Design review - Iteration 1 - Version 2

Zohour Abouakil
Sofia Boutahar
David Courtinot
Xiaowen Ji
Fabien Sauce

Reference : model-checking.design
January 30^{th} 2015

Signatures

Quality responsible :
Clients :

Sommaire

Partie I

AST and CFG representations

After studying API Clang, we came to the conclusion that the AST is made of over nuances that we needed to build the CFG. Indeed, the atom for a CFG is what is generally called a textit statement whereas the simplest instruction gives an AST representation composed of multiple nodes. We also found difficult to handle both parsing and binding of the graph nodes. That is why we have chosen to transform the AST into a series of high-level objects that its original nodes, which will be at places blocks CFG, or macro-blocks containing a subgraph.

I.1 Transitional representation of the Clang AST

I.1.1 Parse the Clang AST file

I.1.2 From ASTNode to ProgramNode

Decl class hierarchy

Partie Stmt

For this first iteration, we skip the C++ object part in order focus exclusively on the imperative part Inspired by the Clang API, we came to the following class diagram:

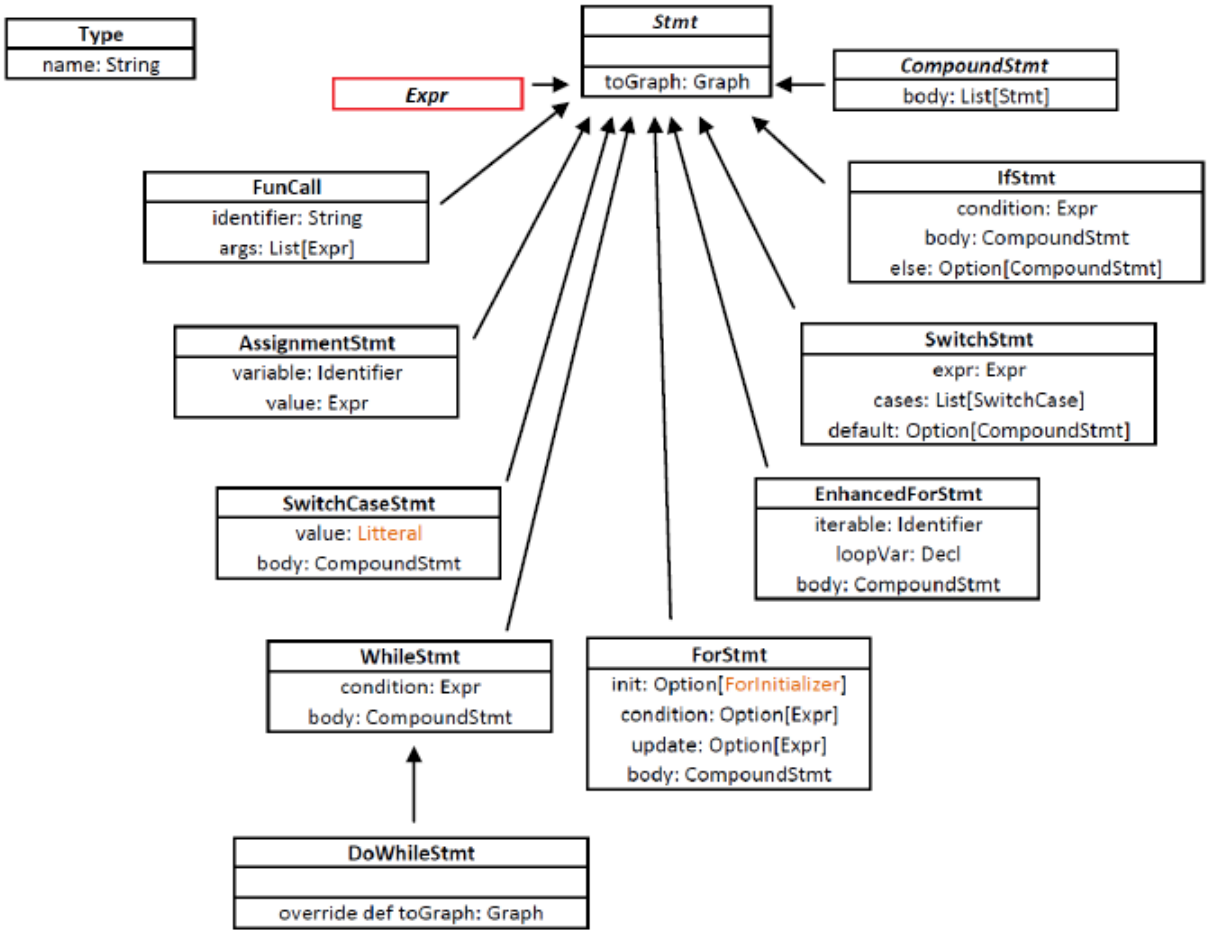

Figure I.1 - Class diagram from the imperative part of **Stmt**

Note that our model does not strictly represent a C++ code. For example, we do not prevent a *if* of containing 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 clearly 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).

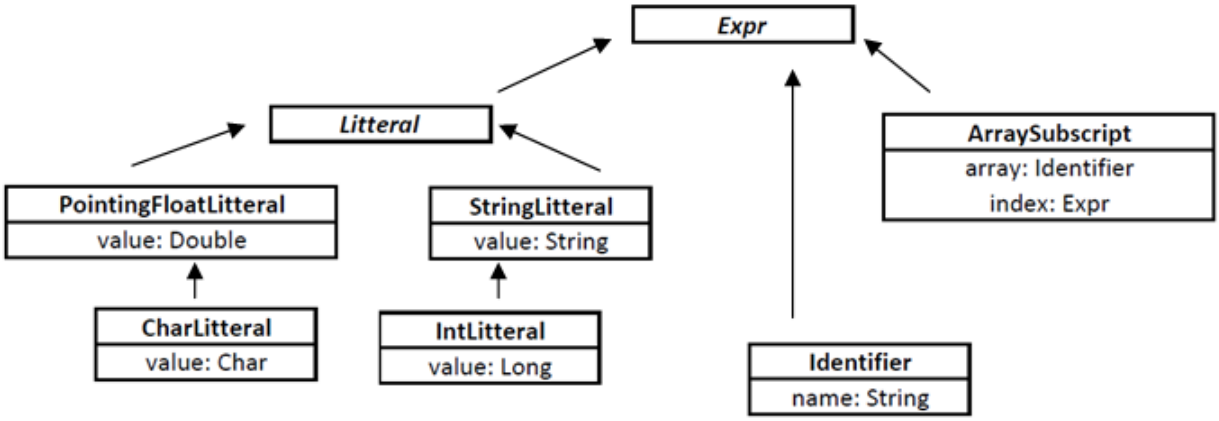

Figure I.2 - Partial class diagram of the **Expr** part

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

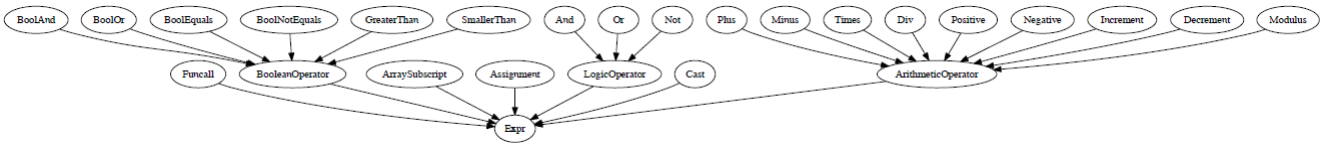

Figure I.3 - Class diagram heavier (still incomplete) for the **Expr** part

Important observations:

- To faithfully represent the CFG of our programs, we must take into account the fail-fast mechanism in assessing conjunctions / disjunctions. The importance of this mechanism for our project is illustrated in the following figure .
- However, since the evaluation's order of the expressions is not 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 edge effect (increment, assignments...).

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

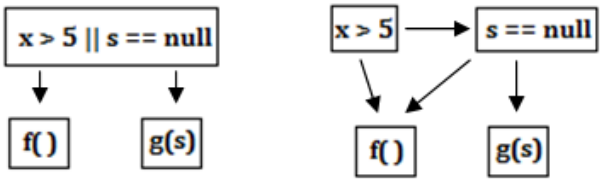

Figure I.4 - On the left the considerate code. On the center, we didn't take into account the fail-fast for the CFG, and finally on the right we consider the fail-fast

As it can be seen in the figure above, if we try to insure the property *og s is always initialize when g(s) is called fg*, the first CFG will not allow us to conclude ă (or it will conclude **true**, wrongly) while the second allows us ă to say that there are executions, according to the value of a, 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)

I.2 AST to CFG

Partie II

Model checking