DEFINING A PRAGMATIC METHODOLOGY FOR SOFTWARE ASSESSMENT BASED ON A « WHITE-BOX » APPROACH

DEFINITION D'UNE METHODOLOGIE PRAGMATIQUE D'EVALUATION DE LOGICIEL FONDEE SUR UNE APPROCHE « BOITE BLANCHE »

Lucas DUBOC et Franck SADMI
BUREAU VERITAS
60 Avenue du G dé Gaulle
92800 Puteaux

Florent KIRCHNER
CEA LIST
8 Avenue de la Vauve
91120 Palaiseau

Summary
The present paper sums up the work and results of an on-going partnership program between BUREAU VERITAS & the CEA LIST, started in 2015, whose main objective is to propose a generic and pragmatic methodology in the form of a set of guidelines for software development and assessment.

Initiated by the observation of an absence of dedicated software safety assessment standards and tools in several industrial sectors, and benefiting from the BUREAU VERITAS knowledge of software safety certification standards and the CEA LIST expertise in software code verification, these guidelines are focusing on development processes and the use of efficient tools to verify software through a white-box approach.

They have been designed to be usable in all the domains that deal with the quality, reliability and/or safety of software systems, especially in sectors where safety concerns are not prevalent (e.g. car amenities - multimedia, air conditioning, etc.-, robotics and smart devices - automated handling, connected bracelets and watches, etc.-) or those lacking existing regulations (e.g. marine & offshore, part of the defense sector, industrial and agricultural machines, etc.).

Résumé
Cet article récapitule le travail et les résultats d’un programme de partenariat entre BUREAU VERITAS & le CEA LIST, initié en 2015, dont l’objectif principal est la proposition d’une méthodologie générique et pragmatique sous la forme d’une liste de recommandations pour le développement et l’évaluation de logiciels.

Partant du constat qu’un certain nombre de secteurs industriels ne bénéficient pas de référentiel dédié ni d’outils associés pour l’évaluation de la sécurité du logiciel, et profitant de la connaissance par BUREAU VERITAS des référentiels de certification de la sécurité du logiciel et de l’expertise du CEA LIST en vérification de code logiciel, ces recommandations se concentrent sur les processus de développement et l’utilisation d’outils efficaces pour vérifier le logiciel par une approche « boîte-blanche ».

Elles ont été élaborées pour être applicables dans tout domaine s’appuyant sur la qualité, la fiabilité et/ou la sécurité des logiciels, notamment dans les secteurs où les préoccupations liées à la sécurité sont moins prégnantes (ex. des équipements automobiles – multimédia, air conditionné, etc.-, la robotique et les appareils intelligents – systèmes de manutention automatisés, montres et bracelets connectés, etc.) ou ceux qui ne possèdent pas de réglementations existantes (ex. la marine & offshore, une partie du secteur de la défense, les machines industrielles et agricoles, etc.).

1. Context of the partnership

1.1. Importance and diversity of software assessment schemes

Digital technologies have overwhelmed industrial and domestic markets, and software systems are underlying almost all technical products. Software gradually replaces mechanical components. Simple or complex, the behavior of any electronic equipment is directly controlled by a piece of software, whose failure can lead to severe human, strategic, economic, or environmental consequences.

In industrial sectors such as aviation, nuclear energy, and railway, where life- and safety-criticality have historically represented the main concern, advanced functional safety standards have become conditions to access the market. As safety is their prime concern, these standards are very demanding for the organizations that have to conform to their objectives. Nevertheless, their high level of requirement can hardly apply to a majority of markets where it would be unthinkable to allocate as far as 20% of the overall software development projects to certification costs.

On the other hand, for sectors where safety aspects are secondary, best practices of software development are widespread without being gathered in dedicated guidelines.

This is the case within the marine & offshore industry, which constitutes the historical core business of BUREAU VERITAS, where the International Association of Classification Societies IACS is responsible for the establishment of standards to verify and assess ship safety.

In this market, the standard requirements related to the assessment of ship software components were generic and limited so far. Unified Requirements (IACS, 2010) had been dealing with high-level requirements for software assessment, typically test
communication at system level. Yet experience gathered from the aforementioned best practices for critical systems has shown that a system-level testing approach is too limited — either in terms of coverage, or conversely in terms of cost-efficiency — when verifying software that implements rich functionalities and dealing with multiple inputs / outputs. What’s more, the more complex the software system is, the more costly it is to perform verification activities, and the later undetected defects may remain. Finally, once put into service, modifications are often applied to software components to provide new functionalities or correct detected errors. Such changes also bring in complexity and are potential sources of flaws within the software system.

For all these reasons, verification activities providing assurance that the software system will function correctly as intended during all its life — from commissioning to decommissioning of its host equipment — have to be carried out throughout software development activities. Intrinsic complexity of the software system has to be taken into account and processes of development have to be adapted accordingly.

1.2. State of the art of software code verification methods

Prompted by criticality concerns, various approaches to software assessment have been investigated by the verification community. In particular Software Assessment tools focus on detailed program analysis techniques, i.e. tools and methods that provide software developers and auditors with strong and demonstrable confidence in their artefacts. Overall, these techniques allow the safety experts to verify that programs and their functionalities behave according to their specification. These techniques can be further divided into two classes:

1. The analysis of programs in a non-runtime state, called static analysis, proceeds across the source code just like a compiler would, looking into the program’s operation and internal structure for patterns indicative of unexpected behaviors.
2. The analysis of programs in a runtime state, called dynamic analysis, runs the compiled source code on sets of predefined input values to detect unexpected behaviors.

Different types of static analysis techniques can be applied to perform code assessment, some of them based on formal methods. Two main trends have emerged, both using abstractions of the code in order to formally verify particular classes of properties.

1. A first approach, symbolic execution (which broadly speaking encompasses deductive verification and model checking), uses logic solvers that relate code blocks to their expected behaviors, as expressed by specific logical assertions in the source code.
2. A second approach, called abstract interpretation, computes for each variable of the program an abstraction of the values it can take during any program execution, warning when it encounters unexpected behaviors.

Over the past decade, these types of program analyzers have been successfully applied to demonstrate the safety of life-critical systems. Program verification is used in this field to achieve demonstrably equivalent levels of safety as those attained with traditional methods for critical systems, but at a lower cost (Randimbivololona, Souyris, Baudin, Pacalet, Raguideau, & Schoen, 1999).

These practices have been successfully used in the context of various domain-specific certification standards (DO-178B/C, CENELEC EN 50128, IEC 60880, IEC 61508, ISO 62304, ISO 26262, etc.).

Frame-C is a source code analysis platform for C99 ISO C programs, developed by CEA LIST and its partners. It implements both static and dynamic source code analyzers as modular plugins (see section 3.2). Frame-C differs from other static analyzers as it provides a diverse set of formal tools that cooperate through code annotations emitted in the ACSL language (Baudin, Filliâtre, Marché, Monate, Moy, & Prevosto, 2015).

1.3. Stakes of the integration of automated software verification tools within third-party certification programs

As presented here above, the theory of code verification offers a breakthrough in terms of software systems demonstration of reliability and fault tolerance. For a third-party assessment authority, the understanding of its production potential of formal proof material becomes key to adjust and optimize the software certification programs. However, this appropriation process has to face the strong complexity of an advanced research & technology field.

Establishing links between the esotericism of formal methods researches and their application in concrete software verification projects has always been a major purpose of CEA LIST institute. The created partnership with a certification authority like BUREAU VERITAS represents a step forward towards the democratization of these methods and results interpretation among the industrial stakeholders.

Currently the important issue for certification inspectors is to interpret the results of software analysis techniques and tools in terms of compliance with the certification requirements, and to leverage them toward their assessment goals. Formal software analysis capabilities need to be developed internally and software verification tools have to be adapted and qualified with regards to their added value in the certification process, the two main criteria being time-saving and flaw-coverage.

In the meantime, software verification tool providers must demonstrate that their tools perform as expected by the certification process. In avionics or in the nuclear sector for instance, the qualification of software verification tools is a well-codified process that can rely on a basis of reusable elements. Nevertheless these types of process are highly dependent on the context of the certification, and can hardly be completely automated.

1.4. Shared objectives of the partnership

Both BUREAU VERITAS and CEA LIST deal everyday with some of most of the critical industrial sectors and associated standards for software development, and have a strong experience of the best practices in these different environments. As said in section 1.1, there currently exists little guidance when it comes to software verification in marine & offshore applications, but
plenty of experience on lifecycle certification. Furthermore, as presented in section 1.2, recent software analysis techniques have demonstrated their relevance in software verification for product assessment schemes.

It is with this dual assessment methodology in mind – process oriented verification and source code analysis – that BUREAU VERITAS and CEA LIST partnership has been working on SOFTWARE DEVELOPMENT & ASSESSMENT GUIDELINES. This document, whose salient features are presented in section 2, provides minimum objectives that should be met by any software system that needs to demonstrate its ability to achieve an expected performance level, thus satisfying common safety concerns.

On the one hand, it had to be seen as guidelines of development and did not aim to supersedes existing safety standards that are regulatory in some specific industries like railway or aeronautics. On the other hand, those guidelines had to be fit to constitute the reference standard used by BUREAU VERITAS when assessing software systems in marine & offshore in order to establish confidence in terms of safety. The guidelines were designed to be relevant whatever the programming language used for the software system development. No programming language is favored as long as its dangerous or complex instructions is identified and controlled. Even if the guidelines recognized the use of tools, no specific solutions are identified: each project would choose dedicated tools depending on its objectives and potential requests from the certification authorities.

Furthermore, the CEA LIST and BUREAU VERITAS partnership has been working on delimiting and characterizing the evidence artefacts that could be obtained from the Frama-C plugins run on pieces of code, based on the adopted plugin options and the type of verification requirements. This resulted in the delivery of a practical guide describing the methodology used to run Frama-C’s Value analysis on C source code bases. This methodology supports the activities required by the BUREAU VERITAS SOFTWARE DEVELOPMENT & ASSESSMENT GUIDELINES (BV-SW 100) in its Sections 6.5 “Checking of Software Units” and 6.6 “Integration of Software Components”. Each chapter in this document corresponds to one of the main steps of a complete analysis using the Value plugin:

1. preparation of the source code base: choosing the relevant files, preprocessing, and parsing;
2. performing a first analysis: a quick but imprecise first analysis of the code source, to identify possible issues with the source code preparation step, and to have a general estimation of the analysis;
3. refining and improving the analysis: fine-tuning of the several options available, to improve precision while keeping analysis time under control; reducing the number of false alarms and improving analysis coverage.

2. Originality of the software assessment methodology

2.1. Guidelines principle

Simplicity was a major objective during the writing of the SOFTWARE DEVELOPMENT & ASSESSMENT GUIDELINES. The authors strived to define a practical methodology that optimizes both efforts of software development and software assessment. The small size of the document (with a goal fixed at a maximum of 50 pages from the beginning of the development of these guidelines) is designed to ease its use and circulation. The common thread leading the partnership project has been the delivery of practical and easy-to-use set of best practices and recommendations to ensure that software development meets the twofold requirement of optimal operability and high-risk tolerance.

One of the main pitfalls related to this conciseness approach is the possibility of important requirements omission. That is why each recommendation choice and wording has been conscientiously validated by the Operational Safety joint working group constituted by BUREAU VERITAS and the CEA LIST institute. Furthermore, the documented methodology has followed the same philosophy and structure as the main software safety-related standards that are enumerated in section 1, even though it does not attempt at a thorough comparison between these various standards - such as the work of (Ledinot, et al., 2014).

The main idea in software development is to keep in mind that quality and safety are properties progressively achieved by construction, following a process to avoid introduction of errors, to be tolerant with errors and to eliminate errors in the software system.

In terms of philosophy, the document is grounded on the assumption that software developments are framed by well-defined and verified processes which are structured following a standard V-lifecycle.

In terms of structure, the software development methodology is composed of four main parts. The first step of the methodology consists in a risk analysis to identify the criticality level related to the computer-based system that hosts the software system. This criticality level allows the categorization of the software system in the same manner as its targeted performance level. The second part focuses on the choice and qualification of COTS and software support tools as it is an important topic which is too often underestimated in many industrial sectors, with the notable exception of the avionics domain which benefits from a dedicated standard for software tool qualification (DO-330). The third-part deals with classic quality insurance aspects like project, configuration, change & documentation management. The fourth part relates to the development activities, from specification to installation, and their associated acceptance criteria and verification milestones. In addition to these four main parts, annexes describe the link between the risk classification approaches of other main safety-related standards and the criticality scale of the guidelines. They also propose an application of the guidelines objectives to model-based developments.

In order to assist software development and assessment teams, the document contains tagged key objectives along with their applicability for the targeted software performance level. The development and assessment processes of a given software system can be tailored to comply with a limited subset of objectives of the guidelines, depending on the performance level of the software system. Here is an example of one of these guidelines objectives:
OBJ_TOOLS_080
When all the outputs of a software support tool are checked by an activity of the development lifecycle, so that failures of the software support tool are detectable, the qualification is no more necessary; only proof of this complementary checking shall be justified and recorded.
[I, II, III, IV]

Here, the objective is required when developing SOFTWARE SYSTEM of performance level 1, 2, 3 or 4 (i.e. all the potential performance levels defined in the guidelines). Finally, in order to ease the software assessment work, the document offers an assessment checklist that lists in a table each guidelines objective with its applicability, and provides a “justification” field to state if it has been fulfilled or not by the development team.

Nevertheless, the main added value of the SOFTWARE DEVELOPMENT & ASSESSMENT GUIDELINES stands in the innovative positioning of code analysis. Formal techniques are proposed here as solutions for functional and non-functional verifications, including computation accuracy, parameter integrity, and behavioral conformance. While dynamic testing evaluates whether the software system is built correctly or not, thus levelling the confidence in its ability to behave correctly. Therefore, without minimizing the necessity to conduct verification activities to improve the confidence in the ability of the software development process to produce the right software system, the SOFTWARE DEVELOPMENT & ASSESSMENT GUIDELINES stress the part played by static analysis techniques at the software testing stage to enhance the demonstration of the behavioral correctness of the software system. This proposal can be linked to recent approaches that advocate moving from process-based towards more assurance-based certification process, especially in the aeronautics domain (Rushby, 2009) (Holloway, 2013).

The following two objectives taken from the guidelines illustrate this approach:

OBJ_DEV_230
Code analysis tools shall be used to check dead code and run-time errors.
[III, IV]

OBJ_DEV_270
The source code of each software unit shall be compliant with the the software unit detailed design documents. When he takes part in the design, the checker shall, unless non-applicable, define the checking acceptance criteria that relate to:

a) Appropriate event sequencing;
b) Data- and control-flow handling (e.g. testing of boundary conditions);
c) Functional dependencies;
d) On planned resource affectation;
e) Detection and management of loops;
f) Memory management and memory capacity overruns;
g) Exception handling.

The justification of non-applicable criteria shall be documented.
[IV]

2.2. Comparison with other standards

The guidelines are built following common functional safety standards, defining a rigorous development lifecycle and focusing on the use of tools to automate some tasks and improve some verification activities. Four key issues can be identified to compare the guidelines with the well-known IEC 61508 and DO-178C: criticality-dependent rigor, verification coverage, organization and verification of verification.

The first key issue addresses the efforts to be produced during the software system development based on its severity or criticality. This is a common point with IEC 61508 dealing with Safety Integrity Levels (SIL) and DO-178C with Design Assurance Levels (DAL), as the guidelines propose a criticality-dependent scale of development efforts depending on the expected performance level (from 1 to 4, 4 being the most critical). This performance level is an input when developing a software system; it has to be determined during the system analysis where all risks are analyzed.

The main difference in terms of software assessment methodology between the guidelines and safety standards like IEC 61508 or DO-178B resides in the verification coverage requirements. Indeed, while the guidelines particularly focus on functional coverage criteria for the verification of the produced code, IEC 61508 and DO 178B define additional structural coverage criteria usually achieved by means of dynamic testing activities (e.g. MC/DC, path coverage, etc.). This structural approach allows proving that the program possible behaviors that are not covered by functional testing are not dangerous and/or counterproductive in terms of performance. Nevertheless, this apparent scarcity of the guidelines is offset by the strong recommendations in terms of verification efforts on the source code by code analysis tools. This choice has been made because structural coverage can prove to be genuinely time-consuming, even if necessary. Unitary testing, although an important step in the validation of critical systems, offers only statistical verification aligned with the coverage rate of tests: indeed, it is very difficult to cover the entire range of variation of the inputs of a function, even adopting heavily codified development processes such as DO178.

The classical “software module testing” of the IEC 61508 has been called “checking of the software units” and it explicitly includes either the dynamic testing activity or the use of source code static analysis.

Regarding the development team organization and especially the independence required to carry some activities, the guidelines ask for independence between the “left” (descending) and the “right” (ascending) sides of the V-cycle and also for all verification activities. Nevertheless, these independence objectives do not freeze any team configuration (they can be achieved by peer-review in a same development team). This allows more flexibility for small development teams. The IEC 61508 and DO-178C are not really precise either on this specific topic; the only clear requirement is about the independence of verifiers and assessors.
While the functional safety standards IEC 61508 and DO178-C require tool qualification, the guidelines moreover recommend a usefulness analysis. It has to be done at the beginning of the software development, and it aims to evaluate pros and cons of performing manual or automated activities.

The last point deals with the verification of verification: as in IEC 61508 and DO-178C, all activities have to be verified and eventually assessed. In case of third-part assessment, a certificate of compliance may be issued regarding the guidelines.

3. Presentation of the CEA LIST static analysis tools on a use case

3.1. Presentation of the use case problematic

SIREHNA is a subsidiary of DCNS Group, and a part of the technological research center DCNS Research. SIREHNA features different fields of expertise such as hydrodynamics and control of mobile maritime units.

BUREAU VERITAS and CEA LIST aim to provide a formal framework defining verification objectives and milestones during software development. SIREHNA supports this project by providing its industrial vision both in terms of system features and operation, as well as process and industrial constraints for their development. The goal is to implement and apply the tools proposed through BUREAU VERITAS & CEA LIST initiative to ensure that they are relevant to industrial constraints and assess their performance in a concrete development context.

In this context the work done with BUREAU VERITAS and CEA LIST enables SIREHNA to anticipate new standards dedicated to the naval field.

3.2. Overview of Frama-C and Fluctuat

Two main tools have been used during this proof of concept: Frama-C (Kirchner, Kosmatov, Prevosto, Signoles, & Yakobowski, 2015) and Fluctuat (Delmas, Goubault, Potot, Souyris, Tekkal, & Védrine, 2009), which are both mainly developed at CEA LIST.

Frama-C is an Open-Source (LGPL-licensed) framework dedicated to the analysis of C programs. It is built around a kernel tasked with the parsing and type-checking of C code and accompanying ACSL annotations if any, as well as maintaining the state of the current analysis project. This includes in particular registering the status (valid or invalid) of all ACSL annotations, either user-defined or generated, emitted by the various analyzers. Analyses themselves are performed by various plugins, that can validate (or invalidate) annotations, but also emit hypotheses that may eventually be discharged by another plugin. This mechanism allows some form of collaboration between the various analyzers. Many plugins exist, but in the remainder of this section we focus only on the ones that have been used during the case study on the application of the proposed guidelines over Sirehna’s code.

While Frama-C’s kernel is meant to operate on C programs, the case study has fueled the development of a plugin for analyzing C++ code. This plugin, named frama-clang, whose prototype has been elaborated during the European FP7 project STANCE uses the clang compiler as a front-end for type-checking C++ code and converts clang’s abstract syntax tree (AST) into Frama-C’s own AST, producing C code equivalent to the original C++ program. In addition, frama-clang extends clang with ACSL++ annotations that are converted into ACSL annotation together with the translation of the code.

Two important analysis plugins are Value Analysis and WP. 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 then checks whether the abstraction obtained for the operand represents any value that would lead to a runtime error. For instance, when dereferencing a pointer, the corresponding abstract set of location should not include NULL. If this is the case, Value Analysis will emit an alarm, and attempt to reduce the abstract value. In our example, it will thus remove NULL. 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 done 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 bounds of variables at particular program points.

WP is a deductive verification-based plugin. Unlike 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, once they have been expressed as ACSL annotations. It is however also possible to use it to check that the pre-conditions written for a given function imply that no runtime error can occur during the execution of .

Finally, Fluctuat (which is not a Frama-C plugin but benefits from the Frama-C toolchain) focuses on the accuracy of floating-point computations. It is based on abstract interpretation and computes an over-approximation of the magnitude of the error due to rounding when performing a sequence of floating-point operations. Given a mathematical specification of what the operations

are supposed to compute, it can also indicate the magnitude of the error due to the method used (e.g. Newton algorithm for extracting a square root). The contribution of each individual floating operation to the overall rounding error can also be traced, in order to help designing more robust algorithms if needed.

3.3. Use case code analysis by Frama-C

From a methodological point of view, the aim of this code analysis activity is to prepare the certification. Formal proof on the code gives confidence in the library and ensures that the expressed properties are under control. The application of the development guide extends the current development process followed by SIREHNA to develop its applicative software. This process is based on the development of internal libraries that are likely to be shared, customized and then embedded in the applicative software. Current libraries come with unit tests with single input, validation unit tests with multiple inputs, non-regression tests. Current applicative software also comes with integration tests and non-regression tests. All these tests are good starting points for defining the verification scenarios that will drive the abstract interpretation based analyses. The first step just consists in replacing the input values of the tests by some ranges of values.

For the verification process of the numerical libraries, SIREHNA, CEA LIST and BUREAU VERITAS have decided to experiment a bottom-up verification approach. It benefits from the formal analysis tools WP and Fluctuat that are initially designed for the unitary verification of single components. At component level, some short verification cycles like annotation/analysis/result examination aim to deliver proved annotations in the source code. The annotations carry information about the domains, the loop invariants and the accuracy of the computations. The assembly of components in the library come with an assembly of the annotations guided by the user. The objective is for high level annotations to meet the properties exported by the functions of the library, as verified through whole program analysis.

Unitary analysis

Compared to unitary testing, the advantage of the static analysis tools proposed by CEA LIST is to prove, for instance, the absence of certain classes of errors, or some functional properties expressed as ACSL annotations and to complement unit and functional tests. This provides an increased level of robustness and greater efficiency in error detection.

The teams of the CEA LIST performed a code analysis of various "core" functions developed by SIREHNA in C++, using thus frama-clang as front-end. An initial analysis was carried out on the source code of the ship trajectory generation functions. In a first case study, Frama-C’s Value Analysis plugin was used to demonstrate its intervals values verification capabilities on floating point numbers. More precisely, the entry point for the analysis stemmed from an existing unitary test that was meant to check that the functions for converting Cartesian coordinates to polar ones and vice-versa were indeed inverse to each other (modulo rounding). For the analysis with Frama-C, we replaced floating-point inputs with small intervals around the original input, in order to check whether the function under test was robust to numerical imprecision. The entry point for Value Analysis was thus along the following lines:

```c
void test(void) {
    double x = Frama_C_interval(x0 - eps, x0 + eps);
    double y = Frama_C_interval(y0 - eps, y0 + eps);
    double z = Frama_C_interval(z0 - eps, z0 + eps);
    double x_conv, y_conv, z_conv;
    double lon, lat, height;
    cartesian2polar(x, y, z, &lon, &lat, &height);
    polar2cartesian(lon, lat, height, &x_conv, &y_conv, &z_conv);
}
```

Frama_C_interval is a built-in function of Value Analysis that returns any number between the bounds given as argument, while x0, y0, and z0 represent the original test input and eps the (small) interval of variation we want to examine. Value Analysis was complemented in this task by Fluctuat.

A second analysis focused on source code developed in C language intended to be incorporated on board submarines for the weight balancing check function. This time, Frama-C’s Slicing plugin was used as a front-end to let the Fluctuat tool concentrate on the analysis of the major contributors to numerical errors without undue interference with unrelated computations. Other plugins of interest for Fluctuat are frama-clang (C++ to C translator) and the constant propagation plugin (especially on the values of pointers). Finally, some preliminary experiments have been done to use WP for verifying accuracy of floating point computations. This work follows the methodology devised in an earlier collaboration with NASA (Goodloe, Muñoz, Kirchner, & Correnson, 2013).

Integration analysis

Value Analysis run at system level will then benefit from some invariants and some assertions put by the user and verified at component level. The certification/system level the objective of Value is to prove the absence of run-time errors (dangling pointers, divisions by zero and arguments of functions like asin outside the interval [-1,1]). Depending on the kind of library/system decomposition, a collaboration between the various tools(WP, Fluctuat, Value, etc.) may be needed to achieve a complete verification.

Case studies results

Following the CEA LIST presentations, SIREHNA evaluated the following Frama-C key functions:

- Visualization of the impact of floating point operations on numerical precision, which can identify the calculations that contribute the most to the numerical error.
- The generation of ACSL specification from an existing code, giving a synthetic vision of functions and allowing to apprehend unexpected side effects and trace dependencies between variables.
• Runtime error detection such as NaN, buffer overflow or uninitialized tables.

Strong added values have been foreseen in the following Frama-C features:
• Fully formalize the contract of simple functions directly on the source code when possible
• For more complex functions, formalize the limitations of application (preconditions)

These two features will improve the source code documentation (a step toward literate programming). As these assertions can be formally verified, this will have the following impacts on design process:
• Eliminate unit test and runtime assertions when the correctness of the implementation with respect to the contract can be proven
• Put the focus on potential safety issues that cannot be formally verified, and which will require more efforts (unit tests and/or runtime assertions)

For the application specific functions, it will be possible to specify the range of various quantities (like ship altitude, latitude and longitude). This is often a pre-requisite for Value Analysis and Fluctuat.

4. Conclusion

In this paper we present three new results: the elaboration of a set of guidelines for software development and assessment, guided by marine and naval considerations; the relationship between these guidelines and existing, state-of-the-art source code analysis tools; and the application of the tools to an industrial use case. These results are leading the current trend toward software product-based assessment projects, in order to be tolerant with possible accidents and optimize operational uptime.

The approach they advocate has shown potential benefits to manufacturers and end-users of the domain, and upcoming experiments will further investigate the impact of this approach on the overall software development and assessment processes.

The guidelines are freely available on the web site of BUREAU VERITAS (http://www.bureauveritas.com/white-papers/software-development-assessment).

Finally, in a global software certification program perspective, one important topic that has not been directly tackled by the partnership project yet is the rigorous study of the differences and complementarities of the two broadly-adopted software verification methods that are static and dynamic analyses. This would allow refining the balance of efforts to be put on these two software assessment approaches in order to further optimize the cost-effectiveness of certification programs.
References
Goodloe, A., Muñoz, C. A., Kirchner, F. K., & Correnson, L. (2013). Verification of Numerical Programs: From Real Numbers to Floating Point Numbers. NASA. 7871, pp. 441-446. LNCS.
Rushby, J. (2009). Software Verification and System Assurance. SEFM (pp. 3-10). IEEE.