Taster, a Frama-C plug-in to enforce Coding Standards

David Delmas[1], Stéphane Duprat[2], Victoria Moya Lamiel[2], and Julien Signoles[3]

[1] Airbus Operations S.A.S., 316 route de Bayonne, 31060 Toulouse Cedex 9, France, Firstname.Lastname@airbus.com
[2] Atos Origin, 6 Impasse Alice Guy, B.P. 43045, 31024 Toulouse Cedex 03, France, Firstname.Lastname@atosorigin.com
[3] CEA LIST, Software Safety Labs, PC 94, 91191 Gif-Sur-Yvette Cedex, France, Firstname.Lastname@cea.fr

Résumé Enforcing Coding Standards is part of the traditional concerns of industrial software developments. In this paper, we present a framework based on the open source Frama-C platform for easily developing syntactic, typing (and even some semantic) analyses of C source code, among which conformance to Coding Standards. We report on our successful attempt to develop a Frama-C plug-in named Taster, in order to replace a commercial, off-the-shelf, legacy tool in the verification process of several Airbus avionics software products. We therefore present the types of coding rules to be verified, the Frama-C platform and the Taster plug-in. We also discuss ongoing industrial deployment and qualification activities.

1 Introduction

In the avionics domain, it is a requirement of the applicable DO-178B/ED-12B international standard to both define the rules and constraints for the coding process, and ensure that these standards were followed during the development of the code. The scope of coding standards is described in section 11.8 of [1]. *These standards should include:*

a. *Programming language(s) to be used and/or defined subset(s). [...]*
b. *Source Code presentation standards [...].*
c. *Naming conventions for components, subprograms, variables, and constants.*
d. *Conditions and constraints imposed on permitted coding conventions, such as the degree of coupling between software components and the complexity of logical or numerical expressions and rationale for their use.*

Then, section 6.3.4.d of [1] prescribes *Reviews and Analyses of the Source Code* as a means to check *Conformance to standards, especially complexity restrictions and code constraints that would be consistent with the system safety objectives.*

Section 2 of this paper gives an overview of the Airbus context, regarding coding standards and verification of conformance. Next, section 3 presents the Frama-C platform, and the support it provides for external plug-in development. Then, section 4 describes the Taster plug-in itself, and how its implementation makes use of Frama-C facilities. Finally, section 5 discusses results, prospects and operational deployment.

2 Airbus practice

2.1 C Coding Standards

The coding rules applicable on C avionics programs developed at Airbus may be classified into three main categories: syntactic rules, typing rules and semantic rules.

Syntactic rules address code readability and maintainability.

- Formating conventions prescribe the layout and structure of source and header files, and the naming of identifiers.
- Langage restrictions limit the set of possible C constructs, by forbidding some operators and keywords. For instance, the "," operator is prohibited, and the use of the *static* keyword is restricted - for testability reasons.
- programming guidelines limit the size and complexity of statements and expressions. For instance, the number of nested control structures in a C function may not exceed a given upper bound.

Typing rules address program portability and safety. Such rules mainly define:

- the list of authorized conversions between expressions with different types or type qualifiers. Indeed, most implicit (and some explicit) conversions between incompatible types are prohibited. In particular, conversions between incompatible pointer types are strictly forbidden. Conversions discarding a *const* qualifier from a pointer target type are not permitted either.
- the types or signedness of operands of specific C operators. For instance, arithmetic operators may not be used on operands of enumerated types, bitwise shift operators may not be applied to signed quantities, and equality testing between floating-point numbers is tolerated only in special conditions.

Semantic rules address program safety and security.

- Dataflow-related rules forbid the use of local variables prior to initialisation, or the assignment of the loop variable in the body of a *for loop* statement, or a local variable to escape its scope through a

pointer, or expressions with undefined evaluation order, and require that all output parameters of functions (defined at design level and annotated in prototypes) should be assigned whatever the calling context.
– Run-time error avoidance-oriented rules provide programming guidelines to avoid array access out of bounds and invalid pointer dereference.

2.2 Verification of conformance

When we decided to start developing Taster, the coding standards verification process at Airbus could be described as follows.

Conformance to most syntactic rules was checked by an in-house scripting-based toolset. This tool is written in the Python programming language, and uses `gcc`[4] and `GNU cflow`[5] as utilities.

Conformance to typing rules was checked by a COTS[6] tool. However, we were faced with an obsolescence issue. Indeed, this COTS tool only runs on an obsolete platform, and no evolutive maintenance is provided. Such a situation cannot be durable in the avionics domain, where both products and development workshops have to be maintained over decades, in a continuous engineering approach allowing for continuous process improvement.

Conformance to all remaining rules was verified during manual code reviews. Note that the COTS tool also implements bug-finding heuristics for some of the dataflow and run-time error-related rules. These heuristics are industrially useful in a code debugging phase; however, they cannot be used for verification, as they are unsound (cannot find all bugs). For all these remaining rules, tool support is needed to reduce the cost of the verification of conformance.

3 The Frama-C platform

3.1 Overview

Frama-C [3] is a framework that allows static analysers, implemented as plug-ins developed in Objective Caml [9] (OCaml for short), to collaborate towards the study of a C program [7]. Although it is distributed as Open Source, Frama-C is very much an industrial project, both in the size it has already reached[7] and in its intended use for the certification, quality assurance, and reverse-engineering of industrial code.

[4] http://gcc.gnu.org/
[5] http://www.gnu.org/software/cflow/
[6] commercial, off-the-shelf
[7] Between 100 to 300 thousands line of OCaml code, depending on which plug-ins you are counting.

It is being actively co-developed by two French public institutions:

– CEA LIST (Software Reliability Laboratory) ;
– INRIA Saclay-Île de France (ProVal team, common with LRI at Université Paris-Sud).

The developement started within collaborative research projects[8] in which Airbus participated. Although coding standards enforcement is only a very small subset of what may be performed with Frama-C, this is the first large-scale industrial use implemented at Airbus.

3.2 Architecture

Frama-C is architectured in three different parts (see Figure 1):

– **Cil** (C Intermediate Language) [11] extended with an implementation of a specification language called ACSL (ANSI/ISO C Specification Language) [2]. This extended Cil is an intermediate language upon which Frama-C is based (see Section 3.3).
– **Kernel**: it provides data structures and operations, helping developers to deal with the Cil Abstract Syntax Tree (AST), as well as general-purpose services providing a uniform set of features to Frama-C (see Section 3.4).
– **Plug-ins**: they perform several analyses or source-to-source transformations by extending the Frama-C kernel through several extension points. These plug-ins can be used by other ones through their API registered (statically or dynamically) in the Frama-C kernel (see Section 3.5).

3.3 Extended Cil

Frama-C intends to handle any ANSI/ISO C programs. For this purpose, it is fully based on Cil [11] which is an OCaml library that provides a parser (which uses an external customizable C preprocessor) and an AST-level linker for C code. Cil transforms the initial C files into a single highly-structured AST featuring a reduced number of syntactic and conceptual C constructs. That eases the implementation of C analysers without restricting the target programs to a strict subset of C.

Cil also provides a toolbox that permits easy analysis and source-to-source transformation of C programs. For instance, it provides some simple but useful miscellaneous datastructures and operations over the AST as well as some syntactic analyses like a (syntactic) call graph computation or generic forward/backward datafow analysis, which can be used by Frama-C developers.

[8] This work has been partly supported by the French ANR (Agence Nationale de la Recherche) during the CAT (C Analysis Toolbox) and U3CAT (Unification of Critical C Code Analysis Techniques) projects.

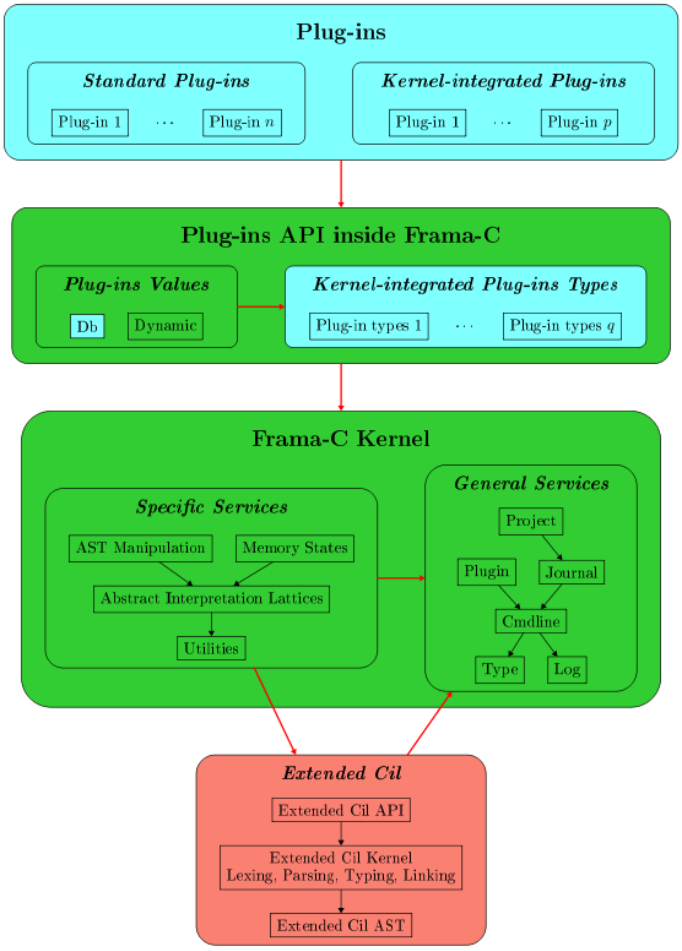

Fig. 1. Architecture Design

Frama-C extends Cil with an implementation of the specification language ACSL [2] in order to deal with formally-specified C programs. Cil is also modified in order to properly interact with other general services prodived by Frama-C like the project system (several analyses performed on several AST in the same session) [12] or the uniform way to display messages to end-users.

3.4 Frama-C Kernel

Frama-C provides several services to handle C programs. Here we only focus on the two most important ones for the Taster plug-in.

Typed and Untyped AST Two different trees are available : the untyped AST and the typed AST.

The untyped AST is very close to the analysed C code as it is the direct result of the parsing of the source code. Nevertheless no information about C types is available in this tree.

On the other side, the typed AST contains more information about the constructs of the C code, above all the C types. Such an AST is simpler to manage for semantic analysers. But it is less close to C code, so it is more difficult to understand for end-users and to manage for purely syntactic analysers.

Visitors Frama-C provides inheritable visitor classes in order to permit easy visiting of the (typed) AST. It allows to traverse the AST by redefining the default behaviour of each of its elements.

Although an original visitor was defined by the Cil library, Frama-C extends it in order to perform copying or in-place modification of an AST. The Frama-C visitor also inherits from the Cil one in order to consistently update the internal state of Frama-C itself during the visit, especially to correctly deal with the project system.

There is also a visitor dedicated to the untyped AST.

3.5 Plug-in Development

At the plug-in level, Frama-C is a big library which promotes code reuse and plug-ins collaborations through a plug-in API database in which each plug-in may invoke a function of another one.

For this purpose, the platform offers several registration points to connect a plug-in, implemented in its own directory, to the platform (see Figure 2 for the most important ones). By this way, the Frama-C kernel and other plug-ins may execute selected parts of a plug-in when required. Frama-C provides detailled documentation on the way to develop a Frama-C plug-in fully integrated within the platform [13].

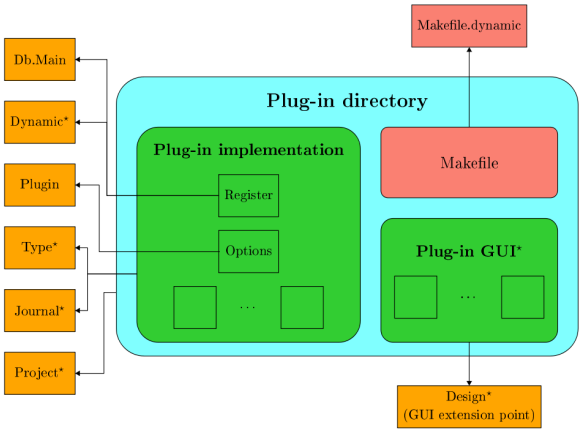

Fig. 2. Plug-in Integration

3.6 Value Analysis Plug-in

The Frama-C value analysis plug-in [6] performs a correct over-approximation of the set of possible values taken by a variable at a given program point. For this

purpose, it uses abstract interpretation techniques [4], [5].

For a left value or an expression from a specific statement, the result of the evaluation contains all the values that this element may have during the execution any time the point just before (or just after) the selected statement is reached.

The result of this evaluation can be requested directly by the user (through the GUI or in batch mode) or accessed by a custom plug-in through the value analysis API.

Furthermore some additional functions of the value analysis API provide to custom plugins information about the representation of the possible values for a given code element, so that a custom plug-in may reason entirely in terms of abstract locations, and completely avoid dealing with the problems of pointers and aliasing.

There are several types of warning displayed by the value analysis:

- Division by zero;
- Unspecified logical shift;
- Floating-point alarms;
- Uninitialized variables and dangling pointers;
- Invalid memory accesses;
- Unspecified pointer comparison alarms;
- Conflicting side-effects.

In addition to those warnings, some messages may warn that the analysis is making an operation likely to cause a loss of precision, which are only informational messages.

Several options are available, among them "-slevel" which makes the analyzer unroll loops or propagate separately the states that come from the "then" and "else" branches of a conditional statement. This option makes the analysis more precise (at the cost of being slower) for almost every program that can be analyzed.

4 The Taster plug-in

TASTER (Typed Abstract Syntax Tree ExamineR) is a Frama-C plug-in which allows to verify C source code complies with a set of given C coding rules. It performs syntactic and semantic checks on the abstract syntax tree (AST) and the control flow graph (CFG) provided by the Cil [11] layer of Frama-C.

For each coding rule, one or several implementations of the associated check have been fulfilled, depending on the required verification level.

These checks may be classified into three main categories: syntactic and typing analyses, bug-finding heuristics and semantic analyses.

4.1 Syntactic Typing analyses

The Taster plug-in defines OCaml classes inheriting from Frama-C visitors and overloading methods in order to scan the typed and untyped abstract syntax trees and to verify coding rules presented in 2.1.

This overload allows the Taster plug-in to traverse the corresponding AST and perform the required checks.

Most Taster checks are implemented on the typed AST. However, some syntactic rules refer to the precise syntax of the original code. Such rules need to be checked on the untyped AST, using the `Cabs` module provided by Cil.

Like Cil and Frama-C, Taster is entirely implemented in the OCaml programming language, it hence benefits from OCaml powerful programming features (see Section 5.3).

4.2 Bug-finding heuristics

Like typing analyses, these features use Frama-C and Cil visitors to navigate through ASTs.

However, such features also need to build upon dataflow information, especially reachability information between statements. Therefore, Taster's bug-finding heuristics gather indirect information from the Cil generated CFG of the analysed C code. As a matter of fact, most such analyses boil down to forward and backward searches through the CFG.

Their actual implementation in Taster consists in a set of recursive OCaml methods, which make extensive use of dedicated operations from Cil APIs, especially Cil operations returning previous or next statements from a node in the CFG, and operations computing direct reachability information between distant nodes.

Likewise, Taster's heuristics are able to hunt for uninitialised variables or array accesses out of bounds while walking *Eulerian paths* [8,10] in the CFG. Note that part of these heuristics depend on normalised comments in the source code of C function prototypes.

Such normalised source comments are included neither in the untyped, nor in the typed AST. Therefore, Taster extracts comments from raw lexing data provided by the `Cabs` layer of Cil, which gathers all comments from all source files into a single global array.

4.3 Semantic analyses

Taster implements experimental features of semantic analyses based on the use of the `value` analysis plug-in introduced Section 3.6.

First, the Taster plug-in asks the value plug-in to perform value analyses from all library functions of the source code to be analysed. Then, Taster retrieves information in two ways:

- The first approach is to gather alarms and warnings emitted by the value analysis that match coding rule violations;
- The second approach is to read the values computed for all program variables, directly in the format provided by the value analysis plug-in.

Both ways allow in principle for the proof of absence of run-time errors, provided stubs or annotations input enough correct context information. As the value analysis implements (non relational) static analysis by abstract interpretation false alarms may be raised.

So far, these functionalities are only used experimentally within Taster for (incomplete) detection of uninitialised variables, out of bounds array accesses, and local variables escaping their scope through pointer dereferences. The detection is said to be incomplete, as the current version of the Taster plug-in does not retrieve all relevant context information.

4.4 Use of Frama-C APIs by Taster

The Cil API of Frama-C is not the only one used by Taster. For instance, the `Globals` and `Kernel_function` modules provide both useful methods and data structures to handle global variables and function prototypes.

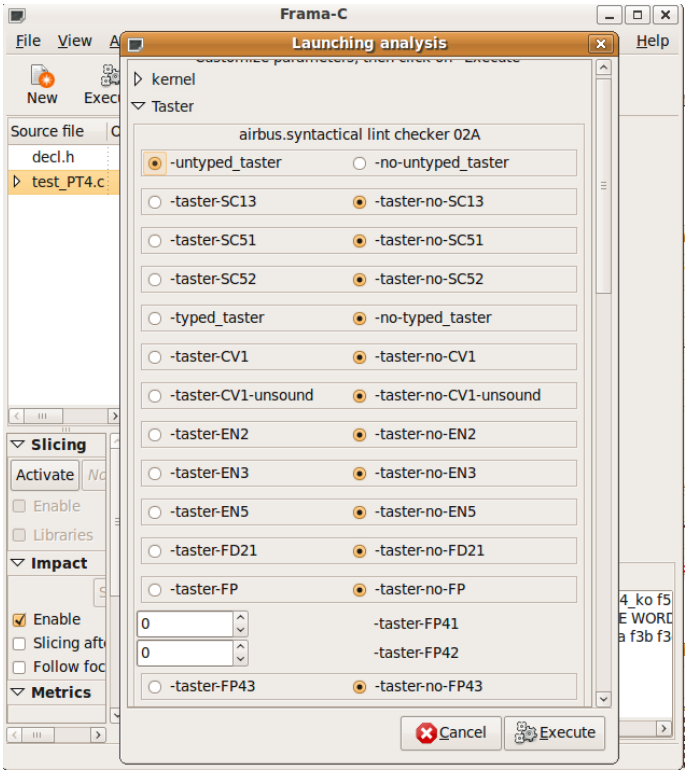

Fig. 3. Taster's options on the Frama-C GUI

Among modules used by Taster, one is to quote:

- the `Cil_utils` library provides useful functions to browse Cil representations of C data structures;
- the `Cabs` layer of Cil furnishes with structures and operations to traverse the AST as well as to access comments from the source;
- the `Db` and `Dynamic` databases contains all registered plug-ins – such as Taster, so that they can be used by other plug-ins.

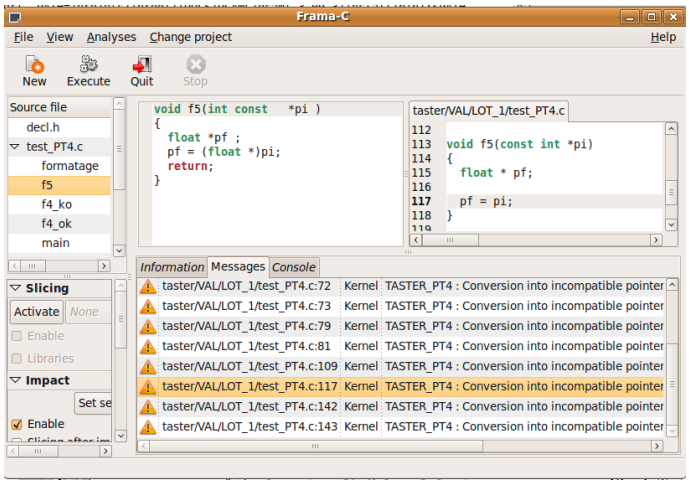

Fig. 4. Taster's warnings in the Frama-C GUI

- the `Plugin` module allows to register each plug-in. The plug-in registration allows batch mode users to view specific plug-in help menus from the general `-help` option of Frama-C, and GUI users to set specific plug-in options (see Fig. 3) and visualize Taster plug-in results directly on the Frama-C GUI (see Fig. 4);
- the `Alarms` database defines types to represent alarms, and functions to manipulate them;
- the experimental value analysis-based features make use of the `Location`, `Db.Value`, `Relations_type` and `Cvalue_type` modules.

5 Results and prospects

Fig. 5 gives information on the overall architecture of the Taster plug-in, together with some figures on the size of its components, to be compared with the size of the Frama-C layers above which they are built[9].

The current version implements a set of options in the **tasterParameters** component, allowing for 27 distinct analyses:

- **ast_lint** and **bib_ast_lint** implement 15 syntactic and typing checks on the typed ASTs and 5 bug-finding heuristics on the CFGs provided by the Frama-C and Cil layers;
- **untyped_ast_lint** implements 3 syntactic checks on the untyped ASTs;
- **value_ast_lint** implements 4 experimental semantic analyses, processing outputs from the Frama-C value analysis plug-in.

5.1 Industrial deployment

During the development of Taster, we have successfully:

[9] In our context, we reckon we would have to write about ten times more lines of code if we were to re-implement the same features in C.

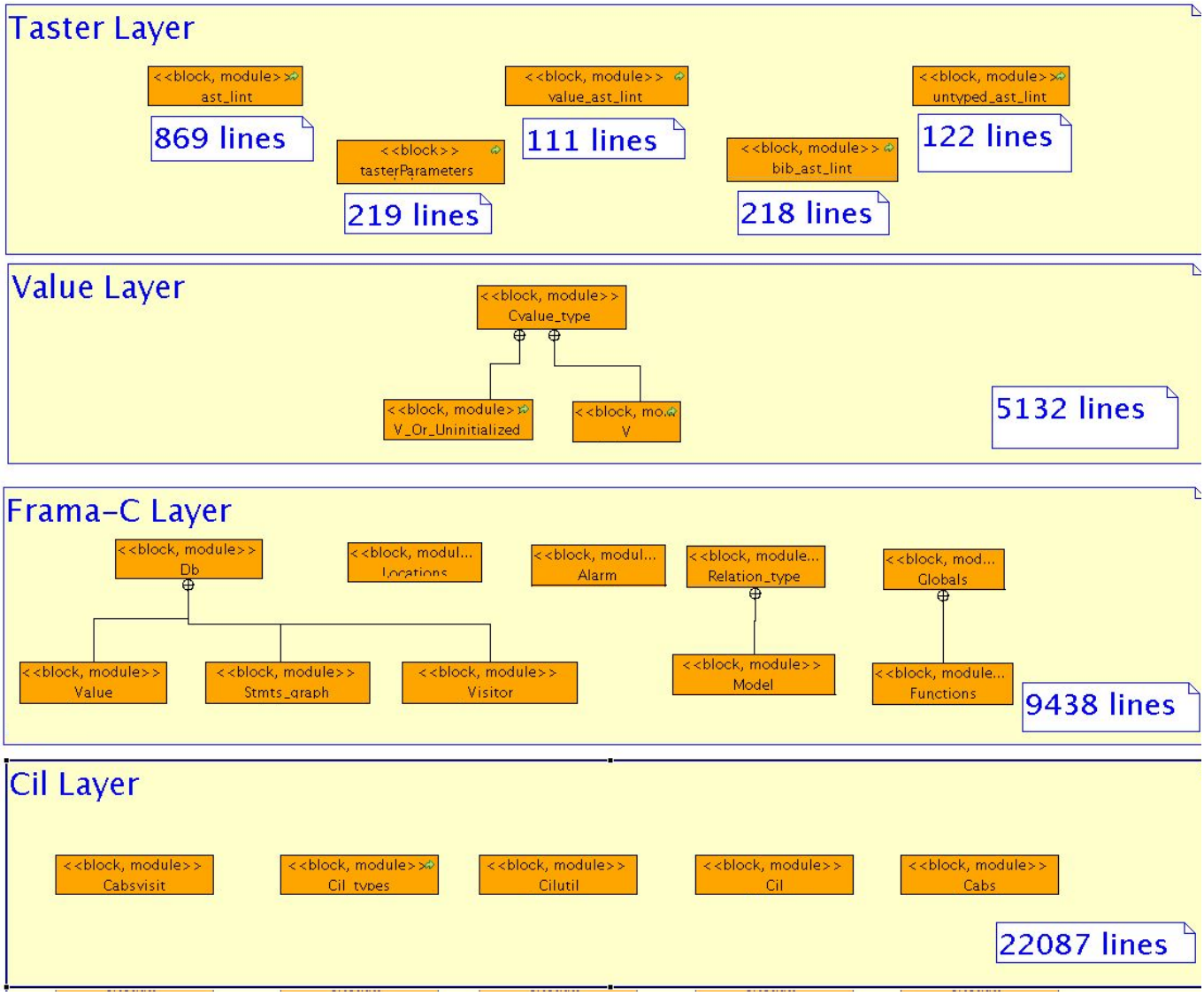

Fig. 5. Components of Taster, and relevant Frama-C layers

- re-implemented in the Taster plug-in all the necessary functionalities of the obsolete COTS tool, as explained in 2.2;
- implemented new analyses automating the verification of conformance to part of the rules that had to be verified during manual code reviews so far;
- experimented the replacement of the COTS tool with Frama-C and Taster on two operational avionics projects.

Following this successful experiment, it has been decided that the workshops for conformance to coding standards of most avionics programs should gradually migrate from the COTS tool to Taster. This deployment has already started on some avionics projects.

Consequently, Taster needs to be qualified as a verification tool wrt. DO-178B. Qualification activities are planned.

5.2 Industrial viewpoint

From an industrial perspective, the main reasons for the success of the Taster plug-in are: adaptation to needs, ease of use, cost-effectiveness, maintainability, evolutivity and qualifiability.

Adaptation to needs As explained above, Taster has been tailored to meet Airbus needs exactly. Indeed, it matches the Airbus internal coding standards precisely, while focusing on rules for which automation is most needed. For some rules, it provides several implementations of checks, allowing for several project-level interpretations of the Airbus standards. For others, it implements parametric analysis options enabling users to define and enforce local policies tuning some general Airbus rules.

Ease of use Taster may be run in batch mode, just like a compiler, with one or several analysis options per

coding rule to be enforced. Its integration to Frama-C makes it also possible to select options and visualise results on the Frama-C GUI.

Cost-effectiveness Most of the complexity of the Taster tool is in fact hidden in the open source Frama-C and Cil layers, which come "for free", so to speak. As a consequence, the development of the Taster plug-in itself was conducted within a few person-months.

Maintainability As opposed to the obsolete COTS tool, we believe there is no risk regarding the maintainability of Taster. Indeed, the open source Frama-C platform is well documented, and developed by an active community supported by durable prestigious national research institutions (CEA and INRIA). Furthermore, the underlying Objective Caml programming language is efficiently supported by INRIA, and used quite widely in France – but also abroad.

As for the Airbus proprietary plug-in Taster itself, easy maintainability is ensured by its concise and modular implementation, together with its precise documentation and thorough regression test suite.

Evolutivity The good properties of Taster's design and the complete control on the plug-in development are strong guarantees for evolutivity. Checks are implemented independently from one another, and the specific code for most of them seldom exceeds a few dozen lines.

Qualifiability The same data will ease the qualification of Taster as a verification tool wrt. DO-178B.

Note that qualifying this tool involves qualifying a particular use of the Frama-C kernel. As the Frama-C platforms includes other static analysers much more advanced than Taster, implementing formal verification techniques which we are considering using industrially in the near future, we hope to be able to simplify part of the qualification activities for these new tools thanks to the Taster service history.

Such a scheme should work for embedded software with DAL[10] C and below. However, it is not quite clear so far whether this scheme will be applicable at all for DAL A and B critical programs, as the future DO-178C tends to enforce very stringent qualification requirements for such formal tools to be used instead of classical verification methods.

5.3 Developer viewpoint

Objective Caml Using the Objective Caml programming language favors the development of the Taster plug-in in many respects:

- Object-orientation allows for visitor inheritance, which permits visiting ASTs;
- Objective Caml is a strongly typed programming language which provides legibility and conciseness. These characteristics make it easier to develop safe and robust programs in OCaml than in other languages like C;
- Objective Caml provides the pattern-matching feature. The combination of this feature with type constructors and variants is an efficient means to implement sophisticated syntactic checks in a complete, simple and readable way. With this support, the code of some checks fits into only five or six lines.
- Program implementation is close to functional features, this point helps software proofreading and maintenance;
- Frama-C and Taster benefit from the portability of OCaml on various platforms, both in bytecode and native mode;
- Owing to the native mode feature of OCaml, the Taster plug-in runs much faster than if it were written in an interpreted language like Java or Python.

All these advantages make OCaml a language suitable for an industrial purpose. That fully confirms the conclusion already made in a previous experience report [7] which did not take into account the point of view of the external plug-in developers.

Frama-C We have encountered and resolved a few difficulties along the development of Taster:

- Cil data structures do not differentiate implicit type conversions from explicit casts, so we have to use a dedicated hook to identify implicit conversions;
- As explained in 4.2, we had to cope with the unavailability of code comments in Cil and Frama-C ASTs;
- the official release[11] of Frama-C that we chose did not preserve comments. To meet one of our needs, we had to change a boolean constant in the source code of the Cil layer.
- Code normalisation by Cil changes some arrays into pointers, so we had to process a special "array size" generated annotation in order to discriminate arrays;
- Frama-C analyses pre-processed code, so we had to cope with loss of information on the use of macro-functions in the original source code.

Furthermore, handling with the Cil layer, the AST structure and the use of the value plug-in has given us a good knowledge of the overall platform.

5.4 Prospects

Extension plans for the Taster plug-in are geared towards further reducing the amount of manual work in

[10] Development Assurance Level, as defined in [1]

[11] named Beryllium 2

code reviews. This involves automating completely or partly more and more verifications, among which conformance to semantic rules.

Further reductions in code reviews Although we have already covered a little more than the scope of the legacy COTS tool, more syntactic and typing rules can be implemented rather easily into Taster, in order to further minimize the set of rules to be checked *via* manual code reviews. We are planning to add associated checks to the tool in the near future.

Formal verification of conformance to some semantic rules As explained in 2.2, little support is available so far to ensure conformance to semantic rules: only (unsound) bug-finding heuristics can be used in a debugging phase, hence the need for sophisticated manual code analyses.

Yet Frama-C includes semantic analysis plug-ins which may be used to partly automate these tasks. As shown in 4.3, we have started experimenting the use of such features within Taster. We are willing to pursue this work, in order to derive sound and "nearly automatic" formal verifications for part of Airbus semantic rules. First candidates could be dataflow-oriented rules, for which imprecisions due to over-approximations performed during (non relational) abstract interpretation based static analyses should not be a big issue. Industrial uses of such analyses will involve providing correct context information by means of annotations or stubs, without increasing the workload of developers. We believe this should be feasible rather automatically in our context, thanks to relevant information to be extracted from design models.

Besides, implementing such semantic analyses into Taster and qualifying their industrial usage wrt. DO-178B might provide a useful basis towards progressively using more and more advanced features of Frama-C for the verification of embedded avionics software.

Références

1. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Inc., 1992.
2. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language (preliminary design V1.4)*, preliminary edition, October 2008.
3. Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. *Frama-C User Manual*, November 2009. http://frama-c.cea.fr/download/user-manual-Beryllium-20090902.pdf.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, US, 1977. ACM Press.
5. Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In *IFIP Congress Topical Sessions*, pages 359–366. Kluwer, 2004.
6. Pascal Cuoq and Virgile Prevosto. *Frama-C's value analysis plug-in*, October 2009. http://frama-c.cea.fr/download/value-analysis-Beryllium-20090902.pdf.
7. Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 281–286, New York, NY, USA, 2009. ACM.
8. Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. In *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, pages 128–140, 1741. Based on a talk presented to the Academy on 26 August 1735.
9. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system.* http://caml.inria.fr/pub/docs/manual-ocaml/index.html.
10. Édouard Lucas. *Récréations mathématiques.* A. Blanchard (Paris), 1891.
11. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.
12. Julien Signoles. Foncteurs impératifs et composés: la notion de projet dans Frama-C. In Hermann, editor, *JFLA 09, Actes des vingtièmes Journées Francophones des Langages Applicatifs*, volume 7.2 of *Studia Informatica Universalis*, pages 245–280, 2009. In French.
13. Julien Signoles, Loïc Correnson, and Virgile Prevosto. *Frama-C Plug-in Development Guide*, September 2009. http://frama-c.cea.fr/download/plugin-developer-Beryllium-20090902.pdf.