Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach

Famantanantsoa Randimbivololona[1] and Jean Souyris[1]
Patrick Baudin[2], Anne Pacalet[2], Jacques Raguideau[2], Dominique Schoen[2]

[1]Aérospatiale Matra Airbus, M8621 - 316, route de Bayonne - 31060 Toulouse cedex, France
[2]CEA Saclay, LETI-DEIN, 91191 Gif-sur-Yvette cedex, France

Abstract. This paper reports an industrial experiment of formal proof techniques applied to avionics software. This application became possible by using Caveat, a tool dedicated to assistance in comprehension and formal verification of safety critical applications written in C. With this approach it is possible to reduce significantly the actual verification effort (based on test) in achieving the verification objectives defined by the DO 178B [4].

1. Introduction

1.1. Purpose

The aim of this paper is to report an industrial experiment of formal verification.

The avionics software verification process must meet the DO 178B [4] requirements. Due to the increasing software size and the hardware technology evolution traditional verification techniques, i.e. the tests, tend to be less and less cost effective in meeting the DO 178B [4] verification requirements.

An alternative and complementary approach consists in using static verification techniques and particularly *formal proof of property*.

The work reported in this paper is a part of a project aiming at the introduction of formal proof in the operational avionics software development. This introduction must be effective in 2001.

All the experiments, and specially the case studies of this paper, have been made by embedded software developers, sometimes with the assistance of formal proof specialists.

1.2. Context

The corresponding applications belong to on-board computers involved in electrical flight control, air/ground communication, alarm and maintenance systems. All these applications have safety, reliability and availability requirements. The consequences of these requirements on the software affect the architecture, fault detection capabilities (functional verifications, asserts, hardware monitoring), recovery from fault detection, etc...

The essential part of the verification cost is due to these features since they require dedicated means.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1798-1815, 1999.

1.3. Proof of Property

Property -or program- proof is a well known technique, based on Hoare's [1] or Dijkstra's [2] theories. An interesting characteristic of these theories is that they can be computer aided, i.e. a *tool can be developed to help prove properties*.
In order to meet the objectives defined in section 1.1 the requirements for such a *tool* are listed below.

Ability to prove avionics C code. This is the strongest requirement because formal verification is dedicated to real software products.

Ease of learning and use. The main point, here, is the ability of the tool to be used by "standard" software developers, not only by a team of formal proof specialists.

Early payback. Tool aided formal proof must be used in replacement (not in addition) of the most tedious and expensive phases of the testing process.

Easy integration. The use of the tool should not break down the actual verification process and environment.

A tool which meets this requirement is Caveat, developed by the French Commissariat à l'énergie atomique (CEA). This tool -evaluated by Aerospatiale during the European project LAW [3] - is a "verification assistant" able to perform proof of property.

1.4. Avionics Software Characteristics

Functions. The different classes of functions of an avionics software product are numerical computation, hardware handling, communication protocols, security/protection mechanisms, fault-detection and recovery, Boolean computation.

Properties. An avionics software must have the following types of property : functional, safety, robustness and temporal.

Architecture and sizes. The design and coding rules of an avionics software lead to a modular architecture. They also limit the size and complexity of the individual modules.
The size of an entire avionics software product may be up to 500,000 lines of code.

Algorithms. From that point of view, avionics software is never very complicated. For instance, the loops are very simple (eg : array initialisation, search within an array). So one of the great difficulties of automatic property proof, i.e the analysis of loops, is simplified a lot.

1.5. Development Process of Aerospatiale Matra Airbus Avionics Software

This section gives an overview of the *actual* avionics software development process. It is a typical "V" process.

We will see in section 3.1 how we intend to introduce formal proof in this process.

Specification. There are two families of specifications : *Formal specification*s using the following specification languages : SAO, SCADE, LDS and *textual specifications*, written in natural language.

The formal specifications are most of the time automatically coded.

Design. There is no design activity for the automatically coded pieces of code (from formal specification).

In the case of textual specifications, the design process is based on the HOOD [5] method.

Coding. As stated earlier, the code can be produced automatically when associated with formal specifications (SAO, SCADE, etc) or "intellectually" produced from the HOOD [5] design when the specification is in textual format. Several languages are actually used, e.g. assembly languages, Intel PL/M, C, etc. *Only the C language is considered in this paper.*

Verification process. With the exception of reviews and analyses, all the verifications are performed by *tests* ; the basic principle of the test being the notion of *execution*.

There are three sets of tests : *Unit tests* whose objective is to prove that each module meets its requirements ; *Integration tests* are performed to prove - progressively - that the modules interact correctly, on a host system first and then on the target hardware ; finally, *Validation tests* performed in an environment whose characteristics are very close to the aircraft finally prove that the software meets its customer's requirements.

The typical software properties to be proven during these three verification phases are : functional and safety sequential properties in Unit testing and in Integration testing on the host platform ; real-time and hardware access properties in Integration testing on the final target ; functional, safety and real-time properties in Validation testing.

These properties are not treated as such, they lead to test case generation and test execution.

2. Caveat

Caveat is a tool based on static analysis of source code, for comprehensive analysis and formal verification; it is dedicated to safety critical applications written in C.

Some technical aspects and formalisms used in Caveat and some industrial constraints taken into account in the design of the tool are described below.

2.1. Technical Aspects and Formalisms

The tool is based on the following well-known techniques.

Static analysis of source code. Tables and internal trees coming from compilation, are used to perform detection of anomalies, synthesis of properties and proofs.

A dedicated property language. Based on conventions coming from on Z[9] and VDM[10], it allows properties of the first order logic to be expressed (generated or to be proved).

Models for pointers and arrays are defined to describe such entities in the predicate language; specific operators are added to facilitate the writing of predicates dealing with structures, arrays or pointers of the C language.

Features are added to the property language in order to describe different kinds of property : explicit and implicit operands of a function, class of operands (In , Out, Inout), dependencies of outputs on inputs (From), postcondition (property that must be satisfied at the end of a function : Post), precondition (property that is assumed to be satisfied when the function is called : Pre), local-condition (property that must be verified at a specific location inside a function : Assert).

Some of these properties are automatically computed by the tool during a property synthesis phase. An example of the generated properties is given on figure 1.

Weakest precondition computation. The technique described by Hoare [1] is used to compute the condition that must be satisfied by the inputs of the function to ensure that the given property will be satisfied after execution of the code, if it terminates.

The semantics of each instruction is taken into account, modifying the initial predicate towards the weakest precondition.

An algebraic simplifier. It is used during the weakest precondition computation, to reduce, as soon as possible, the size of the formulae and during the demonstration phase.

It is based on a set of about 2000 re-writing rules of the following shape :

left-term [left-proc] --> right-term [right-proc]

where left-proc and right-proc are optional C procedures that may help describe the re-writing.

The tool looks for the matching rules in the initial predicate, computes the substitutions and applies them to obtain the simplified expression.

The simplification strategy of a term proceeds recursively to its sub-terms.

The rules deal with associativity, commutativity, distributivity, equalities, inequalities, arithmetical and boolean operations, numerical constants, and specific notations.

An automatic theorem prover. It is fully integrated ito the tool : it takes, as an input, the predicate coming from the weakest precondition computation (the goal) and the result of the demonstration is expressed in the property language.

The aim of the theorem prover is to demonstrate the goal under some hypotheses. There are two kinds of hypotheses :

. axioms specific to areas and independent from application,

. specific hypotheses in relation with the application : preconditions of the function, postconditions of called functions.

The demonstration is performed by generation of sub-goals using some inference rules. The choice of a rule depends on the syntactic structure of the initial goal. When no decomposition into sub-goals is possible, other inference rules using hypotheses are applied. The algebraic simplifier is also called.

Of course, being fully automatic (as opposed to of the Larch Prover [8]), it may fail. The wish is to avoid asking for assistance from the user for the choice of such or such a strategy during the demonstration phase, because it would suppose a specific skill in that domain.

In case of failure, the result returned by the tool is the remaining part of the initial goal that is not proved. Graphical facilities are provided to understand the structure of the result.

Example and counter-example generation. In case of failure of the demonstration, it is important for the user to know whether the failure comes from the tool or from a real error in the code. The tool offers other possibilities, such as the generation of examples or counter-examples for debugging purposes.

When the user asks the tool to generate counter-examples, the tool computes a predicate giving conditions on inputs which refutes the property. These conditions, after constraint solving, give input values with which the function may be executed to exhibit the problem.

An interactive predicate transformer. Another possibility in case of failure is the re-writing of the remaining predicate resulting from the demonstration. The interactive predicate transfomer offers a list of possibilities to rewrite the remaining formula, in order to facilitate reading, understanding, simplification and even demonstration: the user may introduce "let" notations to reduce the size, break the predicate into cases, rewrite it in a normal disjunctive or conjunctive form, etc. For instance, the disjunctive normal form allows independant proofs to be performed on

each member of the disjunction; if only one of the members is proved, the initial property is proved.

Processed Language. Unlike other similar products (see 2.x), caveat performes proofs on the source code, not in an intermediate code, because it is the lowest representation of the executable code easily understandable by the user, on which reasoning can be performed.

The chosen processed language is ANSI C, as defined in the ISO standard, because of its widespread use in industry, but implementation makes it possible to deal with other programming languages without restarting from scratch.

The current version of the tool has some restrictions in the use of ANSI C : features like function pointers, recursive calls, alias are not implemented yet.

2.2. Industrial Constraints

The industrial constraints of the targeted applications match Caveat capabilities.

Suitability for processed language. The features of C language not addressed by Caveat correspond to limitations of critical application coding rules in the aeronautical and nuclear fields.

Size of application. Two other specificities give Caveat the ability to address industrial applications : *iterativity* and *interactivity*. They both make it possible to capture the right level of information from the user and give him back just the necessary information to go further, avoiding getting him bogged down in details due to the size of the application.

Verification work may be performed step by step, in an iterative process: the user asks the tool to verify a property on a function without being obliged to describe anything previously : as an answer, the tool exhibits just the missing information (if any) for performing the proof. Interactivity allows the user to give the missing information and go further in his verification process.

Facility of use. Iterativity and Interactivity are supported by the interface of the tool. It is composed of three main windows (see figure 1) : on the left, the user can read the C source he is working on; the right window displays the C function properties ; the window below shows the result of the last proof done by the tool (if the proof is not established). With a simple selection, it is possible to see the connections between elements of the property or result windows and the source window.

A property file is associated with each C source module. Each function in the C module has its dedicated block of properties. Some of these properties are automatically generated during the initial analysis of sources (prototype of functions, Implicit operands, From, classes of operands In, Out, InOut). During this property

synthesis, anomalies (if any) like uninitialized variables, dead branches are pointed out; the call graph of the application is computed and may be displayed. The result of this initial automatical analysis of sources is displayed in the property window.

The user may then use interactive facilities provided by the interface to add his own properties, either for getting proof, example, counter-example (post, assert), or for detailing the context in which the study is performed (pre), or for analysing the result of failed proof (interactive predicate transformer).

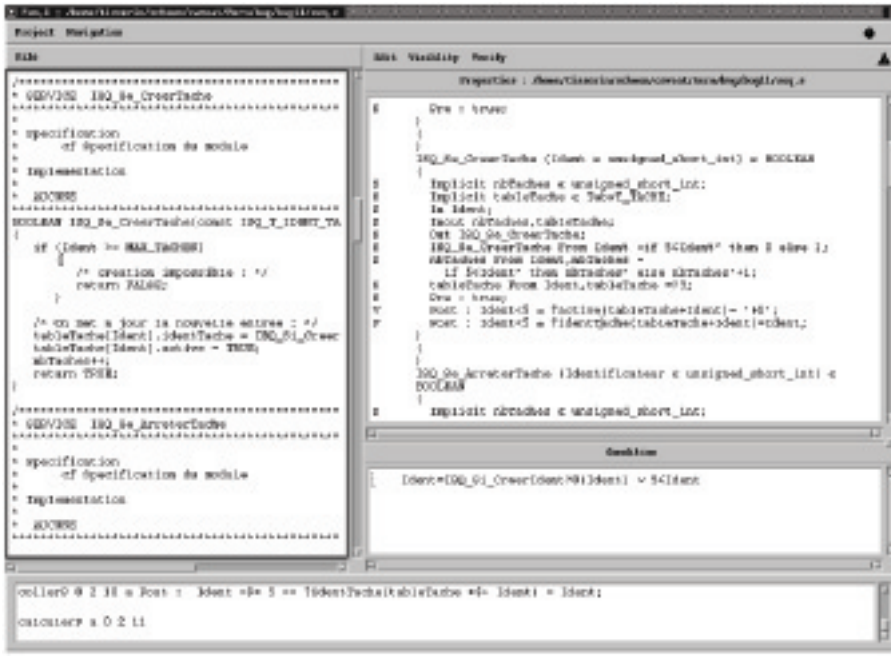

Figure 1.

Figure 1 illustrates the capability of the tool to peerform proofs even if pieces of information are not provided : for instance, a loop invariant is not necessary if the loop does not modify the variables of the property. Figure 1 shows that P1 is satisfied without providing any information about the C function ISQ_SiCreerIdent(). On the other hand, the post property P2 fails : the tool indicates that information on the output of the ISQ_SiCreerIdent() function is missing to perform the proof : this is an example of the iterative work : the user is invited to add this information (and only this) and may remain concentrated on his initial work.

Re-use of previous work. Many applications use identical software components. Validation work performed on these components may be re-used by means of user's libraries in which already demonstrated properties are memorized. This feature is part of the iterative aspect of the proof process defined as a basis of the Caveat tool.

Batch mode. The tool provides the possibility to record commands, and replay them in batch mode. This is very useful when interactivity is no longer needed, for instance for non-regression verification or when results on the verification process have to be given to a licensing authority.

Tracking of dependencies. The tool performs proofs by using the properties of low level components: axioms, semantics of operators, properties of the called functions... The tool manages the links between properties. It is then possible to know the consequences of the modification of low level properties on upper levels, and thus evaluate the work that must be repeated.

2.3. Comparison with Other Tools

A verification tool. Caveat is merely a verification tool. It does not claim to cover the whole software development life-cycle. Is is clearly dedicated to the last development phases, the programmation and verification phases in which it aims to help the user to analyse the code, understand it better and verify it regarding some properties.

Caveat is not a verification system like GVE [7] with a specification language and translators into programming language.

No intermediate language. Unlike tools such as MALPAS[6], *Caveat works directly on the source code*. This provides the advantage of eliminating any translation from source code into an intermediate langage, so that the model on which proofs are performed is as close as possible to the code. Consequently, the results are also as close as possible to the initial source, and are thus easier to analyse.

Interactivity. Caveat seems *much more interactive* than other similar tools. The preference is given to expressivity of properties and readability of results rather than automaticity.

3. Using Caveat on Avionics Software

The main objective of this chapter is to show how the methodolgy being built around Caveat allows the claims made in sections 1.1, i.e. cost reduction, and in section 1.3 to be met.

Three significant examples illustrate the way we will use Caveat and the associated methodology.

3.1. Methodological Approach

In section 1.3, we listed the requirements for a tool (and related methodology) able to perform formal proof.

The following requirements are directly met by Caveat (See chapter 2 for the substanciation of these claims) : *ability to prove avionics C code*, *ease of learning and use*.

In order to meet the rest of the claims, i.e. cost reduction, early payback and easy integration), a methodology is being developed.

After an analysis of existing avionics software, we defined the following courses for the application of formal proof using Caveat : *algorithm verification* at unit (module) level and *safety* analysis. Both approaches are related to properties which can be expressed in the first order logic.

Algorithm verification at module level takes place in the current unit testing phase. The objective here is to replace this main activity of unit test by formal proof. If achieved, this objective will lead to *spending significantly less time* in verifying a property, *using less expensive specific hardware and related software* (no execution with formal proof) and, finally, *detecting more problems earlier* in the verification process.

The first claim ("spend significantly less time...") is the consequence of the ease of use of Caveat. With this tool, to prove that a C function has a certain property, the user "only" has to express the property in the first order logic language of Caveat, run the tool and analyse the result. To do the same, i.e. prove that a C function has the required property, using unit testing techniques, one has to generate test cases, code a test program, debug it if necessary, execute both test and tested programs linked together and finally analyse the result. This comparison of these techniques allows us to think that formal proof used in module verification is more cost effective than unit testing.

The second claim ("use less expensive hardware") is due to the fact that, currently, the execution of unit tests is performed on a target able to execute the same binary code as the final embedded target (the on-board computer). So, in order to execute unit tests generated with a dedicated commercial tool, we need a run-time per execution hardware type. With formal proof using Caveat, there is no need for test hardware and run-time dedicated to unit verification.

The third claim ("detect problems earlier...") is due to the exhaustivity of formal proof versus tests.

Verification of dependability properties. This is the second application course of the proof of property technique. Critical software have to meet dependability objectives, derived from safety analyses performed at system and equipment levels. Software safety properties concern higher levels of software functions and usually they involve several modules. In this context, Caveat interactivity helps in building a property step by step.

Consequences on the requirements. Using the formal proof technique on software whose specification is textual (not formal) will *improve the quality of the requirements*. They will tend to be more accurate because a lot of them will have to be translated into first order formulae for verification. So, the formulation of these requirements will have to be semantically close to first order logic formulae in order to facilitate the translation into the property language of Caveat. The textual specification will remain in natural language but with a slightly more *logic* form.

3.2. Case Studies

The examples of this section illustrate the use of Caveat on critical embedded software applications and give some methodological indications about the way to use the proof of properties in this industrial context.

They are also representative of the kind of functions and code found in avionics critical embedded software. They reveal the ability of the tool to cope with these sorts of code.

Different types of function. The three examples are functions frequently met in avionics software. The first one belongs to the logic computation family, which can be met in on-board warning or maintenance on-board computers. The second example is representative of the security mechanisms met in several computers into which complex structured data (in a file, for instance) can be entered. The third one represents the hardware interface. Because on-board computer hardware is completely specific ("home made"), avionics software developers have to write hardware interface layers. These layers allow hardware initialisation, input/ouput handling, hardware monitoring, etc.

Coding characteristics. The first example, Boolean computation, shows what we can call "strongly Boolean code" : a lot of boolean operators (AND, OR...), RS flip-flop, confirms, etc. The specificity of the second one is the fact that it involves code running on top of an embedded UNIX-like operating system. There are wo consequences : firstly, the C modules include a lot of system headers defining plenty of data types, constants, etc... ; secondly, the C modules call system or library functions. As explained above, the third example is representative of the pieces of code with inputs coming directly from the hardware. The particularity, here, is the handling of bits present in data words read from hardware registers. The problem is the same when bits are used to compact information into a few data words (in operating systems for instance).

Properties. The examples described below are also representative of different kinds of required properties. For logic computation (first example) it is important to prove that some erroneous outputs or combinations of ouputs which can lead to dangerous behaviours will not be generated. So, the properties for this kind of function are *safety properties*. For the second example, the required properties are clearly *security properties*. But as security is the main function of the family represented by the second example, the properties it requires are also *functional properties*. The properties required by the third example (hardware handling) are *safety properties* because the objective, in that case, is to prove that a logic computation, based on the value of hardware registers (bit-structured), is safe.

3.3.

Making the proof of a property requires the following steps (they are applied to the examples) :

(1) Identification of the property in the requirements.

(2) Identification, in the design, of the code modules (C modules) involved in the property.
(3) Identification, in the design, of the C function on which the property must be verified.
(4) Creation of the Caveat project, i.e running Caveat on the "project" made of the analysed modules.
(5) Translation of the property from natural language (the way it is written in the requirements) to the property language of Caveat (based on the first order logic). The property is then given to the tool at the appropriate point of the function identified in step 3.
(6) Proof of the property by Caveat : Caveat computes the condition (in the first order logic) which has to be TRUE at the beginning of the C function, in order to be sure that the property is verified at the point of the function at which it was introduced.
(7) Analysis of the result. Two cases :
Caveat says "V" ("Verified") : the remaining verification condition (See point 6) is TRUE thus the property is verified by the code,
Caveat says "F" ("Failed") and gives the remaining condition. In this case, there are two potential reasons why Caveat failed :
- The property is not verified by the code. In this case, it is possible to deduce counter-examples from the remaining condition given by the tool. This can be done with a constraint solver.
- The property is in fact verified by the code but the tool cannot prove it, i.e. the remaining formula cannot be "simplified" to TRUE. In this case the proof has to be completed "intellectually", aided by the Caveat Interactive Predicate Transformer (IPT). Examples for tests can also be deduced from the remaining verification condition.

3.4. Examples

Safety : Boolean computation.

This case comes from the requirements (in the avionics specification language called SAO) of an on-board computer which performs a lot of logic computation. The analysed SAO sheet produces ten Boolean outputs (out01...out0a) from 32 inputs (In01..In20). It is composed of about 70 logic symbols like Boolean operators (AND, OR) or flip-flop, confirms, etc.

This example is typical of the *dependability verification* course (see section 3.1).

Step 1 : An interesting safety property is : "At any time, at least one output must be set".

Step 2, 3 and 4 have been performed. For the rest of the description, it is not important to know the related names of the modules, functions or tool files.

Step 5 : The ten outputs are declared as an array of unsigned char (out[10]) and only two values can be given to its elements : 0 and 1. An output (eg : out[2]) is "set" when its value is 1. In the property language of Caveat, we obtain :

$$\text{Post :}$$
$$\uparrow(\text{out+0}) \neq 0 \vee \uparrow(\text{out+1}) \neq 0 \vee \uparrow(\text{out+2}) \neq 0 \vee \uparrow(\text{out+3}) \neq 0$$
$$\vee \uparrow(\text{out+4}) \neq 0 \vee \uparrow(\text{out+5}) \neq 0 \vee \uparrow(\text{out+6}) \neq 0 \vee \uparrow(\text{out+7}) \neq 0$$
$$(1) \vee \uparrow(\text{out+8}) \neq 0 \vee \uparrow(\text{out+9}) \neq 0 \ ;$$

Note. $\uparrow$(out+0) in the Caveat property language is equivalent to out[0] in C.

Step 6 : The computation results in "V" ("Verified"). It means that the property is always true, i.e. for all possible input values.

Considering the great number of inputs of the module (32), it is clear that the amount of tests that would be necessary to prove the property leads to a far greater effort than the one needed to perform the mathematical proof.

Security : Uploading checks.

This case study was extracted from another on-board computer. The origin of the analysed piece of code is the uploading facility of this on-board computer. This feature allows avionics applications to be uploaded into the computer, from a floppy disk.
On this floppy disk, there are two sorts of file : a configuration file containing the relevant characteristics of the files to be loaded and the files themselves.

The main actions of the uploading function are : *loading* the configuration file from the media (using an on-board communication protocol) ; *analysing* the contents of the configuration file in order to identify which files are to be loaded and to verify if their associated characteristics allow them to be loaded ; *loading* the files ; *updating* the internal configuration (internal configuration files).

The case presented here involves the module implementing some of the checks before loading. As these checks are performed by a unique C function, this example illustrates both methodological courses (see sect. 3.1) : algorithm (at module level) and dependability verification.

Step 1 : In the case of software security checks, the interesting properties are "mapped" on the requirements. Checks are there for security, so verifying their code with the "safety point of view" is identical to verifying their function (the requirements).
For this example, twenty properties have been identified. Nineteen of them express reasons for stoping loading and the twentieth says : "If all the checks are OK, the loading can carry on".

Two examples of properties :

(P1) "If the name of the file to be loaded exceeds 49 characters then stop loading"

(P2) "If the application to be loaded is an ABC one and the priority it requires for its execution is not between 105 and 200 then stop loading".

Step 2, 3 and 4 : same as previous example.

Step 5 : The characteristics of the file to be loaded are read from the configuration file and loaded in memory via a complex data structure. The checked characteristics are fields of this complex data structure, called TFC_Ri_FichierConfig in source file.

The translation of both properties into the property language (see step 1) gives the following post-conditions :

Property P1 :

Post P1 :
strlen?0(↑InfoComune(TFC_Ri_FichierConfig.Log+num_elem).NomDest)
≥3335 0
(1)⇒ TFC_Si_VerifElement=0;

Let us comment on this formula :

♦ ″TFC_Si_VerifElement″ is the name of the analysed function and "⇒ TFC_Si_VerifElement=0" means "implies the value returned by TFC_Si_VerifElement is 0".

♦ *"strlen?0" in P1* stands for the return of the function strlen(). This function is called by the analysed function.

♦ "↑*InfoCommune(TFC_Ri_FichierConfig.Log+num_elem).NomDest*" *in P1* represents "TFC_Ri_FichierConfig.Log[num_elem].InfoCommune.NomDest" in C language.

Property P2 :

Post P2 :
((↑TypeLog(TFC_Ri_FichierConfig.Log+num_elem)=0
∧ (↑InfoCommune(TFC_Ri_FichierConfig.Log+num_elem).Priorite<105
∨
↑InfoCommune(TFC_Ri_FichierConfig.Log+num_elem).Priorite>2
00))
(2)⇒ TFC_Si_VerifElement=0;

Step 6 : the computations of P1 and P2 but also the computations of the eighteen other properties result in "V" ("Verified"). This clearly means that the analysed function meets its requirements.

Hardware handling. The case study presented here is symptomatic of a logic based on bit-structured data. In this example, such data come from an hardware register (in fact a CPU board status register). The function analysed in this case must convert each configuration of a subset of the register bits into pre-defined integer values. The computed value must be stored in a global resource because the value computed at a given time must be greater than or equal to the one previously stored. *Remark* : the global resource is shared with another piece of code which can reset it. This comparison mechanism is used to set priorities among the actions associated with the computed values.

Applying the method (see section 3.2) to this example, we get :

Step 1. The following property is considered : "If the stored value is the smallest (among the possible ones) and the currently computed value (from the current hardware register value) is not the smallest then the result cannot be the smallest".

Step 2, 3 and 4. Same principles as in previous examples.

Step 5. *Hypotheses for the property* : the current version of the tool does not automatically recognize the values of *const* data. So, because the analysed C function uses a const array to compute the integer value from the register data, the proof can be performed only if we give Caveat the const array values. This is done using a *pre-condition*.

Pre-condition giving Caveat the const array values :

Pre :
IIS_Ri_TabCoupure=Mk&Tab&DHT_T_TYPE_COUPURE
(4)(@IIS_Ri_TabCoupure, 16, 0,{0→17, 1→0, 2→33, 3→66, 4→0, 5→0,
6→0, 7→0, 8→0, 9→0,10→33, 11→66, 12→0, 13→0, 14→0, 15→66},);

Remark. With this property, Caveat considers that the array IIS_Ri_TabCoupure[16] (used by the analysed function) has the values 17, 0, 33, ..., 66.

Another pre-condition Caveat has to know is the set of possible values of the global resource used to store the previous computed value.

Pre :
(5)Stocke_dem=0 ∨ Stocke_dem=17 ∨ Stocke_dem=33 ∨
Stocke_dem=50 ∨ Stocke_dem=66 ∨ Stocke_dem=81;

The property : Writing a property containing a lot of operands and/or operators in Caveat property language can be very difficult, because it leads to a very complicated formula. It is not due to the tool but to the fact of expressing the property by a mathematical formula. To solve this problem, the best way is to use Caveat incrementally. The purpose here is to let Caveat generate a part of the formula. This can be done using an Assert property. To introduce such a property into a C function,

it is necessary to put a label in the code at which the Assert property must be true. Then Caveat can compute the condition on the function input operands that leads to verifying the property.

Applying this method to the current example, an Assert is used to determine the mathematical predicate equivalent to the proposition (which belongs to the property to be verified) "the currently computed value (from the current hardware register value) is not the smallest" (in the code, the smallest value is 17).

The assert property is :

(6)Assert : At label1 ↑(IIS_Ri_TabCoupure'+val) ≠ 17;

With this property, Caveat will compute the condition for obtaining a value not equal to 17 from the const array IIS_Ri_TabCoupure.

Let C2 be the condition Caveat computed.

The result is :

C2 :

```
¬ (((if  bit&32(↑(@ad12345678), 12)
then (if  bit&32(↑(@ad12345678), 11)
then (if  bit&32(↑(@ad12345678), 10)
then 17=↑(IIS_Ri_TabCoupure+~14)else
else 17=↑(IIS_Ri_TabCoupure+~6))
                  else (if  bit&32(↑(@ad12345678), 10)
                    then 17=↑(IIS_Ri_TabCoupure+~10)
                    else 17=↑IIS_Ri_TabCoupure+~2)))
                 else (if  bit&32(↑(@ad12345678), 11)
                  then (if  bit&32(↑(@ad12345678), 10)
                    then 17=↑(IIS_Ri_TabCoupure+~12)
                    else 17=↑(IIS_Ri_TabCoupure+~4))
                  else (if  bit&32(-(@ad12345678), 10)
                    then 17=↑(IIS_Ri_TabCoupure+~8)
                    else 17=↑(IIS_Ri_TabCoupure)))
⇒ bit&32(↑(@ad12345678), 13))
⇒ bit&32(↑(@ad12345678), 13)
                 ∧ (if  bit&32(↑(@ad12345678), 12)
                  then (if  bit&32(↑(@ad12345678), 11)
                    then (if  bit&32(↑(@ad12345678), 10)
                      then 17=↑(IIS_Ri_TabCoupure+~15)
                      else 17=↑(IIS_Ri_TabCoupure+~7))
                    else (if  bit&32(↑(@ad12345678), 10)
                      then 17=↑(IIS_Ri_TabCoupure+~11)
                      else 17=↑(IIS_Ri_TabCoupure+~3)))
                  else (if  bit&32(↑(@ad12345678), 11)
```

then (if bit&32(↑(@ad12345678), 10)
then 17=↑(IIS_Ri_TabCoupure+~13)
else 17=↑(IIS_Ri_TabCoupure+~5))
else (if bit&32(↑(@ad12345678), 10)
then 17=↑(IIS_Ri_TabCoupure+~9)
else 17=↑(IIS_Ri_TabCoupure+~1))))
∨ bit&32(↑(@ad12345678), 17)
∨ bit&32(↑(@ad12345678), 22))

*Note*s. eg : the predicate "bit&32(↑(@ad12345678), 13)", in Caveat property language, is TRUE if the bit 13 of the data stored at address 12345678 is set (=1).

As we can see, it is better to let Caveat provide the formula.

The property, as written in step 1, is composed of three propositions : "the stored value is the smallest (among the possible ones) ", "the currently computed value (from the current hardware register value) is not the smallest" and "the result cannot be the smallest". With the Assert property computation, the second proposition has just been computed. The other two are quite simple, they do not need any intermediary computation.

So, the property is :

$$(7)\text{Post : Stocke\_dem'}=17 \wedge C2 \Rightarrow \text{TypeDemarrage} \neq 17;$$

where C2 is the condition computed by Caveat using the Assert property.

Step 6. Caveat proves this post-condition ("V").

This example shows that a logic based on bit-structured data coming from the hardware can be efficiently verified without having to use real or even simulated hardware. Again it allows a gain in terms of verification cost.

4. Conclusion

4.1. Lessons Learnt

Caveat and the way we have used it on these examples shows its ability to meet the requirements stated in section 1.3 : (1) ability to prove avionics C code, (2) ease of learning and use, (3) early payback, (4) easy integration.

Requirements 1 and 2 are directly met by Caveat, as explained in chapter 2. Practical examples did not reveal any major difficulty around the C language used.

This examples also revealed that the initial training effort that a "standard" software engineer needs to produce is not greater than for another industrial software verification technique : one week's training is enough to be able to perform formal proofs with Caveat.

Requirement 3 should be met if we consider that a lot of tests (mainly unit tests) will be simply replaced by formal proof, and that each formal proof can be performed in significantly less time and with less hardware than the corresponding tests. The ease of learning and use will also help to meet this requirement.

Requirement 4 is met by the fact that the introduction of the formal proof technique in our development methodology (see sect. 1.5 and 3.1) does not change it fundamentally.

These examples also reveal the necessity for some enhancements of Caveat. They will be implemented soon. The main point is the improvement to automaticity in theorem proving.

4.2. Qualitative/Quantitative Results

Qualitative results.

♦ The good level of maturity of Caveat plus the enhancements planned for this year (better automaticity in theorem proving) confirm its ability to improve the verification of avionics applications significantly.

♦ The property language of Caveat can easily be used by a software developer.

♦ The incremental way to use Caveat helps manage the complexity of the mathematical representation of the source code (see example 3 in section 3.3).
♦ The tool and the method for using it make it possible to detect software faults and find counter-examples which can be used in debug sessions.

Quantitative results.

♦ The computation time of Caveat is compatible with an industrial usage.

♦ In terms of cost effectiveness the examples described in this document reveal the three types of gain stated in section 3.1.

Example 1 is very expensive in terms of testing effort in the sense that the required property depends on thirty-two inputs. So, by using the property proof technique, it is no longer necessary to generate and execute a large amount of test cases (first type of gain).

Example 2. In this case (and for this type of code, i.e. checks implemented by "if..else.."), the gain, in terms of time spent, is around 20 %. If we consider that a lot of avionics functions are similar, in terms of implementation, to example 2, it is possible to replace unit testing on these functions and so, the gain on a quite large part of an avionics software product would be about 20 %.

Example 3 : in this case, the cost is reduced because less hardware is needed to perform the verification. In a classical approach, the kind of verification of example 3 is performed during the integration phase, on the real hardware. So the third reason why cost can be reduced is also true in this example.

After these experiments, and the finalisation of the formal proof methodology, it will also be necessary, for the overall verification process (not only unit tests and safety verification) to define a methodology which combines classical tests with property proof.

References

1. C.A.R Hoare : An axiomatic basis for computer programming, Comm. ACM 12 (10), 576-580, 583 (Oct. 1969).
2. Dijkstra BW 1976, A discipline of programming, in *Series Automatic Computation,* Prentice Hall.
3. Pavey D et al. 1997, LAW : Legacy Assessment Workbench, in the UK *Reliability and Metrics Club's news letter.*
4. A joint RTCA-EUROCAE achievement : DO-178B / ED-12B, Software considerations in airborne systems and equipment certification (Dec. 1992).
5. HOOD Technical Group, Jean-Pierre Rosen : HOOD - An industrial approach for software design (1997).
6. A. Smith - MALPAS Userguide. *Technical Report*, Rex, Thomson & partners Limited, 1991.
7. R. Cohen - Proving Gypsy Programs, in *CLI Technical Reports*, 1989.
8. Sj. Garland & Jv. Guttag - A guide to LP, the Larch Prover, MIT Laboratory for Computer Science, 1991.
9. JM. Spivey - The Z Notation, A Reference manual. University of Oxford, 1988.
10. CB. Jones - Systematic Software Development Using VDM. Prentice Hall Int., 1986.