SAFER MARINE AND OFFSHORE SOFTWARE WITH FORMAL-VERIFICATION-BASED GUIDELINES

Lucas Duboc, Sébastien Flanc, Florent Kirchner, Hélène Marteau, Virgile Prevosto, Franck Sadmi, Franck Védrine
{florent.kirchner, virgile.prevosto, franck.vedrine}@cea.fr
{lucas.duboc, helene.marteau, franck.sadmi}@fr.bureauveritas.com
sebastien.flanc@sirehna.com

Abstract:
As the development of ship software systems has followed the growth curve of digital technologies, Marine & Offshore assessors, like Bureau Veritas, are lacking dedicated software safety assessment standards and tools compared to other industrial sectors like railways or aeronautics. Indeed, in this field of Marine & Offshore, software systems are seen as *black-boxes*, i.e. only assessed through system testing without specific requirements for the software development. Given the impacts of software failures on human, strategic, economic, or environmental aspects, this approach is not sufficient. From this statement and because usual safety standards are very demanding in terms of development processes, Bureau Veritas has decided to issue pragmatic guidelines for the development and the assessment of industrial software. They are focusing on development processes and the use of efficient tools to verify software through a *white-box* approach. In this context Bureau Veritas has partnered with CEA-List which is a major actor in applied formal verification techniques. This paper is illustrated by a use case with Sirehna for the implementation of those guidelines on a critical ship software system.

Keywords:
Industrial software, Safety, Marine & Offshore, Software standards, Static analysis, Formal methods, White-box approach, Certificate of conformity

1. Context of the Partnership

Digital technologies have overwhelmed industrial markets, and software systems are underlying almost all technical products. 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. It is now well known and understood that software testing, because of its intricate logical nature, cannot detect all defects that might have been inserted in the software design and implementation phases. 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, in a fashion that is mindful of cost-effectiveness and maintainability.

1.1. Software verification in BUREAU VERITAS Marine certification schemes

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. For the industrial sectors where safety aspects are less prevalent, best practices of software development are widespread without being gathered in a single document. 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. BUREAU VERITAS is one of the 12 member companies of this association, and was entrusted with its chairmanship from July 2014 to July 2015.

So far, the standard requirements related to the assessment of ship software components were generic and limited. Unified Requirements (IACS, 2010) had been dealing with high-level requirements for software assessment, typically test completion 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. Intrinsic complexity of the software system has to be taken into account and processes of development have to be adapted accordingly.

In this context, BUREAU VERITAS Marine wants to improve its software assessment scheme (based on a white-box approach) to get more confidence in software systems used in the marine & offshore industry.

1.2. CEA LIST expertise in source code verification tools

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 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.).

Frama-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). Frama-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. 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 knowledge 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, buts 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.

This document was designed to be usable in all the domains that deal with the quality and safety of software systems. On the one hand, it had to be seen as guidelines of development and did not aim to supersede 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 was favored as long as their dangerous or complex instructions are 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 constraints and objectives.

The continuing partnership between BUREAU VERITAS and CEA LIST has allowed the team to successfully share knowledge and expertise, in particular on safety systems assessment and certification, and software verification.

2. Originality of the software assessment methodology

2.1. Guidelines rationale

Simplicity was a major objective during the writing of the SOFTWARE DEVELOPMENT & ASSESSMENT GUIDELINES. The authors strived to define a practical methodology that minimizes 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.

Even though it does not attempt at a thorough comparison between the main software safety-related standards that are enumerated in section 1, such as the work of (Ledinot, et al., 2014), the document follows the same philosophy and structure as these various standards. 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. 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_010
Each SOFTWARE SUPPORT TOOL used at any step of the SOFTWARE SYSTEM development shall
be identified.
[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 if the software system behaves as intended in particular in interaction with hardware and environment, static analysis assesses 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_185
Dead code and run-time errors shall be detected. Recursivity is accepted only if controlled by a maximum number of iterations.
[II, III, IV]

NOTE1: Depending of the programming languages, typical run-time errors are:

- Buffer overflows (underflow);
- Out of bound accesses (arrays, pointers, …) and null pointer dereferences;
- Invalid arithmetic operations (division by zero, square root of negative numbers,);
- Non initialized variables access;
- Dangerous type conversions;
- Shift in bitwise operations with a negative value or a value greater or equal than the size of the underlying type (C/C++); etc…

These kinds of errors are occurring during the execution of the SOFTWARE SYSTEM; even if they are syntactically correct.

OBJ_DEV_205
Code analysis tools shall be used to check dead code and run-time errors.
[III, 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 (61508, 2010): criticality-dependent rigor, verification coverage, organization and verification of verification.

The first key issue addresses the efforts to produce during software development that depend on its severity or criticality. This is a common point with the IEC 61508 dealing with SIL as the guidelines keep the idea of a scale of development efforts relating to the expected performance level from 1 to 4, 4 being the highest. 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.

Concerning the verification coverage, some differences exist between the two documents. When checking the detailed design, IEC 61508 defines specific structural coverages like MC/DC, path coverage, etc. whereas the guidelines only focus on functional coverage. This is offset by the strong verification efforts asked 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, so the focus has been placed on functional and source code analysis. Moreover, 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 and the right 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 is not really precise either on this specific topic; the only clear requirement is about the independence of assessors.

While the generic functional safety standard for electronic devices IEC 61508 requires 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.

Last point concerns the verification of verification, like in IEC 61508 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. Application of the methodology on a use case

3.1. SIREHNA 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.

Those skills are directly applied to maritime embedded systems such as Dynamic Positioning Systems; Frigates Rudder Roll stabilization systems, Unmanned surface Vehicles or critical tailored systems such as the French aircraft carrier flight deck tranquilization system. Those embedded systems include software components which perform many functions such as real-time computation of navigation data, multi-degree of freedom control algorithms, actuators orders and supervision of the system by human operators. These systems embedded on military ships, submarines, and offshore vessels operating near oil platforms, are safety critical and their complexity requires a rigorous development, verification and validation process.

Today, IACS does not impose code-level standards in the naval environment for the development of critical software applications. For example, the generic safety standard IEC 61508 is not specialized to the maritime domain.

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, Putot, 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[1] 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

[1] http://stance-project.eu/ and http://llvm.org/devmtg/2014-04/PDFs/Posters/FramaC.pdf

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 `f` imply that no runtime error can occur during the execution of `f`.

Two other plugins mentioned in the rest of the paper are not analyzers *per se*, but can help the main analysis plugins in performing some verification task on complex code. First, the Slicing plugin performs program transformations. More precisely, given a *slicing criterion*, e.g. the values of some variables at a given program point, Slicing will remove all instructions of the original program that do not contribute to this criterion. The result is a simpler program, which is equivalent to the original one with respect to the criterion. Slicing can thus be used as a front-end to other, more specific, analysis when focusing on a given property of interest. Second, the EACSL plugin transforms ACSL annotations into C code with `assert` in order to allow for dynamic runtime checks. It can for instance be used in complement of Value Analysis to evaluate during the execution of unit tests whether an alarm emitted by Value Analysis might occur in practice. Similarly, if a given ACSL annotation cannot be discharged by WP, it is still possible to check whether it holds during concrete executions thanks to EACSL. Note however that EACSL only supports a subset of ACSL, and that the instrumentation it performs might have an impact on the execution time of the program.

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. Sirehna's 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

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. Therefore, 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:

```
void test(void) {
    double x = Frama_C_interval(x_0 - eps, x_0 + eps);
    double y = Frama_C_interval(y_0 - eps, y_0 + eps);
    double z = Frama_C_interval(z_0 - eps, z_0 + 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 `x_0`, `y_0`, and `z_0` 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. At 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]`).

Let us consider the following piece of code as a short example:,

```
/* basic low level function of the library ; LIBRARY-level */
double distance(double x, double y)
  { return sqrt(x*x + y*y); }

/* intermediate function of the library ; LIBRARY-level */
double angle(double x, double y)
  { double dist = distance(x,y);
    if (dist > MINIMAL_DIST) {
      double result;
      if (y > dist*EPSILON || y < -dist*EPSILON)
        result = atan(x/y);
      else
        result = asin(y/dist);
      if (x < 0)
        { if (y < 0) result -= PI; else result += PI; }
      return result;
    };
```

```
    return 0.0;
  }

/* elaborated high level function of the library ; LIBRARY-level */
void cartesian2polar(double x, double y, double* rho, double* theta) {
  *rho = distance(x, y);
  *theta = angle(x, y);
}

/* applicative software ; SYSTEM-level */
int main() {
  double x, y, rho, theta;
  ...
  while (...) {
    /* domain information for the applicative software */
    x = Frama_C_interval(-10.0, +10.0);
    y = Frama_C_interval(-10.0, +10.0);
    cartesian2polar(x, y, &rho, &theta);
  }
  ...
}
```

One problem faced by the certification process on such kind of library/system decomposition is that it may be non-conclusive. On the one hand, WP/Fluctuat cannot prove some properties like the result accuracy at the library level since it depends on the domains only provided at system level. On the other hand, Value cannot prove at system level that `y/dist` as the `asin` argument is in the interval `[-1, 1]` due to its internal logic targeted to be efficient on Run-Time Errors but not precise enough to know that `abs(y) <= sqrt(x*x+y*y)` without any domain information. Thus, a collaboration between the various tools is needed to achieve a complete verification. More precisely, the following verification cycle can be proposed.

1. Annotate the library with ACSL contracts and use deductive verification – WP tool, with the Gappa (de Dinechin, Quirin Lauter, & Melquiond, 2011) theorem prover that has a built-in understanding of floating-point operations.
2. Define unitary scenarios (possibly based on the existing unit tests).
3. Use Abstract Interpretation at component level with the Fluctuat tool.
4. Adjust coefficients in annotation formulas following the results of the unitary analysis.
5. Assemble the various components
6. Use Abstract Interpretation at system level with the Value Analysis plugin of Frama-C and checks that the functions of the libraries are called with arguments in the appropriate bounds.
7. If needed, use the EACSL plugin with validation unit tests and/or with integration tests for the remaining alarms in order to classify them as true or maybe false alarms.

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.

SIREHNA foresees strong added value 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.

The outcome of this work demonstrates significant interest in productivity and securing the development process. SIREHNA has started a reflection aimed at integrating Frama-C in the development processes of its system's critical functions. Besides the extension of Frama-C to C++ is followed with great interest.

3.4. Review and application of the guidelines

SIREHNA conducted a review of the development guide "SOFTWARE DEVELOPMENT & ASSESSMENT GUIDELINES."

This document is characterized by its pragmatic approach:

1. Recommendations are concise and written in the form of objectives.
2. A global approach is described and addresses the complete development cycle, project management, use of COTS, development tools.
3. The definition of software categories in relation to their criticality is declined to different processes within one single document.

In addition, the document recalls the various existing standards on the classification of risks and software. This approach by category is directly transposable in SIREHNA processes for software development with different levels of criticality.

BUREAU VERITAS, CEA LIST and SIREHNA worked on an actual project to link the recommendations contained in the guide with the development process followed on this project (the product line of Dynamic Positioning systems). The analysis generated feedback on the application of the software development guidelines and confirmed the applicability in real life development process.
This analysis has been done during the year 2015. SIREHNA has delivered the whole documentation of a Dynamic positioning system to BUREAU VERITAS. Considering that SIREHNA has been developing software for a long time, their process of development is matured enough to assess it. BUREAU VERITAS worked as a third party assessor filling the objectives matrix based on the SIREHNA documentation.

The first activity consisted in selecting the Software Category (performance level or SIL) of the software. In the IACS requirements, the systems whose failures could immediately lead to dangerous situations are Category 3 (highest category). In the annex of the guidelines, a correspondence is given between IACS Categories & Guidelines Software Categories. Thus, the Dynamic Positioning system has been rated Software Category 3 (Software Category 4 being the highest class in the guidelines). All the objectives of the guidelines have been browsed knowing that the software system needs to reach Software Category 3.
The main conclusion of this assessment is that the SIREHNA development process is compliant with the guidelines.
Nevertheless the guidelines objectives recommend to qualify more formally tools & COTS. The functional coverage of the Software System is achieved by a strong set of validation test cases (including nominal & robustness cases). Some tests cases were partially modified to check the behaviour of the software out of his bounds, especially because the DP operates depending on numerical computations with a certain accuracy acceptance.

This analysis generated feedback on the application of the software development guidelines (some objectives have been re-worded or added) and confirmed its applicability in real life development process.
The analysis will be carried on during 2016 to assess a new version of the DP. This will permit to deliver a certificate of compliance on the DP software.

4. Conclusion and Perspectives

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 both guidelines and tools to an industrial use case. These results are leading the current trend toward software validation in the marine and offshore industry to deal 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 process. Finally, while they were developed with a specific application field in mind, the guidelines presented here are generic enough that they could be applied to numerous other industrial fields where safety is a quickly emerging concern. The guidelines are freely available on the web site of BUREAU VERITAS (http://www.bureauveritas.com/home/about-us/our-business/industry-offer/software).

5. References

61508, I. (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems.*

Antoine, C., Trotin, A., Baudin, P., Collard, J., & Raguideau, J. (1994). CAVEAT: a Formal Proof Tool to Validate Software. *Convention on Nuclear Safety.*

Baudin, P. B., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., & Prevosto, V. (2015). *ACSL: ANSI/ISO C Specification Language version 1.9.*

Beckert, B., Hähnle, R., & Schmitt, P. (2007). *Verification of Object-Oriented Software: The KeY Approach.*

Canet, G., Cuoq, P., & Monate, B. (2009). A Value Analysis for C Programs. *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation.*

Cousot, P., & all. (2011). *The Astree Static Analyser*. (J. F. R. Cousot, Producteur) Récupéré sur http://www.astree.ens.fr

de Dinechin, F., Quirin Lauter, C., & Melquiond, G. (2011). Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Computers 60(2)*, (pp. 242-253).

Delmas, D., Goubault, É., Putot, S., Souyris, J., Tekkal, K., & Védrine, F. (2009). Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. *FMICS. 5825*, pp. 53-69. LNCS.

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.

Hoare, C. (1969, October). An axiomatic basis for Computer Programming. *n axiomatic basis for Computer Programming, 12*(10), 576-583.

Holloway, M. (2013). Making the Implicit Explicit: Towards An Assurance Case for DO-178C. *ISSC.* Boston.

IACS. (2010). *E22 : On Board Use and Application of Programmable Electronic Systems.*

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., & Yakobowski, B. (2015). Frama-C, a Software Analysis Perspective. *Formal Aspects of Computing, 27*(3), 573-609.

Ledinot, E., Blanquart, J.-P., Astruc, J.-M., Baufreton, P., Boulanger, P., Comar, C., et al. (2014). Joint use of static and dynamic software verification techniques: a cross-domain view in safety critical system industries. *ERTS*, (pp. 592-601).

Marché, C., & Filliâtre, J.-C. (2007). The Why/Krakatoa/Caduceus Platform for Deductive Program. *19th International Conference on Computer Aided Verification.*

Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., & Schoen, D. (1999). Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. *FM. 1709*, pp. 1798-1815. Springer.

Régis-Gianas, Y., & Pottier, F. (2008). A Hoare logic for call-by-value functional programs. *Ninth International Conference on Mathematics of Program Construction.*

Reynolds, J. (2002). Separation Logic: A Logic for Shared Mutable Data Strutures. *17th IEEE Symposium on Logic in Computer Science.*

Rushby, J. (2009). Software Verification and System Assurance. *SEFM* (pp. 3-10). IEEE.

Sotin, P., Jeannet, B., & Rival, X. (2010). Concrete Memory Models for Shape Analysis. *International Workshop on Numerical And Symbolic Abstract Domains.*