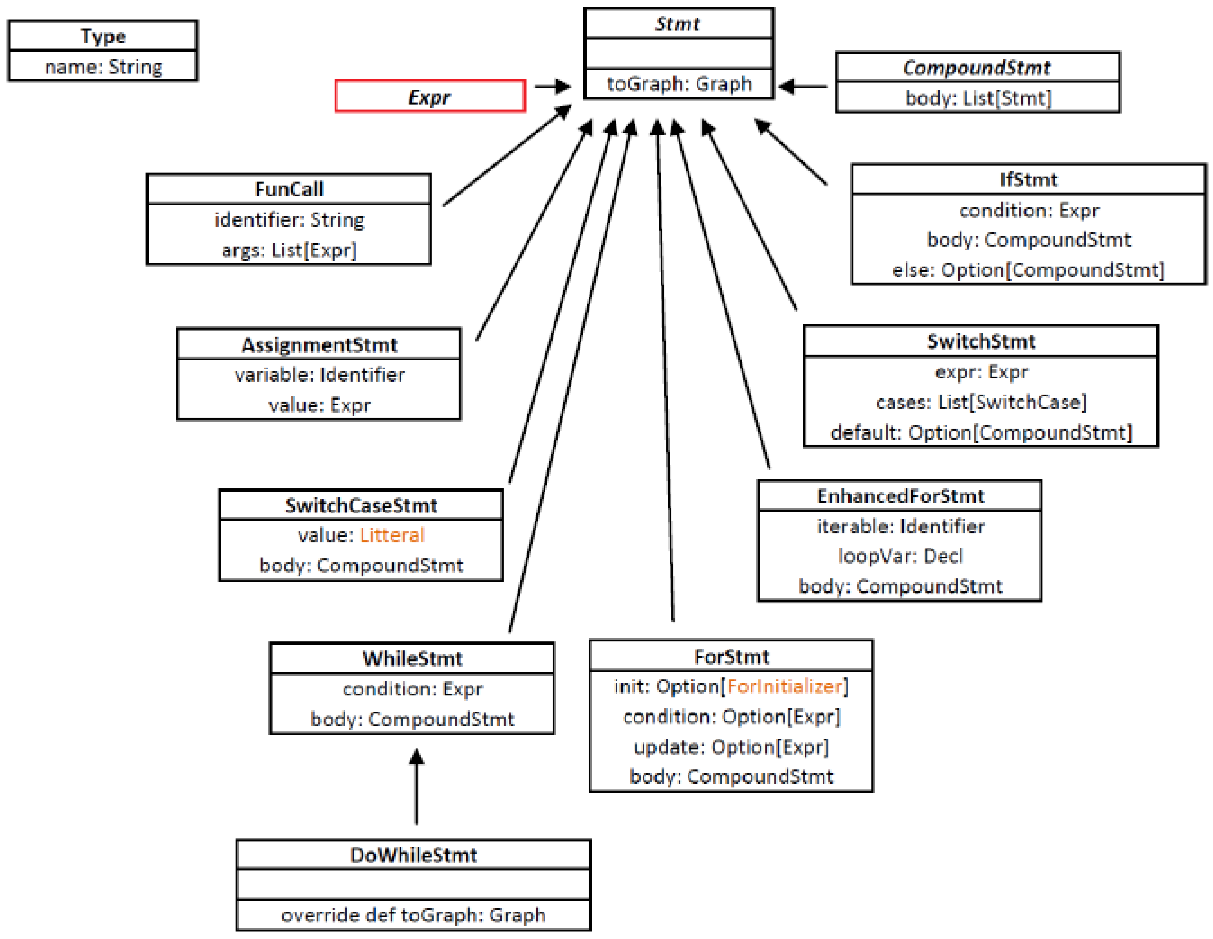


Figure I.3 - Class diagram of the imperative part of Stmt

Note that our model does not strictly represent 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 checked semantically by the Clang compiler and given our future needs, we thought it would be wiser to aim for a simple model. Moreover, some classes have not been totally defined in the previous diagram because we deliberately chose to represent them apart on the diagram below. Finally, other classes were not represented (BreakStmt, ContinueStmt for a better readability).

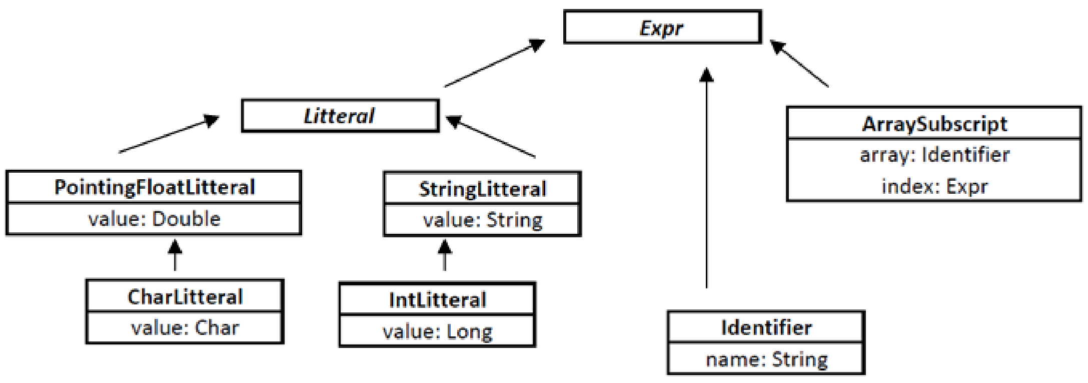


Figure I.4 - Partial class diagram of the **Expr** part

Finally, we present a more complete diagram for Expr even though we still do not know if we will keep it that way, its complexity being considered superfluous for the nature of the algorithms we will use on these structures.

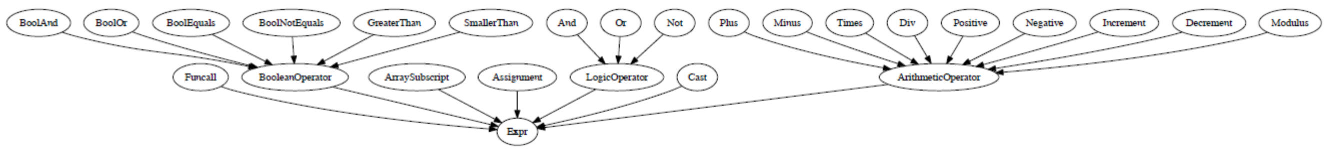
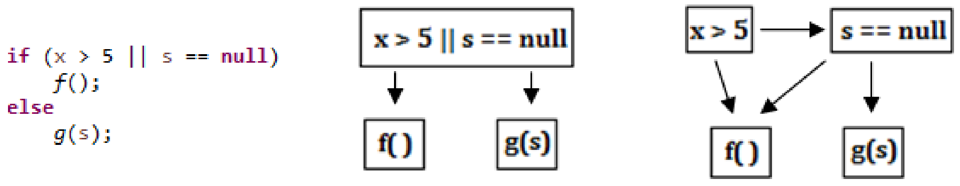


Figure I.5 - Class diagram heavier (still incomplete) for the **Expr** part

Important notes

- to accuratelys represent the CFG of our input programs, we must 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 limit these considerations to simple expressions. Therefore we will not take care of expressions (boolean or not) containing sub-expressions side effect (increment, assignments...).



```
if (x > 5 || s == null)
    f();
else
    g(s);
```

Figure I.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

As shown 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).

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. Hence, it appeared a more flexible and clever conception to separate the Stmt/Decl classes from their actual equivalent in the code, which is a ProgramNode. The ProgramNode contains the information of the position of the Stmt/Decl in the code thanks to the CodeRange class. As we were using case-classes, we could not inherit some constructor from ProgramNode that would initialize the CodeRange, so we just made the CodeRange optional and added a setter. It seems to be poor design at first sight but it was the best way to keep the advantages of both the pattern-matching and the factorization of an attribute on an abstract superclass.

I.2 AST to CFG

The class GraphNode is used to represent a node of the CFG, it is still not clearly defined as we are missing the operations required on this class for the model checker. Nevertheless, we can already be sure that a GraphNode will wrap a ProgramNode value (most certainly only certain kind of ProgramNode such as DeclStmt, Assignment and so on, and never CompoundStmt or classes containing an inner CompoundStmt instance). It will also provide a list of predecessors and a list of successors so that we can explore the graph.

Finally, we have thought about the problem of the label function. We must be able to labellize many different kinds of nodes with any kind of predicate. We do not want that adding a new predicate require rewriting the ProgramNode children classes, so we opted for a Visitor pattern. Thus, adding a new Label will only consist in writing a class extending the Labellizer trait.

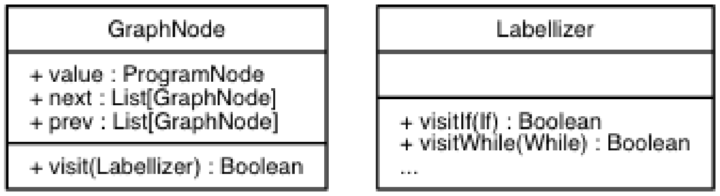


Figure I.7 - CFG class diagram

Partie II

Model checking