Formal Verification of Hand-Coded Software Some Industrial Experiments and Lessons Learnt

E Ledinot, D. Pariente

Formal Verification of Hand-Coded Software Some Industrial Experiments and Lessons Learnt

E. Ledinot[1], D. Pariente[1]

1: Dassault Aviation, 78 Quai Marcel Dassault 92552 Saint-Cloud Cedex
{emmanuel.ledinot, dillon.pariente}@dassault-aviation.fr

Abstract: This paper gives an account of an on-going attempt to prove the safety properties, of a hand-coded safety critical embedded software of industrial size. The method used is based on annotating the C source files with assertions that encode the safety-related functional properties to be satisfied by the software, and then generating proof obligations to be discharged by some theorem provers. We discuss what has been achieved and what difficulties were encountered, from which we derive requirements regarding the evolution of the verification tools involved in that experiment.

Keywords: Software verification, Hoare Logic, Theorem Proving, Caveat, Caduceus, Why, Coq, Simplify.

1. Introduction

Since 1990, Dassault Aviation has carried out numerous formal methods studies and assessments. The first ones were focused on synchronous languages (first Esterel [4], then Lustre), for control and data flow formal specification, coding and model checking through collaborations with research teams. Over the last few years, much effort was devoted to the integration of UML modelling and signal flow programming (Matlab, Scade, Esterel), in order to introduce these new methods and tools in the Flight Control System (FCS) software development process.

Following the Model Driven Engineering approach (MDE), parts of an embedded software were re-engineered with Matlab/Simulink, Rational Rose and Esterel Studio [17]. By the end of 2003, the first control module formally specified in a graphical way, automatically generated (~15 Kloc), and proven was embedded in a military aircraft operational software.

In the meantime, some experiments on formal verification of hand-written code were initiated, because in numerous situations pieces of software of high criticality level cannot be generated from formal specification models. This is especially the case for programs encoding algorithms with complex control or data structures (nested loops, extensive use of pointers, etc.), or for system level programming (drivers, schedulers, coding/uncoding of data formats etc.). This is also the case when software engineers don't want to be constrained by restricted specification languages.

This is the basis of our motivation to assess the theorem proving approach to formally verify annotated hand-coded imperative programs, from which are generated proof obligations to be later discharged by automatic deduction tools, or by interactive theorem provers when the available decision procedures fail.

In the following, we will briefly introduce the underlying technical background and the two tools we have been assessing for the last two years: Caveat [6],[3] and Caduceus [5]. Then we proceed by stressing some of the lessons we learnt and by presenting some general requirements to improve these formal verification tools. We aim to use them operationally in the mid-term, trying to follow the way that has been paved by B-method & tool [1].

2. Background

To be able to present our past and current experiments in proving the correctness imperative programs, we propose at first a brief and non formal presentation of the underlying method and foundational background. For rigorous explanations, please seek the abundant references and tutorials related to the subject, for instance [13].

2.1 Reminder of basics : Floyd-Hoare's logic

Correctness proof of imperative program is essentially based on Floyd-Hoare's logic [12]. The main purpose of this logic is to provide a set of rules defining how to formally reason about the properties of imperative programs.

Hoare's logic is based on a triple which describes how the execution of a statement changes the state of data variables. A Hoare triple is defined as :

$$\{p\}\ S\ \{q\}$$

where p and q are respectively the precondition and the postcondition (expressed as first order logic formulas), and S is a programming statement. The whole expression means that if p holds before the execution of S, then q will hold after the execution of S, *if* S terminates (partial correctness).

The assignment rule is the following one :

```
Rassign:      -----------------
              {p[y/x]} x:=y {p}
```

which states that if p holds after the assignment (x:=y), then p, where all occurrences of x have been substituted by y, holds before that assignment. For example, *{x>-1} x:=x+1 {x>0}* is a valid triple.

In the same way, Hoare defined the following well known rules:

```
Rsequence:    {p} S {q}     {q} T {r}
              ---------------------
                     {p} S ; T {r}

Rcondition:   {p /\ C} T {q}{p /\ ¬C} E {q}
              ------------------------------
              {p} if C then T else E {q}

Rloop:   {p}=>{inv}      {inv  /\  C}  S  {inv}
              {inv /\ ¬C} => {q}
         ---------------------------------
              {p} while (C) do S {q}
```

where *inv* is an invariant of the loop, which holds before, after, and for each execution of the loop body. As a matter of fact, this rule describes a recurrence principle. Since these rules are partial correctness rules, loop termination for instance has to be established. This is done by identifying a loop variant (*var*) which must be positive when entering the loop, and decrease towards 0 at each loop turn.

2.2 Verification conditions, proof obligations and weakest preconditions

A weakest precondition (WP) is computed by backward propagation of a given property through statements, according to Hoare's logic rules. Hence, a WP function takes two arguments: a statement *S* and a property *q*, and returns the most general condition *c* over the execution state just before executing S, such that *q* is valid after *S* execution. This is usually written:

```
c := WP( S, q )
```

A verification condition is defined w.r.t. a given property *q* to be proved valid after execution of a statement *S*, and a precondition *p* which holds before *S*, as follows: *q* holds after *S* if *p* holds before *S*. Formally, we can write the verification condition as:

```
p => WP( S, q )
```

Sometimes, verification conditions may be too difficult to prove in only one formula like the previous one. Typically, for loops, the verification condition has to be split into several proof obligations; if we want to prove valid the following triple:

```
{p} while (C) do S {q}
```

whose verification condition is:

```
p => WP( while (C) do S, q )
```

we will have to prove valid three proof obligations related to invariance (the necessary invariant *inv* is generally provided by the user), and two proof obligations related to termination (w.r.t. the variant *var*).

2.3 Hoare's logic extensions: dealing with function calls

Hoare's logic needs to be extended so that WP can still be computed through function calls. The solution can not be to inline called functions as it could make WP computations rapidly explode in size. The way current program provers work is more reasonable: function calls are replaced with the properties (pre and postconditions) of the called functions.

In the following example, in the body of function *f*, the call to function g will be "replaced" by $\{pre_g\}$ and $\{post_g\}$ which are respectively the precondition and postcondition of function *g* (after substitution of function parameters and predicate parameters):

```
{pre_g}
void g(…) { … }
{post_g}


{pre_f}
int f(…) { Sf1;
g(…);
Sf2;}
{post_f}
```

Before propagation of pre/postconditions of called functions

```
{pre_g}
void g(…) { … }
{post_g}

{pre_f}
int f(…) { Sf1;
       {pre_g} /* g(…); */ {post_g}
       Sf2;}
{post_f}
```

After propagation of pre/postconditions of called functions

The new verification condition to be proven valid is:

$\{pre_f\} \Rightarrow WP(Sf1,\{pre_g\})$

which specifies that the preconditions of function g must be satisfied by any statement executed before. The verification condition :

{pre_f} => WP(Sf1,{$post_g$}=> WP(Sf2,{$post_f$}))

which expresses backward propagation of postcondition *post*$_f$ through the code of function *f*, and "embedding" the postconditions of function *g*.

Remark about WP computation:
It is worth mentioning that, in many tools, WP is not computed as described, i.e. by backward propagation of properties through statements. The actual computation is based on building a universally quantified formula for all successive states of all assigned variables, and ending with the user's postcondition. The following example of code and its associated verification condition illustrates this mechanism:

```
int x,y;
void g();
{x>y*2}

void f()
{
        x=0;        forall x0:Z, x0=0
        y=1;        => forall y0:Z, y0=1
        g();        => forall x1:Z, x1 > y0 * 2
        x=x+5;      => forall x2:Z, x2 = x1 + 5
}
{x>0}               => x2 > 0
```

RHS: code with function assignments and call

LHS: generated verification condition

The two functions *f* and *g* are respectively annotated by postconditions. The verification condition is displayed on the right hand side: it is computed on the basis of a forward propagation and quantification instead of classical WP computations.

3. Assessment of verification tools

In this paragraph, we will focus on two different C program verification tools: Caveat, developed at CEA [6] and Caduceus developed at LRI [5]. Caveat and Caduceus are based on Hoare's logic and are both dedicated to proving correctness properties specified through annotations of C statements. They propose different, and to some extent complementary, features, methodologies of use, advantages and limitations.

Both tools were assessed on the same embedded software, a 70+ Kloc module in charge of fault detection, isolation and recovery (FDIR) for a group of sensors. It mainly consists in voting algorithms and a failure management logic. About 70 consolidated parameters are delivered by this sensor management module, and five safety properties have to be proved per parameter.

3.1 Experiments with Caveat

Caveat [6], [3] is a static analysis tool designed to help verify safety properties on critical software. It operates on ANSI C programs. It was developed by CEA, the French Nuclear Agency, and is used as an operational tool by Airbus-France (to replace some unit tests by formal verification), as well as EdF, the French electricity company.

It is based on WP computation and first order rewriting techniques (proof engines). The main features of Caveat are navigation facilities, property synthesis, and automated or interactive deduction.

Program verification methodology, in brief
First step: Caveat performs an automatic analysis of the whole project, which leads to the generation of:
- properties related to preventing runtime errors such as null pointer assignment, division by zero, out of bound array access, etc,
- the input variable list,
- the output variable list (including side-effects),
- the functional expressions for every function output.
Depending on selected strategies (discussed below), these functional expressions can be used as postconditions during WP computations.

Second step: the user can annotate the code with:
- other preconditions,
- postconditions, especially when Caveat was not able to generate functional expression for outputs needed to prove a given property (failing to generate a functional expression is generally due to the presence of a loop),
- loop invariants and termination conditions (variants),
- intermediate assertions (optional), aiming to simplify the task of WP computation.

Third step: a given property (a proof obligation, a postcondition, or an assertion) is selected by the user who then triggers the Caveat simplifier and prover to prove that property. Caveat returns either true, or false (in case of false, a counter-example is displayed), or provides the user with a residual formula (remainder) whenever the rewriting engine was not able to prove the property.

In case of a residual formula, the user is provided with an interactive rewriter. The rewriting task (e.g.: transforming the given formula into conjunctive normal form, disjunctive normal form, splitting it by case, folding/unfolding definitions, etc.) aims at transforming and simplifying manually the residual as much as possible, in order to obtain a true or false value or a remaining formula that can not be simplified anymore. Often, this remaining formula is a missing precondition that the user should add as a new annotation.

Note that the remaining formula must be accurately analyzed by an expert user familiar with logic and with a deep knowledge of the C code, as it is generally not clearly readable (residuals are the results of WP computation and automatic rewritings that render original properties quite unrecognizable!).

User strategies
It is important to note that the following strategies can improve effectiveness and productivity of program verification with Caveat:

- postconditions annotated by the user can be preferred to functional expressions generated automatically,
- user assertions can be left over during WP computations,
- when taking into account postconditions or assertions, it is possible to ask Caveat to quantify any variable belonging to the predicates. Universally quantified variables do not need to be dealt with by WP computation. This results in saving computation time, and in simplifying the proof obligations and proof residuals.

Any use of these strategies needs a deep analysis and experience, as its efficiency strongly depends simultaneously on the structure and size of the properties, verification conditions and source code.

Experimenting on small examples is of course relevant, but does not provide all the needed insights when facing an actual industrial size software.

Some lessons learnt :
Compared with other program verification tools, and after experimenting with Caveat on real applications, the major advantage of the tool turns out to be the generation of functional expressions for all function outputs (these functional expressions are used as automatically generated postconditions). During our assessment, we definitely appreciated this functionality.

Some drawbacks imposed limitations difficult to overcome. For example, we had too often to cope with the automatic prover weaknesses: this one was not powerful enough to handle large size pieces of code and large verification conditions.

Even for some proof obligations that looked trivial, a proof was not always computable: the automatic rewriting engine of Caveat (without any heuristic or means to disable this automatic step) systematically enlarged the verification condition size so that subsequent manual interactions were unfeasible in an economically reasonable amount of time.

Some limitations were also encountered on non ANSI C syntactic features, and most importantly on aliases (pointers pointing to the same memory location). Unfortunately the embedded C code we chose to experiment with Caveat made extensive use of aliases that Caveat couldn't handle.

These problems and a few others are going to be fixed. As an example, Caveat is known to be able to deal with aliases in its latest releases, and work is done to automate relevant invariant generation using abstract interpretation.

Results
We succeeded in proving four safety properties over a 3Kloc alias-free sample of the 70+ Kloc software mentioned previously. After a significant training period that made a software engineer become a Caveat experienced user, it took him nearly six man-months to achieve this encouraging but limited result.

A first objective of the evaluation was to prove five safety properties for one parameter on the full code, and then to prove the 350 properties (5 properties for each of the 70 parameters). But the software to be proven correct, which was developed in parallel with our experiment, turned out to make an extensive use of pointers and aliases. Unfortunately, two years ago Caveat was not able to handle true aliases efficiently. To try to overcome this limitation, we moved to Caduceus whose first assessable version was released in September 2004.

3.2 Experiments with Caduceus

Caduceus [5] is a promising new C program verification tool which is still under development at LRI - Université Paris Sud. It is a proof obligation generator also based on Hoare's logic, though with some foundational background rooted in the type-theoretic approach of proof assistance [10], [8]. This background may have some importance in the long run when verification of verification tools will become a major issue, especially for the use of this kind of tools on software development processes subject to certification.

Program verification methodology, in brief
At first, an important point to outline is Caduceus' ability, thanks to an inner memory management model, to deal efficiently with aliases. This allowed us to apply Caduceus to our benchmark 70+ Kloc C embedded code, containing potential and actual aliases.

The first step of the program verification methodology consists in annotating the C code. For the moment, and it is to our opinion the main drawback of Caduceus, no functional expression is automatically generated from program expressions for output or global variables. This results in a big difference with Caveat annotations: Caduceus annotations are more voluminous and cumbersome to write because arithmetic and logical computations have to be manually paraphrased at the predicate level. On large pieces of software the size of annotations blows up, jeopardizing scalability.

Annotations are specified as comments inside the code (especially for loop invariants), or in the function declaration files (as headers ".h" files). These annotations are expressed into Java Modelling Language (JML) and some extensions. JML is a behavioural interface specification. It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus. JML is not limited to Java code, of course, and LRI has adapted it to the needs of property specification on ANSI C programs.

As usual, the possible annotations are preconditions, postconditions, invariants and variants, assertions, and assignments (i.e., for each function or loop, the list of updated/assigned variables must be provided by the user).

Of course, when working on a large source code, some of the annotations mentioned above are *very* time-consuming to write manually. In many cases, trivial postconditions or preconditions, simple invariants and variants, and *all* assigned variables could be generated automatically.

In the particular case of safety critical embedded software for which only a subset of possible C statements are authorized, a lot of non functional annotations could be generated as early as the parsing phase of the code.

LRI is currently working on these issues, and some important results are expected in the next months. To deal with the 70+ Kloc code we developed an in-house prototype of annotation generator. This generator, mainly developed in CAML, produced automatically (in 10 minutes on a 1GHz PC) all the expected simple invariants, variants, loop and function assignments, and most of the preconditions related to non-null pointers. The same task done manually by an experimented user was evaluated to a 3-month workload. It took 3 weeks to develop the first version of this prototype generator which does not address all possible C statements, but only those currently used in our benchmarking embedded code.

Second step: what we call Caduceus is indeed a tool suite (Fig. 1). It is a pipeline of processors: the Caduceus compiler feeding the Why compiler [10], [16], which in turn generates proof obligations in different formats to be handled by different theorem provers: interactive proof assistants such as Coq [8] and PVS or decision procedures such as Simplify [15] and haRVey [11].

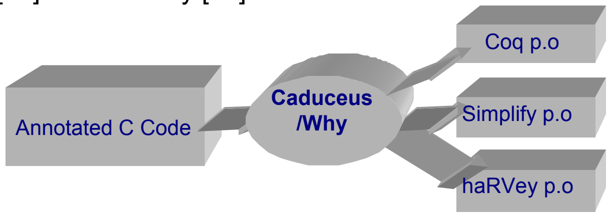

Figure 1: The Caduceus/Why processing flow

From the end-user point of view, the Why compilation is completely transparent, and it is mentioned here because Why is used as an intermediate language. The Why language is able to express high level order functions and to manipulate abstract types, thus providing a powerful means for WP computation. Why is designed so that type theory based consistency checks of the generated proof obligations with respect to annotations could be performed.

An important point to notice is that the annotation style depends on the targeted theorem prover: the way hypotheses and axioms are expressed must be defined according to some inner features of the targeted provers.

For example, with Simplify, the following two sets of axioms, that both define by recurrence the sum of the elements of an array, are treated quite differently by Simplify's deduction strategies:

```
axiomA1: \forall int tab[], int i, int j;
i>j => sum(tab,i,j)==0
axiomA2: \forall int tab[], int i, int j;
(i<=j)=>sum(tab,i,j)==tab[i]+sum(tab,i+1,j)

axiomB1:   \forall   int   tab[],   int   i;
sum(tab,i,i) == 0
axiomB2: \forall int tab[], int i, int j;
sum(tab,i,j) == tab[j] + sum(tab,i,j-1)
```

Axioms A1 and A2 behave well with Simplify's proof engine, while B1 and B2 make Simplify loop for ever.

Of course, a Simplify specialist would certainly find a rationale to explain that behaviour (and Prolog programmers will be familiar with logically equivalent definitions that are drastically different from the computational point of view!).

But from a software engineer standpoint, this kind of sensitivity is somehow unfortunate: the annotated C source files should be unique, whatever number of deduction tools are used in the subsequent stages of the verification process. The annotations should depend only on the software correctness properties to express and on human readability. One would like to have them not depending on tricky and obscure tool-feature issues.

Some lessons learnt
At this time, the experiment with Caduceus amounts nearly to one man.year effort and is still in progress. The lessons learnt by mid-2005 drove deep modifications to Caduceus, especially on the C memory model that axiomatizes pointers' management. These modifications will be evaluated in 2006,.and we hope that by then many current limitations will be overcome.

We successfully made the proofs at unit level, and at the lower levels of the software integration hierarchy. More precisely, the call-graph of the software is a 11-level call tree. The main module is at, level 1 (the highest), and unit proofs are made at level 11 (the lowest). We managed to carry out proofs of the five properties for one parameter from level 11 to level 6.

At levels higher than 6, the number, size and structure of the generated verification conditions were such that neither Simplify nor haRVey managed to prove them.

Let us give briefly some insight on why the higher the proof in the program hierarchy, the more complex were the generated verification conditions. It was mainly due to Caduceus' model and default assumptions on memory separation of global variables.

Memory separation had to be demonstrated for every couple of identically typed fields of global structure variables (used to store sensor measures). Basically these verification conditions consisted in proving that two pointers' base addresses and offsets were unequal. With more than 100 identically typed global structure fields at level 5, 4950 huge non tractable verification conditions were generated…

From level 11 to level 6, an average 60% of the verification conditions were discharged by means of automatic decision procedures. Some of the remaining unproved formulas were proved interactively with Coq. Some specialized and effective tactics were defined in Coq with the macro language to assess the possibility of speeding up these interactive proofs.

In the end, the feasibility of proving the five safety properties at the software boundaries (level 1) is not yet demonstrated. But the reason why we didn't manage to do so are well understood. LRI has undertaken major modifications of the memory model that should lead to smaller and fewer verification conditions at the expense of more sophisticated static analyses in Caduceus. It is expected that Simplify will have a lower failure rate on valid verification conditions generated by this forthcoming version of Caduceus, and that making proofs from levels 5 to 1 will become tractable.

We came up with a few other insights in the course of the experiment:

- So-called "ghost" variables turned out to be needed in Caduceus' annotation language: JML formulas are closed formulas whose only free variables are program variables. For cyclic control programs that do not store in state variables some past values (that may be useless from the programming point of view but that are mandatory to formally specify some functional *temporal* properties), one needs to add modifiable "virtual program variables" at the annotation level to store these states and refer to them.
- Having the ability to use several automatic deduction tools to prove the verification conditions is a definite advantage: cross-demonstration of the same property by two different provers (which are magic "black boxes" from the software engineer point of view) increases the level of confidence one may have in these verification elementary steps. This kind of multiple demonstration approach is likely to be valuable in the future when arguing the case for using this tool suite in certification contexts (DO-178C more specifically).

To sum-up, we mainly faced scalability problems, but we expect to overcome them shortly with the next Caduceus release.

We have also uncovered some phenomena that greatly impede our verification productivity. The most important one is the great sensitivity of the success rate of automatic deduction tools to any change upward in the verification process. Any change in the annotation style (remember the sum(tab) example above), or in Caduceus/Why implementation, or in the automatic deduction tools, may cause some unforeseen regressions in the progress of the verification task: some proved verification conditions were no longer automatically provable while the code and annotations had remained unchanged.

Since proving a program is by far more time consuming than compiling or testing a program, some "robustness" is mandatory to have these techniques economically viable on large projects.

4. Towards IPEs : Integrated Proof Environments

Many issues already tackled in software development processes should be addressed as well in an industrial proof assistance environment dedicated to program verification.

It could be useful to consider that proof results (from decision procedures) or proof terms (from theorem proving assistants) should be seen as “binary files”, or object files. In that way, files of annotated C instructions as well as proof scripts of interactive provers can be considered as source files. As for the provers (be they automatic or interactive), they woud play a role similar to that of compilers. Following this analogy, the need for versioning, configuration management, automatic reassembly of “binary objects” etc. should appear obvious in program proof development processes.

To elaborate a bit on that, we clearly felt the need for an integrated development environment (IDE) and related project management tools.

Some interesting developments along these lines have been made with Eclipse at INRIA Sophia: JACK (Java Applet Correctness Kit) [14] already provides some very useful functions such as traceability from proof obligations to annotations, proof validity management, computation of project progress statistics, impact analysis and incremental automatic reconstruction of “binary objects” triggered at high level (at the annotation level for instance).

Of course what turned out to be necessary in Java applets formal verification projects is also necessary for any other similar project on any other programming language. A Caveat and/or Caduceus Eclipse plug-in similar to JACK’s will be developed once the scalability problems are solved.

It is likely that an industrially mature C correctness proof environment will also have to integrate other verification tools such as static analyzers, and perhaps even model checkers.

As mentioned before, on industrial size projects the `\assigns` annotations can’t be manually provided by users. A static analyzer front-end has to generate them automatically. Some kinds of loop variants and invariants can also be automatically generated, thus sparing software engineers costly man.months of boring work. Software engineers should only be in charge of functional annotations and be released from manually setting annotations that could result from program control and data-flow analysis.

We wonder whether static analysis based pre-processors may not have to be introduced beyond mere assistance to setting non functional annotations.

Since we are facing a complexity issue at the higher levels of software integration, we wonder as well whether a code slicer would be helpful. To reduce the size of the verification conditions, one would generate them only on slices of the code, one slice per annotation.

Of course, such an approach has major drawbacks, especially if one envisions to use these techniques in D0178-compliant processes. We definitely hope that the new Caduceus memory model will solve the v.c size explosion problem. If not, we would need to resort to a program slicer.

We also had a look at other techniques to cope with software complexity, such as pre-processing the code by predicate abstraction techniques [2][7] and then using model checkers on the abstracted model. We will investigate such techniques in case of failure of the theorem proving approach, or later as an additional mean of enhancing productivity in formal verification of hand-coded software, especially for temporal properties.

5. Conclusion

We have discussed an on-going attempt to prove about 350 safety properties on a 70+Kloc manually written C program, featuring many aliases and pointer issues.

We started with Caveat by 2003 and then moved to Caduceus by mid-2004, because its memory model could handle true aliases that Caveat could not at that time. It was the first release of Caduceus, which was still in its infancy.

Although we used two theorem provers, with great support by their development teams, and devoted a 1,5 m.y effort to the project at Dassault, we still did not manage to achieve our goal, 35 years after Hoare’s seminal paper [12],

We are still facing scalability issues. However we hope to solve them in the next few months thanks to major modifications in Caduceus’ C memory model. We have obtained so far limited but encouraging results, therefore we remain confident in the feasibility of our project.

Once the five properties are proved at the main level for one parameter, we will face a productivity issue.

In some sense, the overall verification project consists in 70 similar verification sub-projects: the same five safety properties have to be proved on 70 different but similar parts of the program (one per voted sensor parameter). We would like not to spend 70 times the effort spent on the five first properties for the first parameter.

Failure rates and strategies of automated deduction tools, Coq tactics and tacticals like any other kind of automation tool, will have to be tuned carefully to take advantage of the structure and similarities of the code parts, and thus spare a significant amount of verification effort. True feasibility will be established once the 350 properties over the 70 parameters are proved within a 1 to 3 m.y effort.

By then, the industrial maturity of the tools, their integration in some of our current processes and D0178 related issues will be at stake.

6. Acknowledgement

The authors are grateful to P. Baudin, J. Raguideau, D. Schoen, J.C Filliâtre and C. Marché for their reactivity and technical support for the last two years.

7. References

[1] Behm P., Benoit P., Faivre A., Meynadier J-M : "*Météor : A successful application of B in a Large Project*", FM99 - Formal Methods, (Toulouse, France), 1999 LNCS 1708, Springer Verlag p369-388.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, Sriram K. Rajamani, *Automatic predicate abstraction of C programs,* Volume 36, Issue 5, ACM Press New York, NY, USA, May 2001.

[3] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen and N. Williams, *"CAVEAT: A Tool for Software Validation"*, International Conference on Dependable Systems and Networks DSN 2002, Washington D.C., USA, June 2002.

[4] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, Robert de Simone*, "Esterel: a formal method applied to avionic software development"*, Science of Computer Programming, v.36 n.1, p.5-25, Jan.1.2000

[5] http://caduceus.lri.fr

[6] http://www-list.cea.fr

[7] Edmund Clarke, Daniel Kroening, Natasha Sharygina, Karen Yorav, *Predicate Abstraction of ANSI-C Programs Using SAT*, Formal Methods in System Design, v.25 n.2-3, p.105-127, September-November 2004.

[8] http://coq.inria.fr

[9] Patrick Cousot, Radhia Cousot, "*Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints"*, Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, p.238-252, January 17-19, 1977, Los Angeles, California.

[10] Filliâtre J.C. "*Preuves de programmes impératifs en théorie des types*", Thèse de l'Université Paris Sud-Orsay, juillet 1999.

[11] http://www.loria.fr/equipes/cassis/softwares/haRVey

[12] C. A. R. Hoare, "*An axiomatic basis for computer programming"*, Volume 12 , Issue 10, p576 - 580, ACM Press New York, NY, USA October 1969.

[13] Hoare C. A. R., Jifeng H. : "*Unifying Theories of Programming*", Prentice Hall, 1998.

[14] http://www-sop.inria.fr/everest/soft/Jack

[15] D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover - http:// research.compaq.com/src/esc/simplify.html.

[16] http://why.lri.fr

[17] Yann Le Biannic, Eric Nassor, Sylvain Dissoubray, Emmanuel Ledinot, *« Spécification Objet UML de logiciels temps-réel »*. Proceedings de la conférence Real Time Systems, Paris 28-30 mars 2000.

8. Glossary

CEA	Commissariat à l'Energie Atomique
FCS	Flight Control System
IDE	Integrated Development Environment
LHS	Left Hand Side
LRI	Laboratoire de Recherche en Informatique
MDE	Model Driven Engineering
p.o	proof obligation
RHS	Right Hand Side
UML	Unified Modeling Language
v.c	verification condition
SW	Software
WP	Weakest Precondition