Spreading Static Analysis with Frama-C in Industrial Contexts

A. Stéphane Duprat[1], B. Victoria Moya Lamiel[1],
C. Florent Kirchner[2], D. Loïc Correnson[2],
E. David Delmas[3]

1: Atos, 6 Impasse Alice Guy, B.P. 43045, 31024 Toulouse Cedex 03
2: CEA LIST, Software Safety Laboratory, Saclay, F-91191
3: Airbus Operations S.A.S., 316 route de Bayonne, 31060 Toulouse Cedex 9

Abstract: This article deals with the usage of Frama-C to detect runtime-errors. As static analysis for runtime-error detection is not a novelty, we will present significant new usages in industrial contexts, which represent a change in the ways this kind of tool is employed.

The main goal is to have a scalable methodology for using static analysis through the development process and by a development team.

This goal is achieved by performing analysis on partial pieces of code, by using the ACSL language for interface definitions, by choosing a bottom-up strategy to process the code, and by enabling a well-balanced definition of actors and skills.

The methodology, designed during the research project U3CAT, has been applied in industrial contexts with good results as for the quality of verifications and for the performance in the industrial process.

Keywords: Static analysis, abstract interpretation, safety critical software, embedded system, modular analysis, Frama-C, ACSL, runtime error

1. Introduction

Static analysis for runtime-error detection is not totally new; different tools have been proposed since fifteen years. Nevertheless, it is not a widespread practice even in critical software. Static analysis is commonly employed by specialists for independent verifications and after the development of the program. This activity is a good way of improving quality but it is often synonym of additional activity in the main process and additional costs.

In order to facilitate the usage of static analysis, we conducted during the research project U3CAT, methodological studies and tooling development. The main objectives were: good coverage of runtime-errors, scalability, predictable costs, and a good integration in the development cycle.

We largely succeeded in this endeavour: this article reports on the main difficulties encountered, the technical and methodological solutions adopted, and the benefits obtained.

The produced methodology has been updated and used to answer to specific needs in industrial contexts and we report this industrial experience.

Finally, we'll conclude on new usages already identified, but not yet used in industrial context.

2. Context

The Frama-C source code analysis platform

Frama-C is an Open-Source platform dedicated to the analysis of C programs. It differs from other code analysers as it provides a diverse set of formal tools, cooperating through code annotations written in the ACSL language. ACSL is a behavioural specification language that can express a wide range of functional properties, through partial or complete specifications. Analysers themselves may report results in terms of new ACSL properties asserted inside the source code.

Frama-C [6] is built around a kernel that performs the parsing and type-checking of C code and accompanying ACSL [5] annotations if any, and maintains the state of the current analysis project. This includes in particular registering the validity status of all ACSL annotations. Analyses themselves are performed by various plugins that can validate annotations, but also emit hypotheses that may eventually be discharged by other plugins. This mechanism allows some form of collaboration between the various analysers.

Two important analysis plugins are Value Analysis [7] and WP [8]. Value analysis is based on abstract interpretation, and computes an over-approximation of the values that each memory location can take at each program point. When evaluating an expression, Value Analysis will check whether the abstraction obtained for the operand represents any value that would lead to a runtime error[1]. For instance, when dereferencing a pointer,

[1] Rutime errors include: division by 0, undefined logical shift, overflow, underflows on integers, use of non-initialized variable, dangling pointer, invalid memory access , use of non-allocated pointers, problem of overlapping lvalue assignment, undefined side-effect in expressions, and invalid function pointer access.

the corresponding abstract set of location should not include NULL. If this is the case, Value Analysis emits an alarm, and attempts to reduce the abstract value. In our example, it will thus remove NULL from the remaining abstract state. The analysis is correct, in the sense that if no alarm is emitted, no runtime error can occur in a concrete execution. It is however incomplete, in the sense that some alarms might be due to the over-approximations that have been done and might not correspond to any concrete execution. Various settings can be selected to choose the appropriate trade-off between the precision and the cost of the analysis. While the most immediate use for Value Analysis is to check for the absence of runtime error, it will also attempt to evaluate any ACSL annotation it encounters during an abstract run. Such verification is however inherently limited to properties that fit within the abstract values manipulated by Value Analysis. Mainly, it is possible to check for assertions on bounds of variables at particular program points.

WP is a deductive verification-based plugin. Contrary to Value Analysis, which performs a complete abstract execution from the given entry point, WP operates function by function, on a more modular basis. However, this requires that all functions of interest as well as their callees be given an appropriate ACSL contract. Similarly, all loops must have corresponding loop invariants. When this annotation work has been completed, WP can take a function contract and the corresponding implementation to generate a set of proof obligations – logic formulas whose validity entails the correction of the implementation with respect to the contract. WP then simplifies these formulas, and sends them to external automated theorem provers or interactive proof assistants to complete the verification. WP's main task is thus to verify functional properties of programs, expressed as ACSL annotations. It is however also possible to use it to check that the pre-conditions written for a given function f imply that no runtime error can occur during the execution of f.

Frama-C is already used in this industrial context. First usage at Airbus is for an implementation of a coding rule checker called Taster [3] and a second one, Fan-C [4], targets verification of data and control flow based on semantic analysis.

Main principles of a static analysis project

One of the main principle of Value Analysis-based projects is that to computes values of variables for all possible program execution, either starting from the program's 'main' function or another one expressed to the tool by an option on the command line.

The Value Analysis handles the semantics of the C program, but not only. One strength of the tool is to be able to perform analyses on incomplete programs, that is, pieces of source code not containing all definitions of functions called.

The user can define function contract in ACSL defining behaviour of the function and the tool is able to integrate, within its analysis, the semantic of the C program and the semantic of the ACSL.

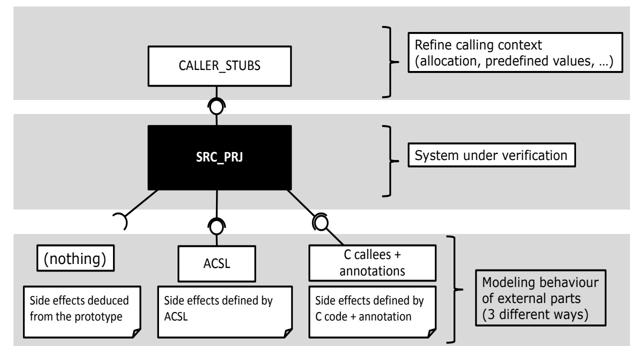


Figure 1 : Topology of a static analysis project

For a C function without C definition neither ACSL contract, the tool is able to consider a default behaviour deduced from its prototype.

Finally, the user has three solutions for an external function : (1) nothing, (2) an ACSL contract, (3) a callee stub written in C language and which could use specific Frama-C builtin functions (see Figure 1 : Topology of a static analysis project).

3. Modular analysis

The modular analysis consists in using Value Analysis on small pieces of programs in a consistent manner.

The ACSL language, by defining the behaviour of software interfaces, facilitates the analyses of independent parts of software. Properties defined in the interfaces can be used in two ways: for verification purpose on one hand and for hypothesis definition on the over hand.

This small and trivial example can illustrate both usages:

```
/*@
  requires \valid(p);
  assigns *p;
  ensures \initialized(p);
  ensures 0<= *p <10;*/
extern void get_index(int* p);

/*@ requires \initialized(&x);
  assigns \nothing;*/
extern void bar(int x);
```

```
int foo(int p[10])
{
  int index ;
  int status ;
  get_index(&index);
  bar(index);
  return p[index];
}
```

Figure 2

The called functions get_index and bar are not defined by their source code, but by partial ACSL contracts.
For get_index function:

- For verification purpose at the calling context:
  - The *requires* clause is able to verify that pointer p is valid (referencing an allocated memory)
- To introduce some hypothesis on the behaviour of the function
  - The *ensures* clauses define some hypothesis on the outputs: at the return of the function, the value pointed by b is initialized and in the range [0;10[.
  - The *assigns* clause specifies side-effects. It can be used to define side effects on other locations than those identified by the function parameters.

For bar function

- For verification purpose at the calling context:
  - The *require* clause check that the value of parameter x is initialized
- To introduce some hypothesis on the behaviour of the function:
  - There is no side-effect defined by the *assigns* clause (pure function)

In this way, the verification of the function foo can be performed on this function alone and under the hypothesis of the correct behaviour of the called functions defined by ACSL contract. Verifications are targeting the behaviour of the function itself and also the calling contexts of the called functions. The same contracts of the called functions will then be used in their proper analysis.
Considering the above example, the verification of “foo” doesn’t need the source code of its callees (“get_index” and “bar”), as the ACSL contracts for both callees are sufficient for the analysis. Besides the verification reaches all contexts, including the function behaviour and the calling contexts of the callees, these ACSL contracts will be used lately during their corresponding analysis.
In this way, the different functions i.e. the different subsets of software can be developed and analysed independently and consistently thanks to the function contracts in ACSL.

ACSL contracts can be used not only in the detection of RunTime-Errors, but also on more specific objectives. For example, the verification of functional ranges of parameter values can be verified in this way.

While this approach applies to Value Analysis, ACSL is also able to define functional properties that will be verified with WP using deductive proof techniques as stated in §2.

4. The bottom-up strategy

One of the main issues of static analysis tools in general is that they can produce many false alarms and/or take large amounts of time. The amount of effort necessary is thus difficult to evaluate before performing the analysis. These unpredictable costs and technical difficulties can dampen industrial applications and contractual commitment possibilities.

The most trivial technique to analyse a software is to perform analysis of the whole program completed by a definition of the called libraries. This strategy can be a winning one in case of immediate success. But success is not guaranteed and engineers can be stucked by the number of false alarms and computation time.

One alternative is to divide the program in smaller pieces, to add ACSL contracts in the subset interfaces and to conduct a modular analysis. This strategy can be a solution to obtain a good result, but defining ACSL contracts in a reverse engineering work can be a costly activity.

Facing these difficulties, we defined a bottom-up methodology aiming at succeeding in the analysis of a whole class of programs and with predictable costs.

This methodology is based on an exhaustive analysis of each function with a bottom-up progression. The lowest layer is analysed first and all issues are handled in order to make dispense of all warnings. Different actions are possible: correction in case of bug, fine-tuning of analysis parameters of the tool, or addition of ACSL clauses to help the tool in case of inaccuracy. Once the first layer is treated and all the analyses raise no alarm, each iteration will consist in the integration of sources of the upper layer in their analysis. These iterations will end when reaching the top of the program. This progressive approach has been successfully applied in several analyses. Two experienced uses cases are presented in the §5.

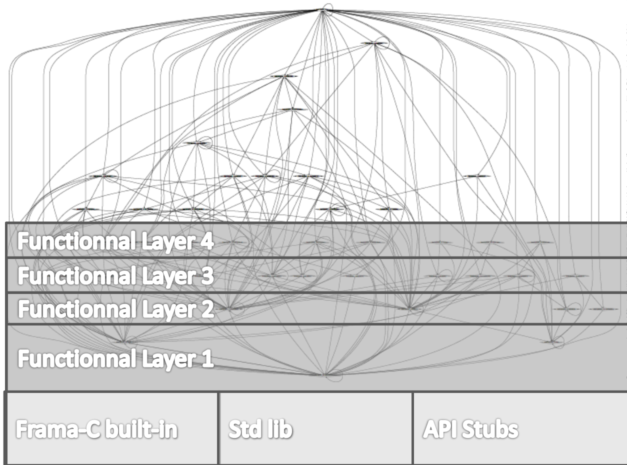


Figure 3: Bottom-Up Strategy applied on 40 C files

By using such a bottom-up strategy, programs that were difficult to analyse as a whole can be analysed step by step up to the main or the entry function.

The opposite strategy of the bottom-up strategy is the top-down strategy consisting in integrating all the source code and performing the analysis on the main function. This strategy can be successful if the whole program is entirely and quickly analysed, otherwise, it leads to the situation previously described and that we encountered where the user is stuck by a large amount of false alarms.

5. Experience report

We report the application of this Bottom-Up strategy on two subset of software. Both are aeronautical software in an intermediate stage of development. Complexity of these two subsets is presented in table below.

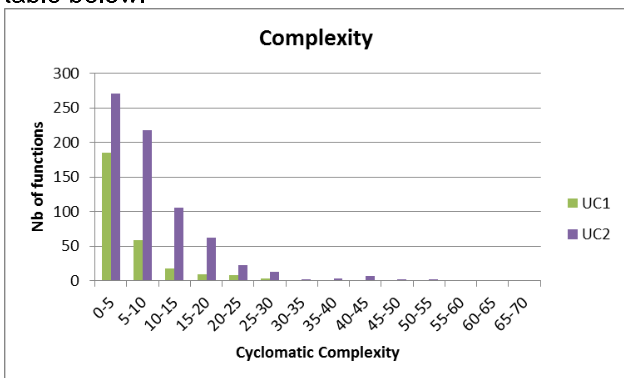


Figure 4

The size of these uses cases are 20 kloc for UC1 and 55 kloc for UC2. UC2 contains complex types considering nested structures, pointers and arrays and with a lot of string manipulations. This higher degree of complexity of UC2 is confirmed by the cyclomatic complexity figures computed by Frama-C. 95% of functions of UC1 have a complexity <15 while 95% of functions of UC1 have a complexity <20.

Objectives

Verification objectives are not expressed in terms of static analysis techniques but by a need of detection of the following threats:

- T1: usage of a non-initialized variable
- T2: usage of a non-initialized pointer
- T3: out of bound access of an array element
- T4: uninitialized output value of a function
- T5: usage of the address of a local value out of the scope of its declaring function
- T6: string management

Solution

A dedicated methodology based on the use of Frama-C's Value Analysis plugin has been proposed to handle all these objectives. Some of these threats are directly handled by usage of the plugin Frama-C/Value. Some additional artefacts have been deployed in order to achieve other objectives.

For example, some callers (as instrumented function on the verification project) have been generated in order to implement the verification of the complete initialisation of all output variables. This mechanism is illustrated for threat T4 in the example below:

```
typedef struct { int a; int b;} S;

void f(S * s1)
{
  s1->a=0 ;
  return ; // filed s1.b is not assigned
}
```

Source code to verify

```
// validation function for f
void caller_f(void)
{
  S l_param1;

  // call of f with parameters to an
uninitialized state
  f(&l_param1);

  // verification that l_param1 i initialized
  //@assert \initialized(&l_param1);
}
```

Caller generated

Figure 5

In this situation, the tool is able de detect that s1 parameter is not fully assigned in the function f.

Application

Following the bottom-up strategy, all functions have been individually analysed, starting from the lower layers.

For each analysed file, the steps are the following

1- Prepare the environment
   a. Define a caller for every analysed function
   b. Define a correct stub for each called function
2- Launch a batch analysis on all functions of the files of the layer
3- Analyse each generated warning
   a. In case of real warning,
      i. Mention the issue on a report file
      ii. Fix the issue
   b. In case of a false warning due to a tool inaccuracy
      i. Help the tool by adding some ACSL annotations in the source code or by using specific options
   c. In case of absence of warning, the analysis of the layer are finished, go the next file
4- Without any warning, the process is finished for this layer, go to the upper layer. Launch a new analysis in case of remaining warnings (go to step 2).

Notes:

- Minor modifications of source code have been made to accelerate the analysis. These modifications concern almost reductions of array size.
- The semantics of parallel execution of the multithread program are not handled

As for the industrial organisation, the analysis has been conducted by a team with the support of an expert. Two persons have been involved in UC1 and three for UC2. Each person has been working on different files. The next files to be analysed were determined with the aid of a "module call graph" consolidated with the list of the already analysed source files and always following a bottom-up progression. All issues reported were checked by another team member and periodical technical meetings were organised.

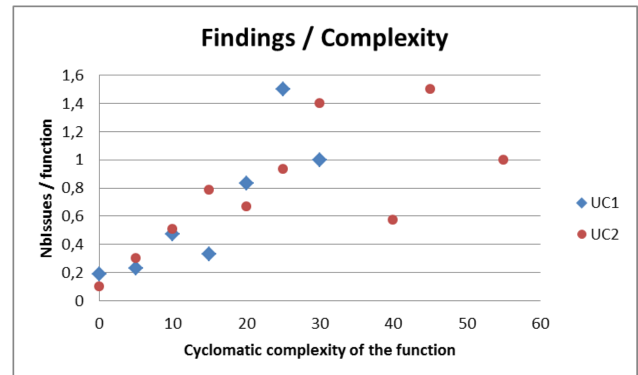
Results

All source files have been analysed. The total number of all findings and their proportions compared to the number of lines of code is presented in the table below.

	UC1	UC2
Findings	54	232
Kloc	17500	57500
Findings / kloc	3,1	4,0

Proportions of findings between both use cases are comparable. Number of warnings in proportion is normally higher for UC2 which is more complex and including string computation.

A closer look reveals a coherent relation in both Use Cases of the medium number of findings per function depending on the cyclomatic complexity of the function. This repartition is quite linear in UC2 for complexity up to 20.



Proportions of the different warning categories are presented in the diagram below.

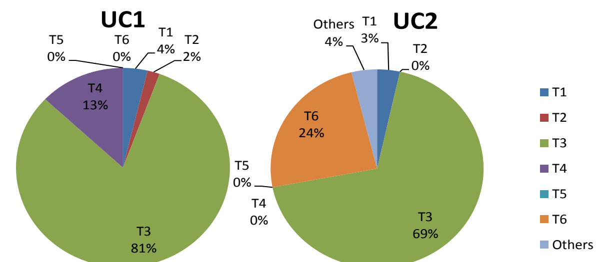


Figure 6

The great majority of findings are concerning array indexes. As the verification is done at a unitary level, these warnings are not meaning that there are as many real bugs detected. It means that indexes of arrays are not checked at each level of functions (this could be at most a lack of robustness, but definitively not necessarily a bug).

Both use cases are diverging on full initialisation of output values: while it represents the second most important category of warnings on UC1, none of them are detected on UC2 thanks to some specific preventive actions conducted on this subset. No warning on string operations are detected on UC1 because of the absence of strings in this subset. One of the most interesting lessons learnt is about the verification of string computation widely used in UC2. String computation is a hard point of verification by static analysis. This experience demonstrates that, under the assumption of some good coding practices (for example: strncpy instead

of strcpy) the software can be analysed with enough precision to limit the amount of false alarms.

6. Feedback

Feedback on the Modular analysis strategy

The modular analysis strategy can be a solution for scaling up in the face of major difficulties with full program analysis. But the definition of ACSL contracts of the subset can be a costly activity if it is done in a reengineering process. Another drawback is that some implicit hypotheses that are taken for the analysis of the different modules and not always correct for the whole program. For example, the different locations referenced by pointer parameters of one function are considered as separated locations; that is not always the case. This can lead to an unsound analysis.

On the other hand, if the modular analysis is applied early during development, just after the coding stage and if contracts are already defined as a part of the design, this becomes a good strategy for increasing rapidly software maturity. Each developed subset can benefit from this valued added approach without waiting for the development of all pieces of software. After software integration, the enhanced quality of the produced code will facilitate an analysis of the whole program that will provide the highest level of trust.

A balance can also be made for stub definition between a solution based on ACSL contracts and a solution based on a full C language definition of stubs. The C language offers multiple ways to define a representative behavior of a called function. One advantage is to use only one language for the user and for the tool too. But as presented in §3, the same ACSL contract (ex: `"ensures 0<= *p <10;"`) used as a property of the called function on one hand is also used as a property to verify on the real implementation of the called function. This duality is not allowed by a C definition of a stub.

Feedback on the Bottom-Up strategy

At the provider side (Atos), who proposed the solution and applied it. The bottom-up approach enabled engineers to analyse the entire software cost efficiently and within their deadlines. This success can be explained mainly by the good scalability of this method and by the industrial organisation enabling several actors to work on the same software.

At the side of the industrial customer (Airbus), requester of this analysis, all the verifications requested have been reached and with a very high degree of confidence due to the application of a static analysis by abstract interpretation solution.
For UC2, a large part of findings are concerning a lack of robustness without safety consequences. Less than 10% have led to a correction in the source code. Very few issues are related to actual bugs with operational impact, which have all been also detected during a simultaneous test campaign.

The results demonstrated the validity of the approach and the ability to detect threats very hard to debug during the software development phase. Considering the reported warnings and the verification capacity, the return on investment would have made it worth applying this strategy earlier in the process. An earlier detection of safety issues would have saved costs in validation efforts.

Other considerations

The main advantage of the bottom-up strategy lies in its scalability. But it is also a way to conduct dedicated verifications on each function that is not accessible through the analysis of the whole program in one shot.

The need for this verification depends on the industrial development process. The user can be interested only in detecting runtime-errors that can really occur in its operational situation, with the whole code integrated. On the other hand, the user can be interested in verifying some properties that shall be assumed by each source function independently to address maintainability and portability concerns (that was the case of the reported UC in §5).

For example, the situation presented in Figure 5 may not impact the program if the callers of these functions are not using the non-initialised field. This situation may not be detected during the analysis of the entire program. Considering this function specifically, returning an initialized value of each output operand in any situation can be something expected and required by the coding rules of the project. In this case, the bottom-up strategy is a way of verifying these properties for each function independently.

Roles and development methodologies

During the evaluation studies, we targeted an industrial process to gain maximum benefit of the usage of static analysis in an industrial process. We targeted the phase of the development process and not a terminal phase of an independent verification.

To the question “*When?”*, the answer is as early as possible in the development process. The analysis is best run during the coding phase and before the tests. The expected benefits are a quality assessment of the source code, a reduction of the costs of the tests and a better quality of the final product.

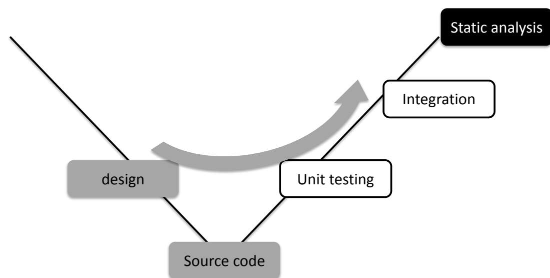


Figure 7: Formal verification of the whole product

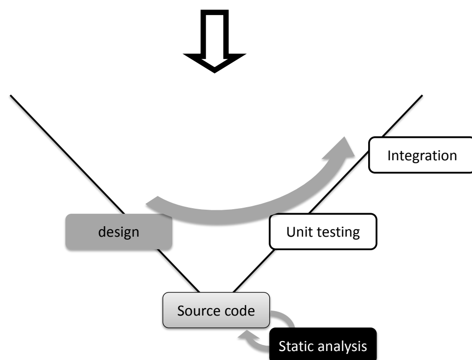


Figure 8 : Formal verification integrated in the coding phase

Another question is “*Who is doing the analysis?*”. Starting from a situation where analyses are conducted by experts (Figure 7: Formal verification of the whole product), we tried to integrate as far as possible the developer in the process of verification of code by static analysis. The idea was to obtain a continuous improvement of the source code and an availability of analysis produced in a short time. Real feedbacks indicated us that the lack of skills and experience in static analysis could be detrimental. Indeed, a beginner in static analysis could be stuck on a difficulty of understanding or a weakness of the tool and spent too much time trying to get unstuck ; at worst the developer can denigrate the solution. In reaction, we quickly focused to a solution mixing skills of developers and experts in static analysis. The best way to integrate experts in static analysis in the development team is to have them prepare the verification projects and to run the first analysis. Once the project is in place, it can be appropriated by the developers. Facing problems in the analysis, experts are able to quickly find a fine tuning of the tool, a fix or a workaround and thus keep in the productivity targets. Work of experts is synchronized with the team developments by using a version control tool. In this way, we are able to keep the benefits of an integrated process and the efficiency of specialized actors.

Simultaneously to these methodological works, tooling works have been done to facilitate this application: automatic generation of stubs, module dependence analysis for retro design, makefile, and also integration in CDT Eclipse, Client/Server prototyping.

6. Conclusion

Static analysis is still a disruptive technique for verification, as it is not yet largely applied. We have demonstrated that methodological efforts can open new areas of applications. This article reports on how we applied the technique with a specific strategy and with actors that facilitated their usage in development projects. Another step would be to systematically apply these verifications during the coding stage and to capitalise on them during post-development activities, in a fashion similar to testing. If we are considering the industrial organisation, including the industrial customers, subcontractors, near-shore and off-shore, and academic laboratories, sharing the analysis projects from the first developers to the end-users can be a solution to improve quality with an optimized cost. Finally, the use of static analysis for runtime-error detection can be a first step before employing other techniques as deductive proof for functional verification.

7. References

[1] Antoine Miné and David Delmas. Towards an industrial use of sound static analysis for the verication of concurrent embedded avionics software. In EMSOFT: To appear in proc. of the 15th International Conference on Embedded Software, 2015. IEEE CS Press.

[2] Pascal Cuoq, David Delmas, Stéphane Duprat, and Victoria Moya Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In ERTSS 2012: Proceedings of Embedded Real Time Software and Systems. SIA, 2012.

[3] Jean Souyris, David Delmas and Stéphane Duprat. Airbus : vérification formelle en avionique. In Jean-Louis Boulanger, editor, Utilisations industrielles des techniques formelles : interprétation abstraite. Hermes-Lavoisier, June 2011.

[4] David Delmas, Stéphane Duprat, Victoria Moya Lamiel, and Julien Signoles. Taster, a Frama-C plug-in to enforce coding standards. In ERTSS 2010: Proceedings of Embedded Real Time Software and Systems. SIA, 2010.

[5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto: “*ACSL:*

ANSI/ISO C Specification Language (V1.9)”, http://frama-c.com/download/acsl_1.9.pdf, 2013.

[6] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, Boris Yakobowski: *“Frama-C: A software analysis perspective”*. Formal Asp. Comput. 27(3): 573-609 (2015)

[7] Pascal Cuoq and Boris Yakobowski with Virgile Prevosto: Value Analysis, February 2015. http://frama-c.com/download/value-analysis-Sodium-20150201.pdf

[8] Pascal Cuoq and Boris Yakobowski with Virgile Prevosto: Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, February 2015. http://frama-c.com/download/wp-manual-Sodium-20150201.pdf

9. Glossary

ACSL: ANSI/ISO C Specification Langage