User guide - Version 1

Zohour Abouakil
David Courtinot

Reference : model-checking.user-guide
March, 6^{th} 2015

Signatures

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

Contents

Part I

Provided features

In this guide, we will keep referring to our project that can be found on Github : *https://github.com/jxw1102/Projet-merou*. In this part, we will describe the packages composing our project, their role and mutual dependencies as well as the other provided scripts or documentation.

I.1 Packages overview

Packages description

The packages we are providing are listed below :

- **ast :** contains the classes used to parse the Clang AST and convert it into a graph
  - **ast.model** : contains our model of the internal representation of the C++ language
  - **ast.test** : contains some semi-automated tests for the AST parsing and conversion to CFG
- **graph** : contains a single class, GraphNode, representing an oriented unweighted graph
- **ctl** : contains our internal representation of CTL expressions as well as the implementation of a model-checking algorithm, in the ModelChecker class
  - **ctl.test** : contains fully automated several test classes unitary testing the classes of ctl, independently of the CFG
- **cfg** : contains our implementation of the base predicates to be used in CTL expressions in the specific case of the CFG
  - **cfg.test** : contains a test file executing the properties defined on cfg.Properties. Those two files are complementary with this user guide and provide some examples and use-cases of our application
  - **cfg.parser** : contains the definition of a small language aiming to represent CTL expressions in the particular case of the CFG in a textual format.

    Note : the CTL grammar is defined in the same file as some elements specific to CFG, which is a pity. A better design, which we will not implement by lack of time, but should be easy to do with the

current code, would be to define the CTL grammar in a class in ctl.parser. Then, this class would be extended by another class in cfg.parser to add the CFG specific features.

Package dependencies and CLASSPATH

The following scheme is self-explanatory :

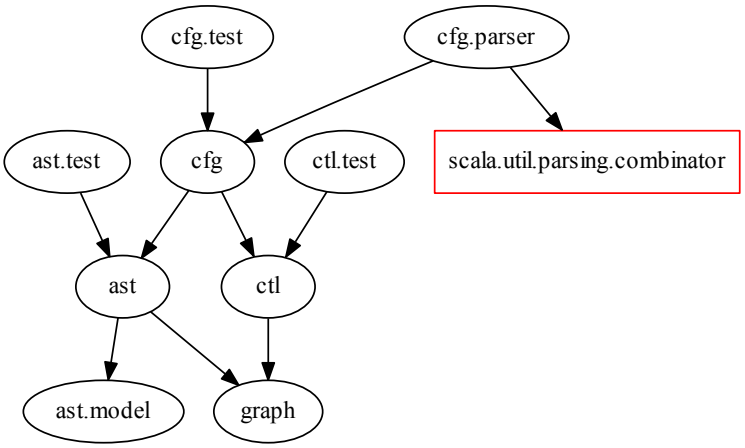

Figure I.1 - Package mutual dependencies in our project. In red are shown the CLASSPATH dependencies.

I.2 Release options and documentation

I.2.1 Release script

We provide to the user a Python script named *make_packages.py*, which can be found in *Projet-merou/ModelChecker*. This script enables to package our application according to your needs. The documentation of the command should get you to understand its options :

```
1  This command packs the relevant binary files in a jar. The output jar is stored in the release folder.
2
3  Syntax : make-package release-name [options]
4
5  release-name can be any of the following :
6         . -ctl-only : packs ctl-related classes only
7         . -ast-only : packs ast-related classes only
8         . -full : packs the whole application
9
10 -----------------
```

```
Options
-----------------

. -src : adds the corresponding source code to the archive
```

The output jar is generated in *Projet-merou/ModelChecker/release*, and the script working directory must be *Projet-merou/ModelChecker*.

I.2.2 Documentation

The scaladoc of the project can be found in *Projet-merou/ModelChecker/doc.jar*. For the most curious, *Projet-merou/docs/Architecture document* contain all the versions of our architecture (some of them may not be complete, just consult the latest version). Please note that it deeply dives into the implementation details and is not necessary to use the application.

Part II

How to construct a CTL expression

A CTL expression is a combination of predicates - elementary properties - and temporal connectors as AX, AF. Temporal connectors are implemented as they are defined in the provided papers. In total, the following connectors are available (some as classes, the others as implicit declarations in the ctl package object) :

- **AX**, **EX :** defined as a case class named AX (resp. EX) and taking a CTL expression as a parameter
- **AF :** defined as an implicit declaration named AF taking two CTL expressions as a parameter. What is usually noted A[p F q] becomes AF(p,q).
- **EF**, **AU**, **EU :** same as AF
- **conjuction**, **disjunction :** defined like a case class And (resp. Or) taking two CTL expressions as a parameter. It can be used with And(p,q) (resp. Or(p,q)) or p && q (resp. p || q) thanks to some syntactic sugar.
- **negation :** defined as a case class Not taking one CTL expression as a parameter. It can be used with Not(p) or !p thanks to some syntactic sugar.

A predicate is an atomic CTL expression that expresses properties like « *This node is a function call with one parameter* » or « *This node is an if statement condition matches the X > Y pattern* » (we will soon detail how patterns work). Predicates are evaluated on every node of the graph. More precisely, the evaluation of a property on a node is actually done by a Labelizer, wrapped by a Predicate. For example, we would write the second example as `Predicate(IfLabelizer(BinaryOpPattern(...)))`.

II.1 Patterns

We brielfy introduced the notion of a *labelizer* as well as the notion of a *pattern*. Most of the time, a labelizer (extending the Labelizer abstract class) takes a pattern (extending the Pattern trait) as a parameter. Therefore, let's first what pattern are done for and how to use them. A pattern represents a basic syntactic property of the code. In terms of logic, it can be considered as a predicate but please note that a Pattern is never a Predicate in our program, as it is missing some semantic information (we will detail it in further details later). As an

example, X + 1 is a pattern describing certain expressions. By convention, we will use upper-case letters for the meta-variables, a lower-case word will designate an identifier in the code. A good comparison to get the idea is the regex `(?<X>.+)\+1` (using the Java capturing named groups (?<name>regex)). This regex accepts any string containing a `'+'` character with a non-empty substring at its left and a `'1'` at its right. Moreover, thanks to the named groupe X, it enables to extract the value of the right member. This is also the behaviour of our patterns :

- check that the "regex" is matched
- return the "groups", namely the bindings between the meta-variables of the pattern and the corresponding matched values. This is a bit more subtle here, because the bindigs have to be compatible, but the basic idea is the same

We will now detail the architecture of the Pattern type, summed up by the following class diagaram (leaves classes are not presented on it, we just kept the traits and abstract classes) :

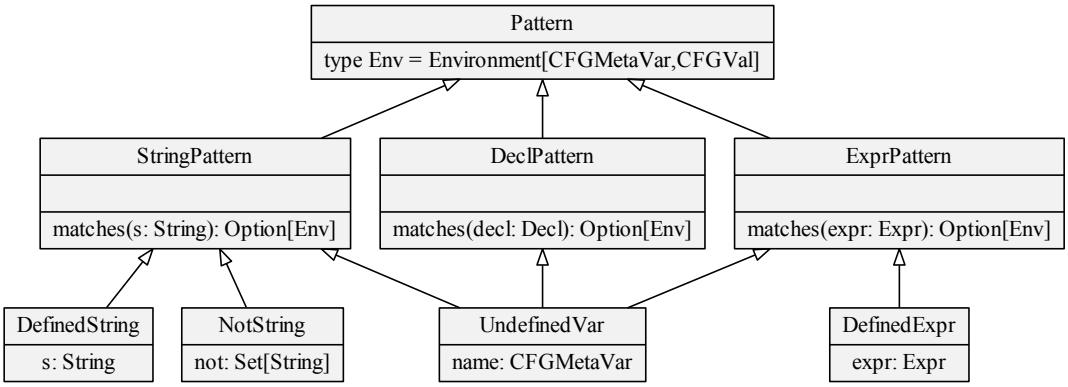

Figure II.1 - Architecture of the patterns

II.1.1 Atomic patterns

Different types of atomic patterns

All the leaves of the above class diagram are considered atomic patterns. As you can see, there are three kinds of atomic patterns :

- defined values : it concerns DefinedExpr and DefinedString. Use defined values when you want to see specific values in the code entity (declaration or expression) your are trying to match with a pattern. There will be pratical examples of it later.
- forbidden values : it concerns NotString. Use it when you want to disallow certain values in the entity you are matching.

- meta-variables : it concerns UndefinedVar. It is the only atomic pattern that returns a (positive) binding in case of a successful match. UndefinedVar takes a meta-variable as a parameter, which will be the meta-variable involved in the eventually returned binding. Finally, please note that this is, to keep with the regex metaphor, equivalent to (?<metavarName>.*). To be clear, it means that it matches anything with no constraint and captures the whole entity as a group.

Use-cases and examples

DefinedString You can use it when you want to get the definition of all the variables of a specific type (let's say int), which will be noted `DefinedSring("int")`. This will have to be composed with a VarDefPattern, as we will see it.

NotString You can use it when you want to get the function calls which return type is different from `void` or `void *`, which will be expressed with `NotString("void","void *")`. This will have to be composed with a CallExprPattern.

DefinedExpr You can use it if you want to find all the expressions using a variable called x : `DefinedExpr(DeclRefExpr("","x","",""))`. **Be careful**, the match operation on a DefinedExpr will always ignore the type of the expression as well as the Clang id. Consequently, it is for example impossible to find all expressions using a variable called x of type `double`.

UndefinedVar Use it when you want to return a binding between a meta-variable and the value of a "matched group" (cf regex).

II.1.2 Composed patterns

Currently, all the classes extending StringPattern are atomic patterns, nevertheless ExprPattern is extended by other classes taking some ExprPattern(s) as parameter(s), which makes it a recursive type. DeclPattern is not a recursive type, but we will see it is also extended by composed patterns. For now, let's just focus on the recursive expression patterns.

ExprPattern : recursive patterns

The recursive expression patterns are listed below :

CompoundAssignOpPattern, BinaryOpPattern and UnaryOpPattern These patterns respectively match CompoundAssignOp, BinaryOp and UnaryOp expressions.

- **Syntax :**
  - CompoundAssignOpPattern(left: ExprPattern, right: ExprPattern, op: StringPattern=NotString())
  - BinaryOpPattern(left: ExprPattern, right: ExprPattern, op: StringPattern=NotString())
  - UnaryOpPattern(operand: ExprPattern, op: StringPattern=NotString())

op is a StringPattern specifying the pattern you want to use for matching the operator, so you can specify a specific operator you want to see, a set of operators you do not want to see or an UndefinedVar if you want to retrieve the operator in the environment in case of a successful match. If you have no specific constraint on the operator and don't want to introduce it in the environment, use the default value NotString() (which expands to NotString(Set())).

- **Example :**

`BinaryOp(Literal("int", "3"), Literal("int", "2"), "==")` matches `BinaryOpPattern(UndefinedVar("X"), Literal("int", "2"), "==")` and returns the environment { X → `CFGExpr(Literal("int", "3"))` }.

AssignmentPattern This pattern is a convenient pattern to represent assignments. It basically uses CompoundAssignOpPattern and BinaryOpPattern.

- **Syntax :** AssignmentPattern(left: ExprPattern, right: ExprPattern, op: StringPattern=NotString()).

  If op is a DefinedString, passing any value other than `"="`, `"+="`, `"-="`, `"*="`, `"/="` will cause an exception to be thrown.

- **Examples :**
  - `BinaryOp(Literal("int", "3"), Literal("int", "2"), "==")` does not match `AssignmentPattern(UndefinedVar("X"), Literal("int", "2"),NotString())`
  - `CompoundAssignOp(DeclRefExpr("","x",""), Literal("int", "2"),"+=")` does not match `AssignmentPattern(UndefinedVar("X"), Literal("int", "2"),NotString("+="))`
  - `BinaryOp(Literal("int", "3"), Literal("int", "2"), "=")` matches `AssignmentPattern(DefinedExpr(Literal("int", "3")),DefinedExpr(Literal("int", "2")),DefinedString("="))` and returns the ⊤ (top) environment

CallExprPattern It matches a function call (CallExpr) and takes three arguments :

- **Syntax :**

  CallExprPattern(name : StringPattern, params: Option[List[ExprPattern]]=None, typeOf: StringPattern=NotString())
  - name : the name of the function
  - params : is the parameters list. Using value None will make the CallExprPattern ignore the parameters, otherwise each parameter will have to match.
  - typeOf : return type of the function

- **Examples :**
  - `CallExpr("int",DeclRefExpr("int","f","0x589f65"),Literal("int","3"))` (which simply corresponds to $f(3)$ with f of type int) matches `CallExprPattern(DefinedString("f"),Some(List(DefinedExpr(Literal("int", "3")))), NotString())` and returns ⊤
  - `CallExpr("int",DeclRefExpr("int","f","0x589f65"),Literal("int","3"))` does not match `CallExprPattern(UndefinedVar("X"),Some(List()), NotString())`
  - `CallExpr("int",DeclRefExpr("int","f","0x589f65"),Literal("int","3"))` matches `CallExprPattern(UndefinedVar("X"),None,NotString())` and returns { X → `CFGString("f")` }.

Some other patterns have recently been implemented, they correspond to more specific situations (malloc, delete, new...) and work the exact same way. You can take look at their documentation to know their semantic, but we will not talk about them extensively in this guide.

DeclPattern : non-recursive composed patterns

The difference here with ExprPattern is that our subclasses of DeclPattern never take a DeclPattern as a parameter. It could make sense to do it for struct or classes but for now we just implement patterns for variables, therefore it would be absurd to make the type recursive as a variable declaration cannot contain another declaration. We have implemented two variable declaration patterns :

- VarDeclPattern(typeOf: StringPattern, name: StringPattern)
- VarDefPattern(typeOf: StringPattern, name: StringPattern)

In both cases, *typeOf* designates the type of the declared variable and *name* its identifier. The only difference between those two patterns is the type of value they introduce in the environment if *name* is an UndefinedVar : in the first case, it will return a CFGDecl, in the other case it will be a CFGDef. CFGDecl(s) (as in declaration) are used when you want to base the equality between declarations on their Clang ID, which means that each declaration in the code is unique. Using CFGDef (as in definition), declarations will be considered as the tuple (type,name), which means two distinct declarations in the code may be equal in that case. We will see the interest of this subtle difference when we will define some complicated properties. For now, let's just consider some examples to make it clear :

- `VarDecl("x","int",None)` (which corresponds to `int x` in C++) matches `VarDeclPattern(DefinedString("int "),UndefinedVar("X"))` and returns { X → `CFGDecl(id,"int","x")` } where id is the Clang id of the VarDecl
- `VarDecl("x","int",None)` (which corresponds to `int x` in C++) matches `VarDefPattern(DefinedString("int "),UndefinedVar("X"))` and returns { X → `CFGDef("int","x")` }

II.2 Labelizers

As we said it before, a Labelizer most of the time wraps a Pattern adding to it a specific semantic. A Labelizer is applied on a node of the CFG, which can be a For(expr: Expr), While(expr: Expr), Switch(expr: Expr), If(expr: Expr), Expression(expr: Expr), Statement(_) (it is not mportant in this guide to explain what a Statement contains). Before listing our labelizers, we will detail the semantic of these nodes :

- For(Option[expr]) : corresponds to `for( ... ; expr? ; ... )` in the source code
- While(expr) : corresponds to `while (expr)` in the source code
- If(expr) : corresponds to `if (expr)` in the source code
- Switch(expr) : corresponds to `switch (expr)` in the source code
- Expression(expr) : corresponds to `expr` in the source code (an assignment for example)
- Statement(expr) : corresponds to any other kind of statement. Most of the time, it will be a declaration.

Now, here are listed the labelizers we implemented :

IfLabelizer, WhileLabelizer, SwitchLabelizer, ForLabelizer and ExpressionLabelizer Those labelizers test the type of the node and additionally try to match the expression it contains with an ExprPattern taken as a parameter.

- **Syntax :**
  - IfLabelizer(pattern: ExprPattern)
  - WhileLabelizer(pattern: ExprPattern)
  - SwitchLabelizer(pattern: ExprPattern)
  - ForLabelizer(pattern: Option[ExprPattern])
  - ExpressionLabelizer(pattern: ExprPattern)
- **Note :** ForLabelizer is a bit particular as it takes an ExprPattern as a parameter. It is simply to handle the case when there is no condition in the *for* statement. In this case, *pattern* has to be None.

FindExprLabelizer, MatchExprLabelizer Those two labelizers ignore the type of the node and simply focus on the expression it contains, if any. If the given pattern is matched, those labelizers could accept For, While etc. and even Statement nodes (indeed, a variable declaration can contain an assignment). The difference between them is that Find will try to find a match in every sub-expression of the input expression whereas the Match will stop at the root of the expression. It is comparable to `"input string".contains("string")` and `"input string".matches("string")` in Java : the first statement will return *true* because it is equivalent to the `.*string.*` regex, whereas the second one is equivalent to `^string$` and will return *false*.

- **Syntax :**
  - FindExprLabelizer(pattern: ExprPattern)
  - MatchExprLabelizer(pattern: ExprPattern)
- **Note :** the FindExprLabelizer may return several results if the pattern has several occurrences in the input expression, whereas a MatchExprLabelizer will return at most one result.

UseLabelizer This labelizer returns all the variables (in form of a CFGDecl) which value is used in a node

- **Syntax :**
  - UseLabelizer(pattern: StringPattern)
- **Examples :**
  - the code `x = z + y + 3` will produce an Expression node that will be accepted by `UseLabelizer(NotString())` and will return $\top$
  - the code `x = z + y + 3` will produce an Expression node that will be accepted by `UseLabelizer(UndefinedVar("X"))` and will return (the ids and types are invented for the example) Set({ X → `CFGDecl("0x256f4a0","float","z")` }, { X → `CFGDecl("0xbc450d","int","y")` })
  - the code `x = z + y + 3` will produce an Expression node that will be accepted by `UseLabelizer(DefinedString("y"))` and will return $\top$

VarDeclLabelizer and VarDefLabelizer Those labelizers only recognize Statement nodes containing a variable declaration (VarDecl). The difference between them is the same as between VarDeclPattern and VarDefPattern.

- **Syntax :**
  - VarDeclLabelizer(pattern: VarDeclPattern)
  - VarDefLabelizer(pattern: VarDefPattern)