"Projet Long" report - Version 1

Zohour Abouakil
Sofia Boutahar
David Courtinot
Xiaowen Ji
Fabien Sauce

Reference : model-checking.final-report
March, 2nd 2015

Signatures

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

Contents

I Project presentation **4**
I.1 Overview 4
I.2 Subject 4
I.2.1 Main idea 4
I.2.2 Project description 4
I.3 Objectives 5
I.4 Constraints 5

II Project management **6**
II.1 Team organization 6
II.2 Project objectives in terms of management and organization 7
II.3 Deliverable documents 8
II.3.1 Delivrable documents expected by ENSEEIHT and the industrial supervisor 8
II.3.2 Delivrable documents expected by the client 8
II.4 Development plan 8
II.4.1 Development organization 8
II.4.2 Team organization approach for coding 8
II.4.3 Tasks organization 8
II.5 Risk management 9
II.5.1 Risk management strategy 9
II.5.2 Risk analysis 9
II.6 Resource management system 10
II.6.1 Versioning tool 10
II.6.2 Communication between team members 10
II.7 Code and documentation 11
II.7.1 Quality checking 11
II.7.2 Verification and validation process 12

III Technical aspects **13**
III.1 Context 13
III.1.1 Motivations 13
III.1.2 Objectives 13
III.2 Definitions 13
III.2.1 The AST - Abstract Syntax Tree 13
III.2.2 CTL - Computation Tree Logic 13
III.3 Model checking 13
III.3.1 Representation of the environments 14
III.3.2 CTL expressions 15
III.3.3 Model checker 16
III.4 Merge 17
III.4.1 Choice of M, N and V 17
III.4.2 Labelize our nodes 18

Acknowledgments

Apart from our personal efforts, the success of any project largely depends on the encouragements and guidelines of many other persons either from the clients or our school professors. We would like to express the deepest appreciation to all our professors at ENSEEIHT who introduced us to Computer Science Engineering and helped us to acquire solid technical skills and widen our knowledge by attending interesting courses and working on many exciting projects during those three years.

We take this opportunity to express our gratitude to the people who have been instrumental in the successful completion of this project. We would like to gratefully acknowledge the enthusiastic supervision of Mr. David Doose and Mr. Julien Brunel, both engineering researchers at ONERA, who had been a source of inspiration and were always guiding us and giving us useful suggestions which helped us in completing the project work. They always did their best to respond promptly and enthusiastically to all our requests, despite their congested daily schedule. I would also like to thank Mr. Jean francois Coiffin, our industrial partner, who helped us to manage the project, raise awareness with problems and methods for this activity and also ensure quality control. Finally, the reception of ENSEEIHT, which provided us every day with a room to work in all day long.

Introduction

This report deals with the work we did from the end of January to mid-March, in the context of the "Projet long". It is proposed by Mr. David Doose and Mr. Julien Brunel from ONERA, the French Aerospace Lab, and deals with pattern recognition in C++ code.

Our team consists in five ENSEEIHT students (Zohour Abouakil, Fabien Sauce) from the computer science course and (Sofia Boutahar, David Courtinot, Xiaowen Ji) from the imagery and multimedia course, two customers (Mr. David Doose and Mr. Julien Brunel) and an industrial partner from Astrium (Mr. Jean Francois Coiffin)

As third year engineering students in Computer science and applied mathematics, we are interested in ground-breaking technologies. Part of our degree, our final year project has been the right place to get in touch with a lot of new technologies and get in touch with highly skilled and professional persons by working on an innovant and ambitious project.

It was the opportunity to discover and set up project management systems that are necessary to respect the deadlines. We will now describe the project goals and management methods that we used and finally we will present the technical aspects of our project.

Part I

Project presentation

I.1 Overview

To get our ENSEEIHT engineering degree, we are required to take part in a project called "Projet long" in teams of five students to work on a common project. The project started on January 19, and lasted eight weeks. It ends up with a defense in which we have to promote our work in front of a jury which evaluates us against different aspects :

- Project management and organization
- Technical accomplishment
- Report and defense presentation
- English evaluation

All over the project, we have had to work side by side with the clients for whom we have to deliver, at the end of the project, a product that suits their expectations.

We chose to work on that project because of the originality of the subject, since it is mixing theoretic computer science and technical advanced principles. Moreover, studying model checking and temporal logic to assert properties on a source code was a topic that some of us studied in ENSEEIHT courses. This project was an opportunity to apply this theory and dive deeper into it.

I.2 Subject

I.2.1 Main idea

The client was waiting for a prototype that allows a search of patterns on a C++ code. The patterns have to be expressed in terms of temporal logic properties.

I.2.2 Project description

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 for potential human failure. The goal of our project is to find out whether the embedded code meets a number of programming rules by defining allowed and deprecated of forbidden patterns. Our clients already have an existing tool named Coccinelle that is developed at INRIA. This tool detects patterns and also offers the possibility to modify the code. However, this tool only works on C code. The objective of the project was to design a prototype for pattern matching in a C++ code.

I.3 Objectives

From a pedagogical point of view, this project was the opportunity to apply a lot of techniques that we saw in our three years of courses in ENSEEIHT and new techniques that we learned while working on it. We took our project management courses as a reference to organize our time and to catch up with the deadlines.

To deliver a good product at the end, we realized that the good coordination in the team, the regular exchange of ideas during meetings and code/design reviews and production of relevant documents are main keys of success.

From a technical point of view, the clients expect us to design and create a tool written in Scala that would detect patterns in a C++ code using temporal logic expressions. They specified that our project would be a combination of two main parts (all of this will be explained in further detail in the « Technical aspects » section) :

- **C++ parsing and transformation :**
  In this part :
  - we take a C++ source code file as an input
  - we parse the AST (Abstract Syntax Tree) file generated by the Clang compiler to get an in-memory AST representation from
  - we convert the AST in a higher level tree data-structure adding some semantic to the structure (grouping *then* and *else* blocks of an *if* statement...)
  - finally, we transform the resulting data-structure into a graph data-structure called CFG (Control-Flow Graph) that is traversable by a model checking algorithm
- **Model checking :** the model checker takes a user-inputted temporal logic expression expressed in CTL (Computation Tree Logic) and then marks all the nodes in the graph verifying this logical expression. We will have to implement an extension of CTL, called CTL-V (CTL with variables), which allows us to quantify the meta-variables as well.

I.4 Constraints

Part II

Project management

II.1 Team organization

The way a project team is structured can play a major role in how it functions. Team structure will probably be adjusted at each stage to meet the evolving nature of the project. Building a good, effective team is vital. Team structure will influence the way the team behaves. It aims to create a collaborative team, where individuals share knowledge, cooperate, support each other and are motivated to achieve the team's goals. Here are the roles we decided to assign to the different members of the team along the project :

- **Project manager :** The project manager is primarily concerned about communications with the industrial and the customers. It has a leading role in the organization and planning of the tasks. The project manager is the person responsible for accomplishing the stated project objectives including creating clear and attainable project objectives, building the project requirements, and managing cost, time, scope, and quality. It is often a client representative and has to determine and implement the exact needs of the customer.
- **Supervisor :** The supervisor has a global technical view of the project. It supervises the advancement of simultaneous tasks. Otherwise, it can rearrange groups and objectives if an unforeseen occurs. The supervisor also has to participate in individual coding or documenting an assigned task. Nevertheless, it is not his primary function. The team supervisor can change from one week to another.
- **Quality manager :** The quality manager is in charge of checking that every deliverable documents meets the quality standards. In other words, any produced code will pass under the watchful eye of the quality manager before being validated. It also ensures the quality and consistency of all documents produced by the team.
- **Test manager:** The test manager is responsible of the validation and testing in global environment written by the developers (each developer has its own set of unit tests). It does not only run tests, it also determines whether the tests are exhaustive or not (code coverage, edge-cases...).
- **Configuration manager :** The configuration manager looks for useful tools that could bring help the team to meet the requirements (for example : Scalastyle, an Eclipse plugin that we use for automated quality checks). In particular, the configuration manager is good at using Git, the version manager we used in our project. This is necessary to have such skill in a group as some invalid repository states are sometimes confusing for an average git user.

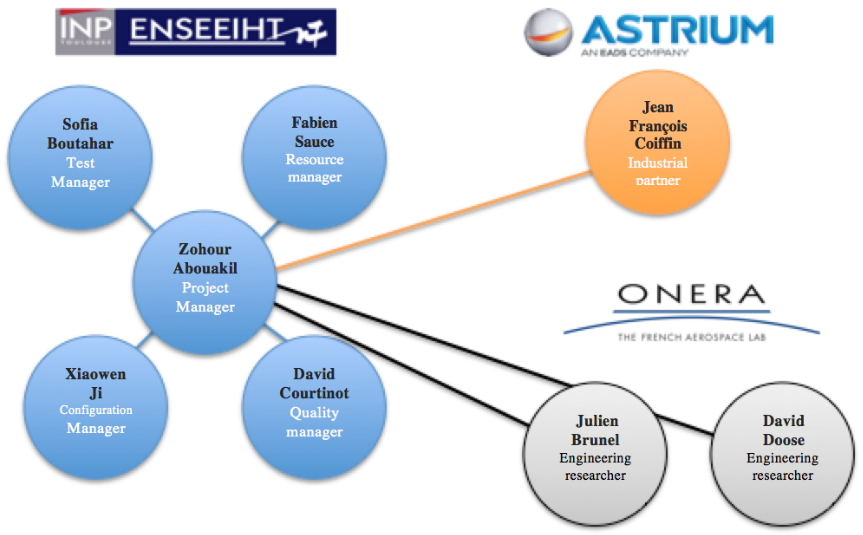

INP ENSEEIHT
ASTRIUM
AN EADS COMPANY
Jean François Coiffin Industrial partner
Sofia Boutahar Test Manager
Fabien Sauce Resource manager
Zohour Abouakil Project Manager
ONERA
THE FRENCH AEROSPACE LAB
Xiaowen Ji Configuration Manager
David Courtinot Quality manager
Julien Brunel Engineering researcher
David Doose Engineering researcher

Figure II.1 - The team organization of our project

Although everybody was involved in the same manner in searching, coding and testing stages, we gave everyone a role in order to improve the team organization. The project manager (Zohour ABOUAKIL) had several responsabilities related to the project management. She has been in charge of the communication with the industrial coordinator, Mr. Jean Francois COIFFIN and the scheduling of the different appointments. David COURTINOT was our quality manager and most of the time technical supervisor thanks to his knowledge of Scala prior to the project. Fabien SAUCE was in charge of documenting the source code and reacting most of the meeting minutes. Sofia BOUTAHAR was responsible for the unitary tests and the regressions tests when there is a new version of the code or a part that is fully developped. Xiaowen was in charge of the version tool Git and took care of the configuration of the few tools we used in the Project.

In addition to being the project manager, Zohour established a contact with the client to see if the team is going in the right direction or needed additional information.

II.2 Project objectives in terms of management and organization

- Managing coordination of the partners and working groups involved in the project.
- Writing a detailed project planning including:
  - developing and maintaining a detailed development plan.
  - managing project deliverables in line with the development plan.
  - recording and managing project issues and escalating where necessary.
  - resolving issues and try to prevent them.
- Managing project scope and change control and escalating issues where necessary.
- Monitoring project progress and performance.
- Managing project evaluation and dissemination activities.
- Final approval of the design specification.

- Working closely with clients to ensure the project meets business needs.
- Definition and management of the testing program.

II.3 Deliverable documents

II.3.1 Delivrable documents expected by ENSEEIHT and the industrial supervisor

- Report in PDF format
- Development plan in PDF format
- Presentation supports

II.3.2 Delivrable documents expected by the client

- Documented source code in Scala language
- Test strategy
- Architecture design document

II.4 Development plan

II.4.1 Development organization

We used the Scrum method, which is widely used, and recognized for its efficiency. At first, we defined a product backlog containing all desired functionalities in the final product. Next, we divided the project into three « sprints »(or « iterations »). A sprint backlog is defined for each sprint, including all we need to realize at the end of an iteration. Each sprint lasts two weeks and consists in improving the software incrementally, to gradually become closer to the product backlog. At the end of each sprint, we organised a meeting, in order to review the progress and propose improvements or modifications of planning, but in the process of a sprint, the sprint backlog cannot be modified. To finish, each day started with a scrum meeting where each team member presented his objective of the day and his current difficulties, if any.

II.4.2 Team organization approach for coding

We used an approach inspired by the XP (EXTREME PROGRAMMING) method which is a practice of pair programming. Considering the amount of code that we had to write, we found it unnecessary that the five team members work separately, and we considered as excellent to work in pairs, in order to prevent errors and bias of the program structure, so that we can save precious time in testing and debugging. As a consequence, four of us worked in pairs and the last one works individually or supervises us. The groups repartition could change as the tasks were completed.

II.4.3 Tasks organization

Task definition

The sprint backlog is a list of tasks that are identified by the members of the project and that has to be completed during the Scrum sprint. During our meetings, we tried to estimate how many hours and development efforts were needed to complete a task.

Sprint 1 backlog :

- AST parsing of procedure C++ code
- CFG conversion from parsed AST
- Model checking with simple properties

Sprint 2 backlog :

- AST parsing of object oriented C++ code
- CFG conversion from parsed AST
- Model checking with simple criteria

Sprint 3 backlog :

- Improved CFG conversion from parsed AST
- Model checking with complex criteria

Task planning

Our sprint backlog was maintained as a spreadsheet. During the Scrum sprint, each team member was requested to keep the sprint backlog updated.

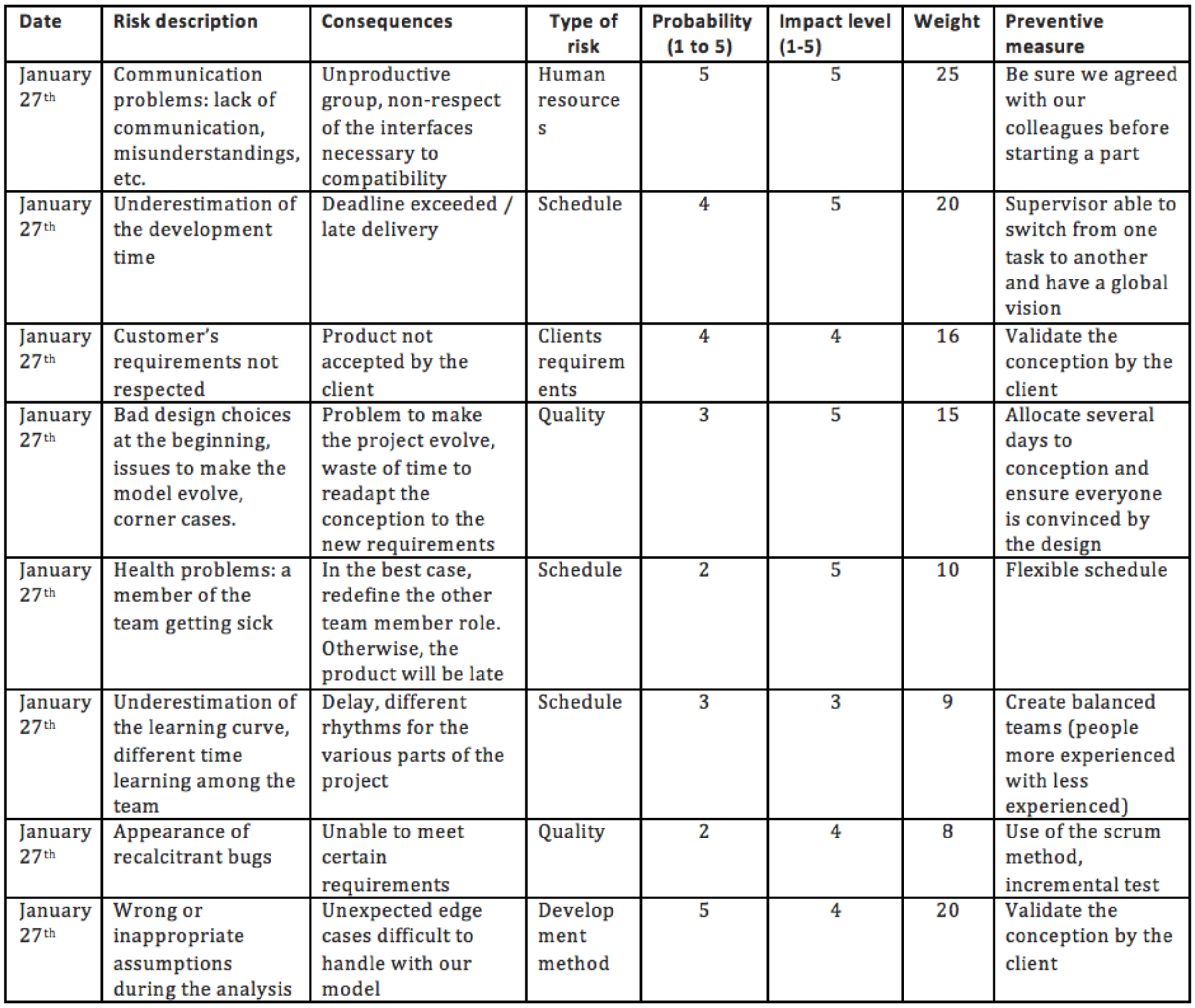
II.5 Risk management

II.5.1 Risk management strategy

The first step in project risk management is to identify the risks that may arise during the project. Some risks have a higher impact than others, and are more or less likely to occur. Basing on those two criterias, we classified the risks in order to visualize which points we had to focus on in priority to avoid problems in the future.

II.5.2 Risk analysis

Below is a table that summarizes the major risks that we could face in our project and how we planned to prevent them.



Date	Risk description	Consequences	Type of risk	Probability (1 to 5)	Impact level (1-5)	Weight	Preventive measure
January 27th	Communication problems: lack of communication, misunderstandings, etc.	Unproductive group, non-respect of the interfaces necessary to compatibility	Human resources	5	5	25	Be sure we agreed with our colleagues before starting a part
January 27th	Underestimation of the development time	Deadline exceeded / late delivery	Schedule	4	5	20	Supervisor able to switch from one task to another and have a global vision
January 27th	Customer's requirements not respected	Product not accepted by the client	Clients requirements	4	4	16	Validate the conception by the client
January 27th	Bad design choices at the beginning, issues to make the model evolve, corner cases.	Problem to make the project evolve, waste of time to readapt the conception to the new requirements	Quality	3	5	15	Allocate several days to conception and ensure everyone is convinced by the design
January 27th	Health problems: a member of the team getting sick	In the best case, redefine the other team member role. Otherwise, the product will be late	Schedule	2	5	10	Flexible schedule
January 27th	Underestimation of the learning curve, different time learning among the team	Delay, different rhythms for the various parts of the project	Schedule	3	3	9	Create balanced teams (people more experienced with less experienced)
January 27th	Appearance of recalcitrant bugs	Unable to meet certain requirements	Quality	2	4	8	Use of the scrum method, incremental test
January 27th	Wrong or inappropriate assumptions during the analysis	Unexpected edge cases difficult to handle with our model	Development method	5	4	20	Validate the conception by the client

Figure II.2 - The risk analysis for our project

II.6 Resource management system

II.6.1 Versioning tool

We used Git to manage our project, more particularly the versioning of the code and the documents that we redacted. We chose Git because it is a free and open-source distributed version control system that handles any software project in a highly efficient way. It also supports rapid branching and merging, and includes specific tools for visualizing and navigating through the development history. All the delivrable documents were managed on a git repository, including documentation and reports. Anyone was allowed to commit at anytime, however any push had to be authorized by the quality responsible after the code has been thoroughly tested against a set of tests by the test responsible.

II.6.2 Communication between team members

We used Google Drive to share all the documents between the team members. In particular, we stored on the drive the following contents : all the research papers and the interesting documents that we found or were given by the clients, pointers to external documentation, all documents related to the project management (planning, current sprint backlog...), minutes of the meetings either with the clients or the industrial coordinator.

II.7 Code and documentation

II.7.1 Quality checking

For ensuring that our coding rules were respected and evaluating the quality of our sources, we used a tool called Scalastyle that enables, using an easy-to-use xml configuration file, to examine the scala code and indicates potential problems with it. Combined with a specific pulgin, this can be used to generate warnings or errors in the IDE the developer is using. Our settings were the following :

Rule	Description	Value
FileLengthChecker	Check the number of lines in a file	1500
FileLineLengthChecker	Check the number of characters in a line	140
FileTabChecker	Check that there are no tabs in a file	enabled
ClassNamesChecker	Check that class names match a regular expression	^[A-Z][A-a-z]*$
ClassTypeParameterChecker	Checks that type parameter to a class matches a regular expression	^[A-Z_]$
FileTabChecker	Check that there are no tabs in a file	enabled
CyclomaticComplexityChecker	Checks that the cyclomatic complexity of a method does exceed a value	12
EmptyClassChecker	If a class/trait has no members, the braces are unnecessary	enabled
EqualsHashCodeChecker	Check that if a class implements either equals or hashCode, it should implement the other	enabled
MethodLengthChecker	Checks that methods do not exceed a maximum length	50
MethodNamesChecker	Check that method names match a regular expression	^[a-z][A-Za-z0-9]*(_=)?$
MultipleStringLiteralsChecker	Checks that a string literal does not appear multiple times	allowed = 2
NotImplementedErrorUsage	Checks that the code does not have ??? operators	enabled
NullChecker	Check that null is not used	enabled
NumberOfMethodsInTypeChecker	Check that a class/trait/object does not have too many methods	maxMethods = 30
NumberOfTypesChecker	Checks that there are not too many types declared in a file	maxTypes = 20
ObjectNamesChecker	Check that object names match a regular expression	^[A-Z][A-Za-z]*$
ParameterNumberChecker	Maximum number of parameters for a method	maxParameters = 5
RedundantIfChecker	Checks that if expressions are not redundant, ie easily replaced by a variant of the condition	enabled
ScalaDocChecker	Checks that the ScalaDoc on documentable	enabled

	members is well-formed	

This project meets a need from our clients. Therefore our codes will probably be used, studied and modified. That is why we have set up this quality approach. All our codes should be understandable and well commented. To achieve that, our programs were constantly reviewed by our clients as we gave them the link to our Git repository.

II.7.2 Verification and validation process

Verification

Verification is the process in which we determine whether the right solution is being developed. Once a functionality of our product is completed, we analyze the results in order to verify that the requirements have been met. In addition, the verification is an ongoing process: at the each completion of each milestone, we performed a verification analysis to make sure we were still on track.

As we have chosen the EXTREME PROGRAMMING model for the programming aspect of the project, we considered that a code passes the quality test if at least the two members of a pair have checked it. This is up to the quality manager to ensure this has been done, otherwise he should do it himself. This was specific to the code quality checks and did not apply to the rest of the delivrable documents.

Validation

Validation is the process in which we make sure that the solution that we came up with is constructed correctly and according to the requirements and the specifications. In this process, we performed various testing procedures on the code and tried to remove the defects as soon as possible. Once the found defects have being fixed, the process repeated itself.

Part III

Technical aspects

III.1 Context

III.1.1 Motivations

As a reminder, the aim of the project was to assert some good properties on embedded systems source code written in C++. More precisely, it consisted in static analysis of C++ programs. Thus, some properties cannot be checked by our software as they require runtime analysis. Nevertheless, a well-written code is more likely to work well, making such a tool interesting for checking the quality of a C++ source code.

III.1.2 Objectives

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 were 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.

III.2 Definitions

III.2.1 The AST - Abstract Syntax Tree

III.2.2 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 »*.

III.3 Model checking

The biggest problem we ran into in this part was to achieved a high level of genericity while keeping a simple architecture, not overly complicated to use for a specific case. We also had to ensure that the architecture is free of any dependance of any kind with the CFG part.

We have identified three kind of entities involved in model checking, each of them corresponds to a generic type :

- a type M describing the meta-variables. M must extend the MetaVariable trait.
- a type N describing the values contained by the nodes of the graph (GraphNode[N]). N can be anything.

- a type V describing the values of the environment. V must extend the Value trait

One could wonder why we chose using (M <: MetaVariable,V <: Value) instead of just (M,V) or (MetaVariable,Value). Here are the advantages of the proposed solution compared to the two others :

- **(M <: MetaVariable,V <: Value) vs (M,V)** : more evolutive, we can imagine adding operations on MetaVariable and Value in the future. This is an advantage for developing new features on the CTL part.
- **(M <: MetaVariable,V <: Value) vs (MetaVariable,Value)** : more accurate type. This enables the developer using the model checker to specialize these generic classes in a more powerful way than it would be in the second case, because the access to specific methods of M and V would be lost. This is an advantage for using the CTL part in any kind of application.

III.3.1 Representation of the environments

Definition 1 A *positive binding* is an element of $M \times V$. It associates a meta-variable to a specific value.

Definition 2 A *negative binding* is an element of $M \times \mathscr{P}(V)$. It associates a meta-variable to a set of illegal values.

Definition 3 Two bindings are said *in conflict* if :

- they are two positive bindings (m, v_1) and (m, v_2) such as $v_1 \neq v_2$
- they are one positive binding (m, v) and one negative binding (m, V) such as $v \in V$

Definition 4 An *environment* is a set of positive and negative bindings. An environment containing conflicting bindings is noted $\perp$.

Methods and class hierarchy

At first, positive and negative bindings were stored in two separate maps but we finally decided to use an abstract class MetaVarBinding extended by two case-classes to represent all kinds of bindings and use a single map. The Environment class contains all the abstract operations between or on environments required by the algorithm :

- intersection (noted & in reference to the & method of the Set trait)
- removal of a binding (noted - as it is a standard notation for removal in a Scala collection)
- opposite (noted ! in reference to the logical negation in Scala)

The following diagram presents the chosen design for environments. Next parts of this section will focus on explaining it all :

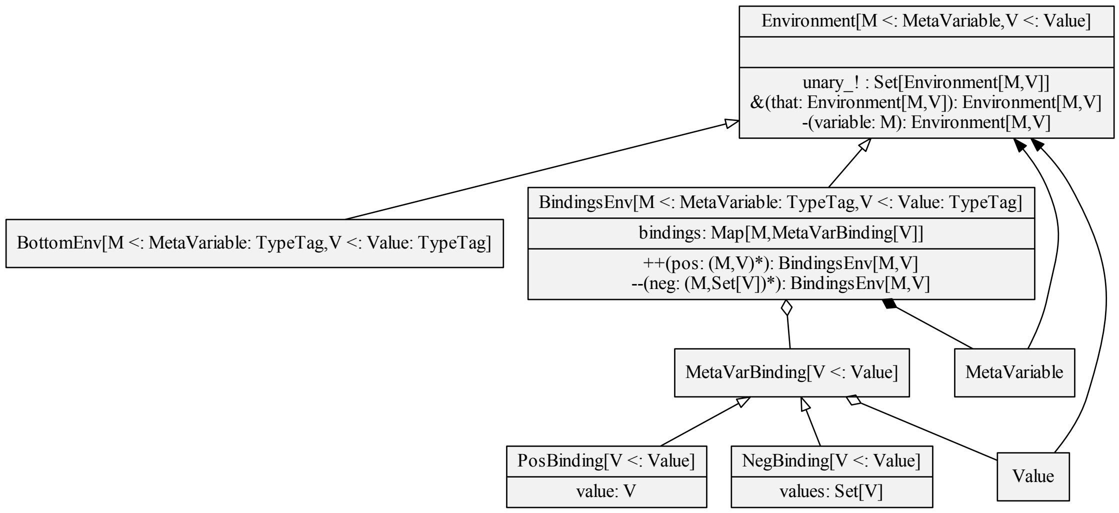

Environment[M <: MetaVariable,V <: Value]
unary_! : Set[Environment[M,V]]
&(that: Environment[M,V]): Environment[M,V]
-(variable: M): Environment[M,V]
BottomEnv[M <: MetaVariable: TypeTag,V <: Value: TypeTag]
BindingsEnv[M <: MetaVariable: TypeTag,V <: Value: TypeTag]
bindings: Map[M,MetaVarBinding[V]]
++(pos: (M,V)*): BindingsEnv[M,V]
--(neg: (M,Set[V])*): BindingsEnv[M,V]
MetaVarBinding[V <: Value]
MetaVariable
PosBinding[V <: Value]
value: V
NegBinding[V <: Value]
values: Set[V]
Value

Figure III.1 - Environement class hierarchy

The specific case of the ⊥ environment

The ⊥ environment was a bit tricky to represent. Indeed, environments depend on generic types M and V, which implies that Environment must be a generic class. Thus, BottomEnv also must to be generic. However, considering the mathematical definition of ⊥, it would be poor design to represent it with a regular class instead of a singleton.

We want all self-contradictory environment of the same generic type to be equal, and if possible we would like two self-contradictory environments of different generic types not being equal, to keep the type consistency. As Scala does not allow a singleton to be generic (which is quite logical by the way=, we have had to use reflexivity to implement the singleton pattern without using the **object** keyword. We also used some implicit definitions to allow nice syntax using the Bottom object to fetch the BottomEnv singleton of the appropriate type given the context.

Regular environments

Having defined the MetaVarBinding class and its children case-classes, it seems natural to use a single Map[M,MetaVarBinding[V]] map to represent the bindings. The only real design problem we have got here was the fact that the reflexivity used to solve the BottomEnv problem was infectious. We have had to use TypeTag(s) in some generic methods and in the BindingsEnv constructor. Thus, any external code calling the main constructor was forced using TypeTag(s) also. To address this issue, we made the main constructor private and declared and secondary constructor TypeTag-free creating an empty BindingsEnv object. The user then has to add bindings to this environment using the ++ and - - methods.

III.3.2 CTL expressions

Defining the CTL-V language

There is nothing too difficult here, the class hierarchy is directly derived from the mathematical definition of CTL operators. The only interesting points are the way of defining generic predicates as well as the way of quantifying variable with the ∃ quantifier. These points will be discussed in the next subsections. Another thing to mention is that we did not define AG, EG, AF and EF as classes but used the mathematical equivalences between the different operators to define them with implicit declarations using the other operators. This way, we don't have to handle them as particular cases in the model checking algorithm since they are first converted into a combination of the other operators.

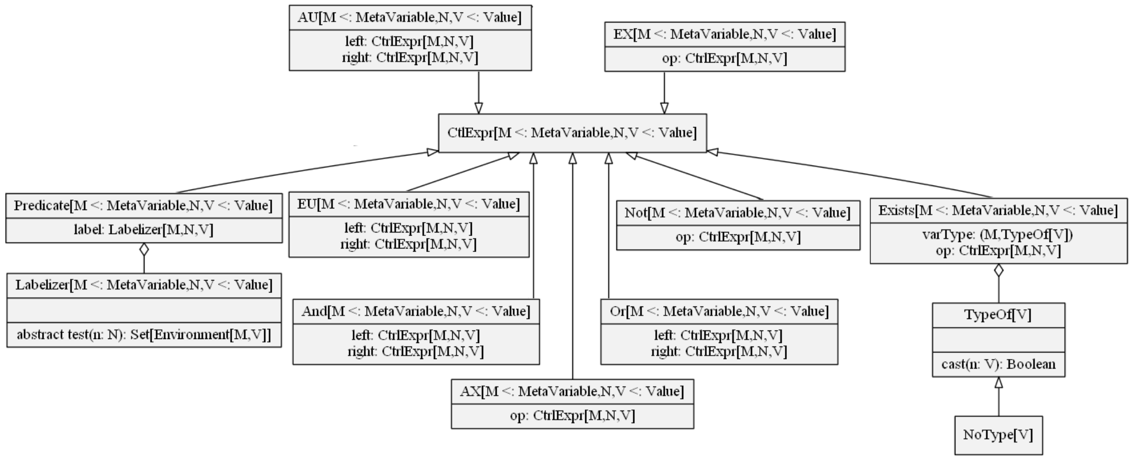

AU[M <: MetaVariable,N,V <: Value]
left: CtrlExpr[M,N,V]
right: CtrlExpr[M,N,V]
EX[M <: MetaVariable,N,V <: Value]
op: CtrlExpr[M,N,V]
CtlExpr[M <: MetaVariable,N,V <: Value]
Predicate[M <: MetaVariable,N,V <: Value]
label: Labelizer[M,N,V]
EU[M <: MetaVariable,N,V <: Value]
left: CtrlExpr[M,N,V]
right: CtrlExpr[M,N,V]
Not[M <: MetaVariable,N,V <: Value]
op: CtrlExpr[M,N,V]
Exists[M <: MetaVariable,N,V <: Value]
varType: (M,TypeOf[V])
op: CtrlExpr[M,N,V]
Labelizer[M <: MetaVariable,N,V <: Value]
abstract test(n: N): Set[Environment[M,V]]
And[M <: MetaVariable,N,V <: Value]
left: CtrlExpr[M,N,V]
right: CtrlExpr[M,N,V]
Or[M <: MetaVariable,N,V <: Value]
left: CtrlExpr[M,N,V]
right: CtrlExpr[M,N,V]
TypeOf[V]
cast(n: V): Boolean
AX[M <: MetaVariable,N,V <: Value]
op: CtrlExpr[M,N,V]
NoType[V]

Figure III.2 - CTL expression class hierarchy

Predicates

To enable to make generic predicates, we simply defined a generic class Predicate that wraps a Labelizer. A Labelizer defines a *test* method that takes an object of type N (the type of the GraphNode(s)) and returns an Option[Environment[M,V]]. If the predicate is not verified by the input node, this method must return None, otherwise it returns the set of positive bindings that make the node match the predicate. This set is possibly empty if the predicate does not involve any meta-variable.

∃ quantifier

The ∃ quantifier is a bit tricky to define because of the ex_binding operation defined in the model checking algorithm, which requires to know the set of all possible values for a given meta-variable. The problem is that the type V may be composed of inconsistent types wrapped by some classes extending V. For example, in the case of the model checking on the CFG, if X has been assigned a negative binding composed of declarations, then we should not consider that X could possibly be an Expr.

To sum up, we must be able to have a type information about the meta-variable at least when encountering the ∃ quantifier. Therefore we introduced the TypeOf class which defines a cast operation. This way, incompatible values will be filtered. This is completely generic as the cast operation is defined by the user depending on its needs.

III.3.3 Model checker

The only interesting thing to mention here is the use of a conversion method for compute the Val set of all possible values. In order to compute Val, all the nodes of the graph are traversed. For each node, we call the converter that returns all the values of type V a given node is likely to add to the environment. Note that a single node can introduce several values in the environment (for example a node f(5,6) would inject 5 and 6 when applying a f(x,y) labelizer), that's why the converter must return a Set[V]. We could have defined the conversion as an abstract method of the GraphNode, but we thought it was more convenient to do it externally. It is exactly the same difference as Comparable and Comparator in Java : Comparable defines the comparison operation directly on the class, it is useful when a natural order exists but it enables only one implementation of the comparison whereas we can define as many Comparator(s) as we want for the same type.

III.4 Merge

III.4.1 Choice of M, N and V

Considering the kind of predicates we wanted to use and the types of node we were using (as a recall, type N will be ProgramNode for the CFG), it was kind of obvious that we would need Expr values in our environment, so we introduced the type CFGExpr, which simply wraps an Expr object. However, it was not so easy determining what other kinds of values would be required. To solve this question, we thought about the labelizers and properties we wanted to be able to implement to deduce our needs. Two important properties led us to introduce types CFGDecl and CFGExpr :

- « *There should be no hidden declarations (declaration of an entity of the same kind, same type and same name in different scopes)* »
- « *There should be no declared entity which is never used* »

We can see there that *declaration* has a slightly different meaning in those two cases. In the first case, two declarations are considered equal (this is the way we have to interpret *hidden* in order to create conflicts in the environments in case of hidden declarations) if they declare an entity of the same kind, type and name. We prefer calling that a *definition* rather than a *declaration*. In the second case, all declarations have to be considered unique because we do not care if a variable of the same type and name corresponding to a declaration was used in the code, we want to know if every declaration, individually, was used. Thus, CFGDef will be used to represent a definition, and CFGDecl to represent a declaration. The only difference between those two is that CFGDecl takes an id as a parameter and bases its equality check on this id whereas CFGDef bases it on the pair (type,name).

Finally, we saw that it could be interesting to make some type checks, or for example, when looking for arithmetic on pointers, which requires to distinguish the = operator, which is allowed for pointers, from + or -, which are deprecated). Therefore, we also allowed strings to be values of the environment, so we created a wrapper type CFGString. The following diagram sums this all up :

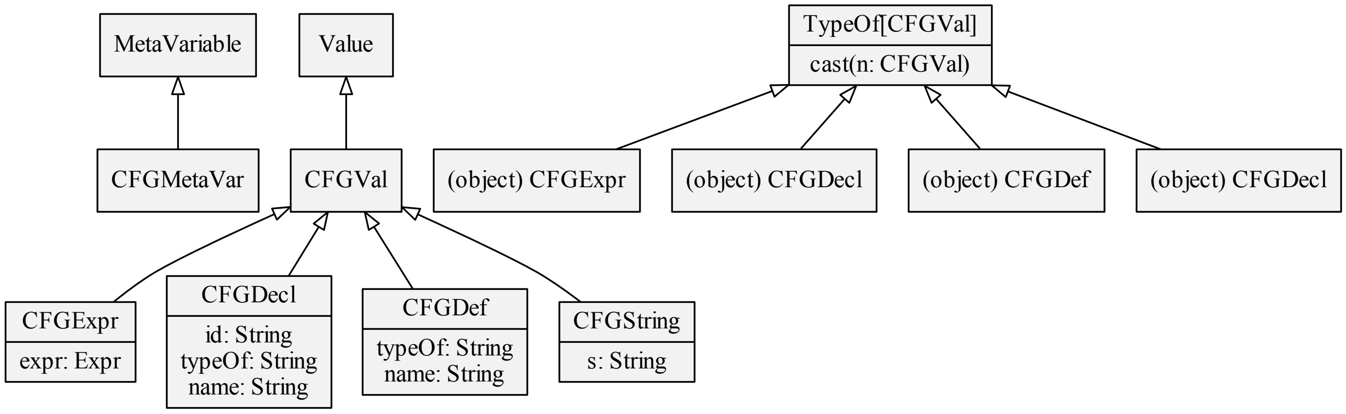


Figure III.3 - Concrete implementation of the generic types of the CTL part

III.4.2 Labelize our nodes

Patterns

Most of the predicates we need to use were based on the idea of a pattern of code. For example, $X = 0$ is a pattern describing all the assignments to 0 of a variable to be determined (using capital-size letter means that this is a meta-variable and not a variable of the code). Using patterns, we are able to tell many things about a node and express many base properties. The patterns had to be able to match all kinds of entities in a code and allow for returning bindings between the meta-variables of the pattern and the matching values in case of success of the matching. For the same reasons as in the previous part, we determined that we needed :

- patterns for expressions : ExprPattern
- patterns for declarations : DeclPattern
- patterns for string values : StringPattern

For expression patterns, we did not want to explore more than one level of depth of the expressions. Deeper exploration of the expression tree would be nearly pointless for the properties we aim to assert on the CFG, moreover some patterns may be ambiguous or more bothering to define if we explored the tree deeper (for example, $5 + x + y + 6$ can be matched in several ways by the $X + Y$ patterns, X and Y being meta-variables). Even if the AST generated by an expression is deterministic, which makes deep pattern matching possible, we thought it was not very clear for the user what he would get when using deep patterns. Therefore, we introduced the AtomicExprPattern, which is only extended by leaves expressions patterns and is the only kind of ExprPattern that can be passed as a parameter to a non-leave expression pattern. This choice is entirely reversible and has no consequence on the rest of our design.

StringPattern and ExprPattern have in common the fact that they can be whether an UndefinedVar, which is a simple wrapper of a meta-variable that matches anything and returns the appropriate binding, or a specific value (DefinedExpr, DefinedString). StringPattern can also be a NotString, which is comparable to a negative binding : it is a set of forbidden values. We did not need to use such thing for ExprPatttern, but nothing in our conception prevents us from adding it later.

Finally, DeclPattern is a bit different than the other two. It can also be an UndefinedVar, but we did not see any use-case for a DefinedDecl pattern. We actually only treated variable declarations, which required to have a powerful pattern-matching based on the name, the type and the assigned expression, if any. We needed two patterns doing mostly the same thing but returning in one case a CFGDecl, and in the other case a CFGDef. Consequently, we factorized the code of those two patterns in VarDeclMatcher. The specific behaviour of VarDefPattern and VarDeclPattern is implemented in the conversion method passed as a parameter in the constructor of VarDeclMatcher.

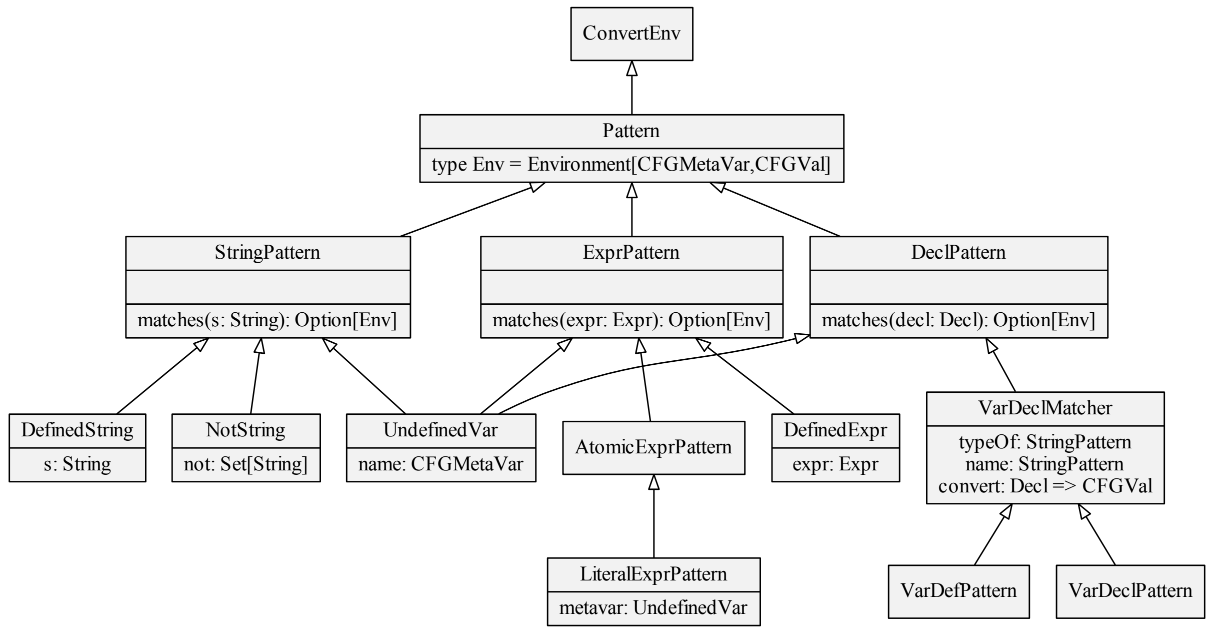


Figure III.4 - Hierarchy of the Pattern type

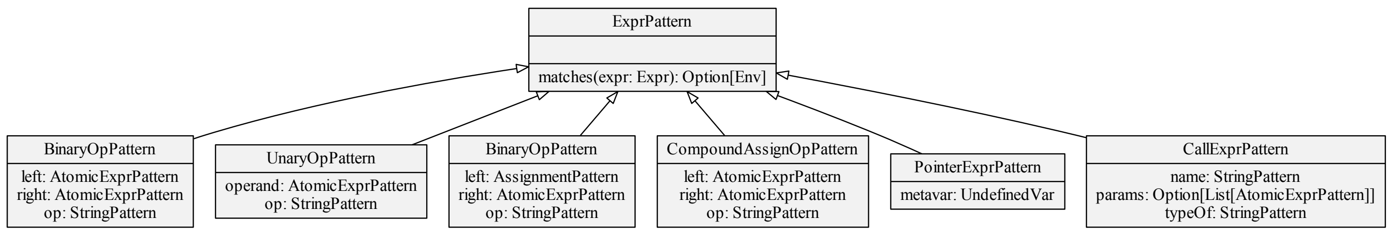


Figure III.5 - Detail of all the expression patterns

Labelizers

Basically, our labelizers simply wrap a Pattern to add a specific semantic. Here is the list of all the labelizers, with their semantic :

- **IfLabelizer**, **ForLabelizer**, **WhileLabelizer**, **SwitchLabelizer**, **ExpressionLabelizer** : all those labelizers firstly check than the node considered is of the appropriate type (respectively : If, For, While, Switch, Expression) and then if the expression it eventually holds matches a given pattern.

- **VarDeclLabelizer**, **VarDefLabelizer** : match any Statement node containing a VarDecl matching a given pattern. The difference betwen those two classs is the kind of CGVal returned in case of success (CFGDecl or CFGDef).

- **FindExprLabelizer** : matches any node containing at least an occurrence of a given pattern. Returns all the occurrences of the pattern in the node.

- **MatchesExprLabelizer** : matches any node holding an expression that exactly matches a given pattern. It returns a single value in case of success, which makes it different from FindExprLabelizer.

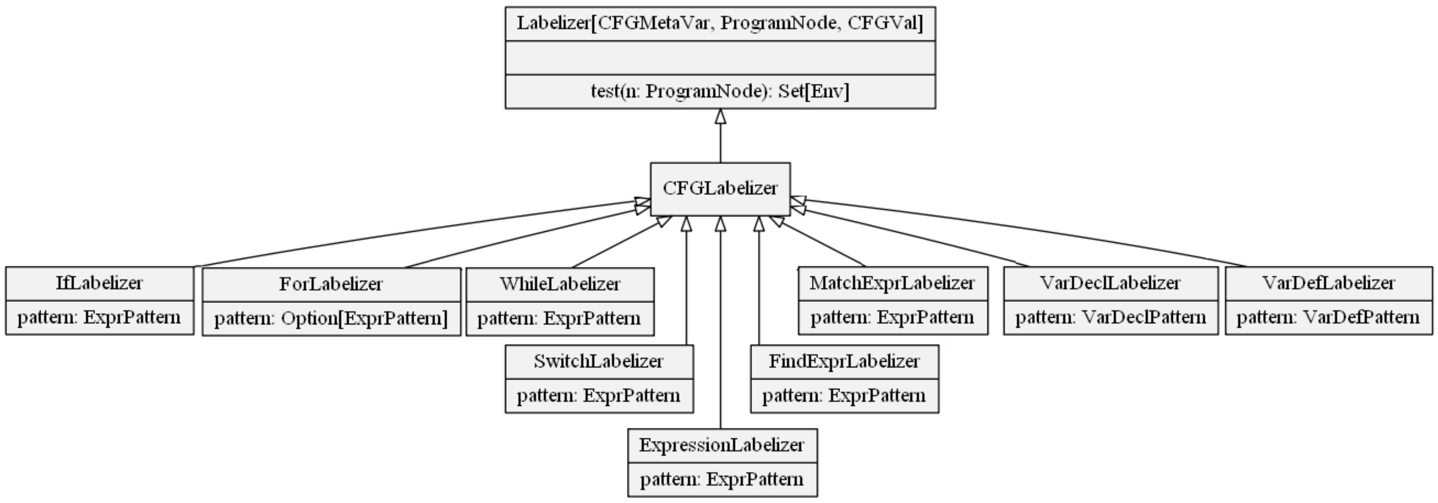


Figure III.6 - Detail of all the labelizers we implemented