Proving Temporal Properties at Code Level for Basic Operators of Control/Command Programs

P. Baudin, D. Delmas, S Duprat, B Monate

Proving Temporal Properties at Code Level for Basic Operators of Control/Command Programs

P. Baudin[1], D. Delmas[2], S. Duprat[3], B. Monate[1]

1: CEA LIST, 91191 Gif-sur-Yvette Cedex, France
2: Airbus France, 316, route de Bayonne, 31060 Toulouse Cedex 03, France
3: Atos Origin, 5, avenue Albert Durand, 31700 Blagnac

Abstract: More and more control/command software is being generated automatically from high-level graphical specifications. Such specifications are typically synchronous data-flow models, built on a set of external basic operators to be implemented in a lower-level language. The semantics of the overall model depends therefore on the semantics of the basic operators, which can be expressed in terms of temporal (inductive multi-cycle) definitions. In this paper, we describe a way to specify and verify these operators formally, using theorem-proving techniques. We report on experiments conducted to prove multi-cycle properties on actual embedded basic operators written in C, using the CAVEAT static analyser with a dedicated method.

Keywords: formal verification, theorem-proving, static analysis, CAVEAT, embedded software

1. Introduction

1.1 Context

Many control/command programs, such as fly-by-wire control software, as well as some other embedded reactive systems, are currently being developed in a model-based approach. Most of their source code is generated automatically from high-level synchronous data-flow specifications. As a consequence, their overall structure is the following:

```
initialise state variables;
loop forever
  read volatile input variables,
  compute output and state variables,
  write to volatile output variables;
  wait for next clock tick;
end loop
```

The computations to be performed are described by system designers at model-level in a graphical stream language such as SCADE™ or Simulink®, by means of external basic blocks. The semantics of the overall model relies therefore on the semantics of the basic operators. Designers only need to know their specifications, which are often expressed in terms of informal inductive definitions. These definitions are multi-cycle properties that describe the (discrete-time) temporal behaviour of the operators with respect to the synchronous clock. The actual operators can be either provided together with the modelling tool, or implemented by the user in a lower-level programming language, for him to meet his specific needs exactly. The latter case occurs typically for safety-critical embedded avionics software to be certified according to the DO-178B/ED-12B international standard.

Well-known techniques are available to verify and validate designs at model-level, e.g. model-checking and simulation. Such Verification & Validation activities require system-level models for all basic blocks, and their results are only valid with respect to these models. For this kind of V&V to be considered representative with respect to the behaviour of the final generated program, one needs to be sure that each basic operator has the same semantics as the associate model.

The traditional way to address this objective is to verify that each basic operator complies with its informal specification, by means of unit testing. However, this approach is both costly and unsound. In this paper, we describe an alternate way to verify these operators, based on theorem-proving techniques. We report on experiments conducted to prove multi-cycle properties on actual embedded basic operators written in C, using the CAVEAT static analyser with a dedicated method. Then we state the successes obtained, and the limits of this approach, e.g. regarding potential run-time errors, accuracy of floating-point computations and functional correctness of the compiled program. Finally, we give complementary approaches currently investigated to address these issues for the complete program, by means of other specialised static analysis tools.

1.2 Basic operators of control/command programs

Different types of operators are used as basic blocks of control/command software:

- Pure boolean/integer operators: All inputs and outputs have boolean or integer types, and outputs at time `t` only depend on inputs at time `t`. Algorithms use no remanent data, and perform no floating-point computations. Such typical operators are logic gates and boolean switches.
- Pure numerical operators: Most inputs and outputs are real-valued. Outputs at time `t` only depend on inputs at time `t`. Algorithms perform floating-point computations, but use no remanent data. Well-known examples are divisions, square roots, trigonometric and transcendental functions.
- Pure temporal operators: All inputs and outputs have boolean or integer types. Outputs at time `t` depend on inputs at ticks `0, 1, ..., t-1, t` of the synchronous clock. Algorithms perform no floating-point computations, but use remanent data. Such operators include delays, timers, flip-flops, triggers, input confirmation operators, etc.
- Both numerical and temporal operators: Digital filters are typical examples.

1.3 Properties to be proved

We want to prove that C functions or macro-functions implement their inductive multi-cycle definitions properly. To do so, we first need to formalise these definitions.

Let $E[k]=(E_1[k], \ldots, E_n[k])$ be the vector of inputs of the operator at time $k\geq 0$, and let $S[k]=(S_1[k], \ldots, S_p[k])$ be the vector of outputs.

The general form of the definition for the operator is:

```
  S[0] = F0(E[0])
  S[1] = F1(E[0], E[1])
  ...
  S[N-1] = FN-1(E[0], E[1], ..., E[N-1])
for all k≥N:
  S[k] = F(E[k-N],..., E[k-1], E[k])
```

In some cases, expressions involving previous outputs are used:

```
for all k<0:
  S[k] = 0
for all k≥N:
  S[k] = F(E[k], S[k-N],..., S[k-1])
```

2. Overview of the CAVEAT tool

The CAVEAT ([2]) static analyser is being developed by CEA LIST. It is a verification tool for programs written in a subset of the ISO-C language. Being qualified as a verification tool according to DO-178B/ED-12B, it has been used industrially at Airbus for several years, in order to verify a 30000 line embedded software subset formally, instead of performing traditional unit testing ([3]).

CAVEAT aims at translating a C program and its formal specification into a set of first-order logic formulas named “proof obligations” (POs). POs are generated thanks to a weakest precondition calculus “*à la*” Floyd-Hoare ([1]). Therefore, the validity of the POs implies the conformity of the code to its specification. To establish the validity of the POs, a combination of automated theorem proving techniques and manual assisted formula-transformations are being used. Fully automatic theorem-provers are known not to be very efficient at deciding inductive properties, and CAVEAT does not provide any direct support for inductive proofs. However, CAVEAT offers the possibility to define recursive logic functions, known as “lambda functions”. No automatic proof strategy is available on these symbols, but bounded applications can be performed interactively.

Formal specifications are described by properties inserted at specific program points. In order to introduce the different types of properties that can be proved with the tool, let us give a simple example of C function:

```
void div(int a, int b, int *q, int *r)
{
    *q = 0 ;
    *r = a ;
    while (*r >= b)
    {
        *q = *q + 1 ;
        *r = *r    b ;
    }
}
```

This integer division algorithm computes the quotient and remainder (q,r) of two positive integers a and b, such that a ≥ b.

Let us write a formal specification for function `div`, inserting different types of properties at different program points:

- Preconditions: these properties are to be attached to the declaration of functions and give a formal relation between the formal arguments and the global variables. Preconditions are assumed to be verified whenever the function they are attached to is executed and are to be checked at each of its call sites. Preconditions for function `div` are:

```
Pre my_precond1 : b > 0 ;
Pre my_precond2 : a ≥ b ;
```

- Postconditions: these properties are almost like preconditions but for the semantics. For all values of global variables and formal arguments that fulfil the precondition, postconditions must hold at the end of all normal executions of the function to which they are attached. Moreover, at each call site of the function, the postcondition is assumed. Postconditions for function `div` are:

```
Post my_post1 : r.[*] < b ;
Post my_post2 : a = b*q.[*] + r.[*] ;
```

- Loop invariants: these properties are attached to loop bodies in functions, and may give formal relations between all visible program variables. They must hold at the beginning of the loop body they are attached to. Besides, each execution of the loop body has to preserve the property. Invariants for the loop number 1 of function `div` are:

```
Inv 1 my_inv1 : r.[*] ≥ b ∧ b > 0 ;
Inv 1 my_inv2 : a = b*q.[*] + r.[*] ;
```

CAVEAT also has support for the following global properties:

- Logical definitions: logic functions can be defined and used to factorize repetitive formulas. They are denoted like in the following examples:

```
Const succ ∊ int -> int =
    lambda x ∊ int . x + 1 ;

Const is_even ∊ int -> Boolean =
    lambda N ∊ int.
        (N = 2 * (N div 2)) ;
```

- Tautology declaration: specific predicates can be declared to be always true, regardless of any specific program point. They can be used as lemmas to help the proof process, or as means to ensure the completeness of the formal specification. These properties are denoted as follows:

```
Always my_tauto :
    is_even(a) ∨ is_even(succ(a)) ;
```

Many other kinds of properties are available in CAVEAT, but only the above properties are used in the present article.

3. A method to deal with multi-cycle properties

3.1 Practical set-up

Overview

Let us describe a case study on a widely-used temporal operator: the “CONF” input confirmation operator, aiming at triggering an action only after some *stimulus* has been active for long enough.

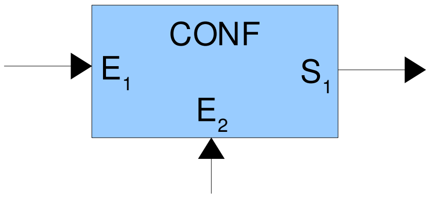

Figure 1: Interface of the CONF basic block

As shown on Figure 1, this operator has the following interface:

- E_1: boolean input variable;
- E_2: non-negative integer constant parameter;
- S_1: boolean output variable.

Functional requirements

An informal specification of this component is: “*S_1 is on if and only if E_1 is on and has been on for the last E_2 ticks of the synchronous clock.*”

Figure 2 shows possible behaviours of the CONF operator with parameter E_2=2.

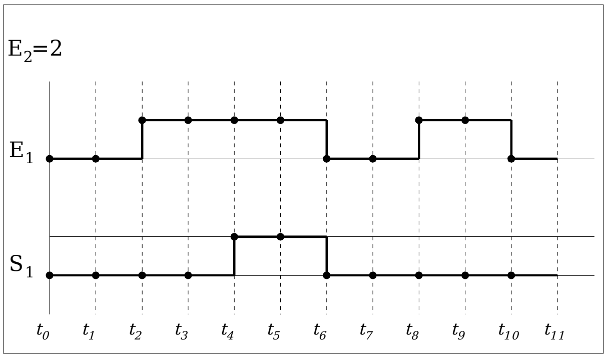

Figure 2: a CONF timing diagram

Let k≥0 be the current value of the synchronous clock, and E_1[k] (resp. S_1[k]) be the value of E_1 (resp. S_1) at time k. Thus, S_1[k] depends on E_1[k], E_1[k-1], ..., E_1[k-E_2].

More precisely:

$S_1[k] = E_1[k] \wedge E_1[k-1] \wedge ... \wedge E_1[k-E_2]$

assuming the following convention:

E_1[i]=0 for all i<0

Implementation constraints

The CONF operator is implemented in a C macro-function with the following interface:

```
#define CONF(E1, E2, S1)
```

Two hypotheses can be used to write the code for this operator:

1. The CONF operator is executed once at every tick of the synchronous clock;
2. all RAM variables have been initialised to zero before time k=0.

Proof Environment

As the informal specification of CONF refers to a synchronous execution model, we need to set up a representative environment. Therefore, we build our proof project as a C source file where the CONF macro-function is being used inside the body of an infinite loop. The loop counter represents the synchronous clock. Besides, we need to store input/output values at all clock-ticks into history-keeping variables. We use arrays for this purpose. Finally, as the CAVEAT tool is designed to prove properties on C functions (as opposed to macro-functions), we use a function wrapper for the CONF macro.

```
void conf_iterate(void)
{
    int i=0 ;
    while (1)
    {
        conf_call(i);
        i++;
    }
}

extern int E[], S[], NB;

void conf_call(int k)
{
    CONF(E[k], NB, S[k]);
}
```

In the above C source file:

- the conf_iterate main function implements the infinite loop;
- the intermediate conf_call function is a wrapper for the CONF macro-function;
- E_1[k] (resp. S_1[k]) is stored in the element at offset k of the E (resp. S) array.

Formal specification

We may now formalise the specification of the CONF operator in terms of preconditions and postconditions of the conf_call wrapper, using the E and S history arrays as formal operands.

To do so, we separate the set of possible “use cases” of the operator into three disjoint conditions:

1- the current input E[k] is off;

2- E[k] is on and E has been on for less than NB clock ticks;

3- E[k] is on and E has been on for NB clock ticks or more.

We notice that these three conditions are combinations of two atomic propositions:

a) The current input `E[k]` is on;

b) `E` has been on for `NB` clock ticks or more.

The former may be expressed directly, using the element at offset k of the `E` history array:

```
LET SUBCOND_INPUT_ON = (E'.[.(k)] ≠  0) ;
```

For the latter however, we need to create a formal expression that represents the number of ticks for which `E` has been on. We do it using the following recursive lambda-function:

```
Const NB_CYCLES_ON ∊ Tab&int -> int -> int =
    lambda input ∊ Tab&int, n ∊ int.
        if (n ≤ 0) then 0
        else(
            if (input.[.(n-1)]=0) then 0
            else (1 + NB_CYCLES_ON(input, n-1))
            );
```

The `NB_CYCLES_ON` function has an array input (`input ∊ Tab&int`) and an integer input (`n ∊ int`), and it returns the number of adjacent non-zero array elements before index `n`. We may visualise the value of `NB_CYCLES_ON(E,k)` on Figure 3.

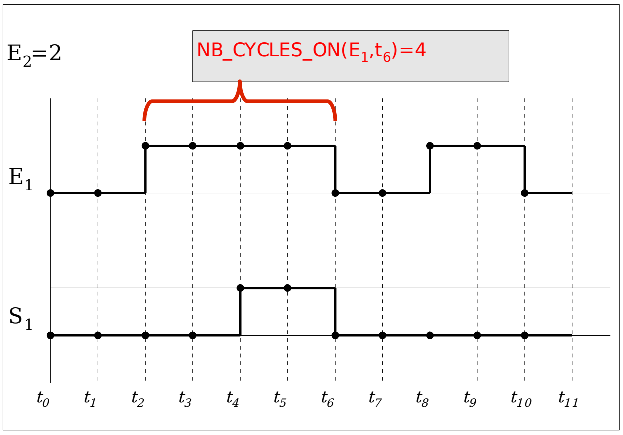

Figure 3: visualising function `NB_CYCLES_ON` on a `CONF` timing diagram

Let us formalise proposition b) with `NB_CYCLES_ON`:

```
LET COND_TIME_REACHED= (NB_CYCLES_ON(E',k) ≥ NB);
```

We may now express conditions 1, 2 and 3 formally:

```
LET COND_INPUT_OFF = ¬SUBCOND_INPUT_ON ;

LET COND_INPUT_CONFIRMED =
    SUBCOND_INPUT_ON ∧ COND_TIME_REACHED ;

LET COND_INPUT_NOT_YET_CONFIRMED =
    SUBCOND_INPUT_ON ∧ ¬COND_TIME_REACHED ;
```

To make sure this specification is complete and unambiguous, we may also ask the tool to prove that:

- no execution condition is forgotten:

```
Always COND0 :
       COND_INPUT_OFF
     ∨ COND_INPUT_NOT_YET_CONFIRMED
     ∨ COND_INPUT_CONFIRMED ;
```

- conditions 1, 2 and 3 are exclusive:

```
Always DIS1 :
  ¬(   COND_INPUT_OFF
     ∧ COND_INPUT_NOT_YET_CONFIRMED) ;

Always DIS2 :
  ¬(COND_INPUT_OFF ∧ COND_INPUT_CONFIRMED) ;

Always DIS3 :
  ¬(   COND_INPUT_NOT_YET_CONFIRMED
     ∧ COND_INPUT_CONFIRMED) ;
```

Finally, let us express the postconditions for the `conf_call` wrapper:

```
FONCTION conf_call

Post RES_CONF1 :
     COND_INPUT_OFF ⇒ S.[.(k)]=0 ;

Post RES_CONF2 :
     COND_INPUT_NOT_YET_CONFIRMED ⇒ S.[.(k)]=0 ;

Post RES_CONF3 :
     COND_INPUT_CONFIRMED ⇒ S.[.(k)]=1 ;
```

Recalling the `E2` constant parameter for the `CONF` operator is a non-negative integer, we also define a precondition on the wrapper function:

```
FONCTION conf_call
Pre COND_VAR1 : NB≥0 ;
```

Formal Proof

Our goal is now to prove the three postconditions on `conf_call`. We thus run the CAVEAT tool on the formal specification and on the following implementation of the `CONF` macro-function:

```
#define CONF(E1, E2, S1)\
{\
     static int count;\
     \
     if (E1) count++;\
     else count=0;\
     \
     S1 = (count>E2);\
}\
```

`RES_CONF1` is proved automatically by the tool, since the behaviour of the code does not depend on history when `E[k]` is off.

On the other hand, RES_CONF2 and RES_CONF3 cannot be proved directly, unless some more properties are introduced. For both, we need to tell the tool about an implicit relationship between the value of the static count variable and the history of inputs: this variable stores the number of clock ticks for which E has been on. This fact can be expressed via the NB_CYCLES_ON lambda-function that we have defined earlier:

```
FONCTION conf_call
Pre COND_SET1 : count=NB_CYCLES_ON(E, k) ;
```

In turn, this precondition for the conf_call function must be proved to be respected in its synchronous execution model. The way to do it is to make CAVEAT check this property results from a loop invariant of the the conf_iterate main function. We therefore claim the following invariant for the main loop:

```
FONCTION conf_iterate
Inv 1 I1_SET1: count=NB_CYCLES_ON(E', i) ;
```

This property holds because of an implementation hypothesis : all RAM variables have been initialised to zero before time k=0. Consequently, we need to add the following top-level precondition:

```
FONCTION conf_iterate
Pre H_INIT_RAM_0 : count=0 ;
```

For the tool to be able to prove the invariance of I1_SET1 in conf_iterate, it also needs to know about an extra postcondition of conf_call:

```
FONCTION conf_call
Post ALGO_SET1: count=NB_CYCLES_ON(E, k+1);
```

Let us recall the complete formal specification :

```
-- Definition of the number of clock ticks
-- for which the input has been on
Const NB_CYCLES_ON ∊ Tab&int -> int -> int =
    lambda input ∊ Tab&int, n ∊ int.
        if (n ≤ 0) then 0
        else(
            if (input.[.(n-1)]=0) then 0
            else (1 + NB_CYCLES_ON(input, n-1))
            );


FONCTION conf_call

-- Definition of all execution conditions
-- for the CONF operator

LETSUBCOND_INPUT_ON = (E'.[.(k)] ≠  0) ;
LET COND_TIME_REACHED= (NB_CYCLES_ON(E',k) ≥ NB);
LET COND_INPUT_OFF = ¬SUBCOND_INPUT_ON ;

LET COND_INPUT_CONFIRMED =
    SUBCOND_INPUT_ON ∧ COND_TIME_REACHED ;

LET COND_INPUT_NOT_YET_CONFIRMED =
    SUBCOND_INPUT_ON ∧ ¬COND_TIME_REACHED ;


-- Completeness and independence
-- of execution conditions

Always COND0 :
       COND_INPUT_OFF
     ∨ COND_INPUT_NOT_YET_CONFIRMED
     ∨ COND_INPUT_CONFIRMED ;

Always DIS1 :
  ¬(   COND_INPUT_OFF
     ∧ COND_INPUT_NOT_YET_CONFIRMED) ;

Always DIS2 :
  ¬(COND_INPUT_OFF ∧ COND_INPUT_CONFIRMED) ;

Always DIS3 :
  ¬(   COND_INPUT_NOT_YET_CONFIRMED
     ∧ COND_INPUT_CONFIRMED) ;


-- Preconditions for this operator
Pre COND_VAR1 : NB≥0;


-- Postconditions to be proved
Post RES_CONF1 :
     COND_INPUT_OFF ⇒ S.[.(k)]=0 ;

Post RES_CONF2 :
     COND_INPUT_NOT_YET_CONFIRMED ⇒ S.[.(k)]=0 ;

Post RES_CONF3 :
     COND_INPUT_CONFIRMED ⇒ S.[.(k)]=1 ;


FONCTION conf_iterate

-- User hypotheses

Pre H_INIT_RAM_0 : count=0 ;
Pre H_POSITIVE_CONF_DURATION : NB≥0 ;
```

We also recall the additional properties:

```
FONCTION conf_call

Pre COND_VAR2 : k≥0 ;
Pre COND_SET1 : count=NB_CYCLES_ON(E, k) ;
Post ALGO_SET1: count=NB_CYCLES_ON(E, k+1) ;

FONCTION conf_iterate

Inv 1 I1_VAR1 : NB'≥0 ;
Inv 1 I1_VAR2 : i≥0 ;
Inv 1 I1_SET1 : count=NB_CYCLES_ON(E', i) ;
```

All these properties are proved with the CAVEAT tool.

Figure 4 shows the dependences between postconditions, preconditions and invariants in the proof process. The arrows mean "*can be proved thanks to*".

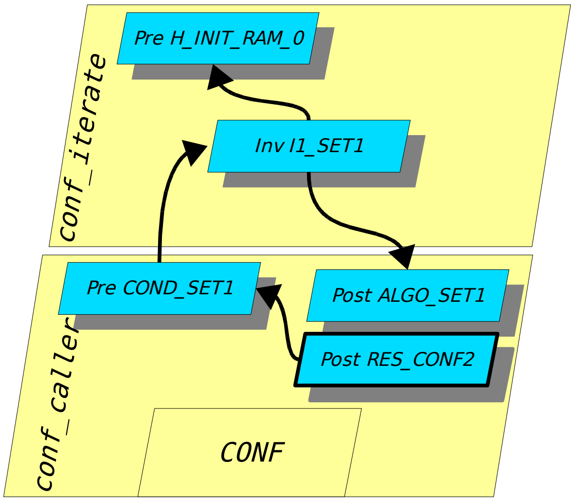

Figure 4: Dependences between properties

3.2 Results obtained

Among the different types of basic operators defined in §1.2, the CAVEAT tools allows:

- automatic proofs on pure boolean/integer operators;
- semi-automatic proofs on pure temporal operators, using the same methodology as with the `CONF` case study;
- semi-automatic proofs on various combinations thereof.

3.3 Theoretical background

The CAVEAT tool is based on Hoare logic, which uses Hoare triples to reason about program correctness. The triple `{P} code {Q}`, where `P` and `Q` are predicates, means: "*if* `P` *is true before execution of* `code`*, then* `Q` *will be true when execution of* `code` *is finished*".

Figure 5 recalls the dependences between properties of figure 4, abstracting away the details of the `CONF` case study. It shows the way we prove multi-cycle properties on temporal operators.

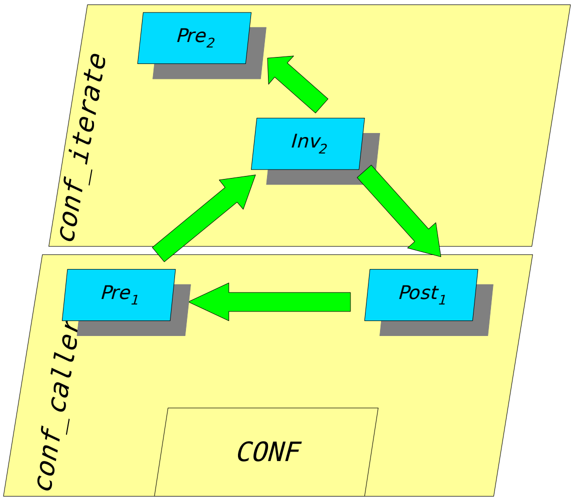

Figure 5: Dependences between properties

When there is a loop as here, the proof scheme may be invalid: if `P` relies on `Q` and `Q` relies (directly or not) on `P`, no valid proof has been found. In fact, there is also a while-loop inside the code of function `conf_iterate`, and proving the partial correctness of loops with invariants is very similar to proving the correctness of recursive programs via mathematical induction.

The tool is able to prove the following Hoare triple:

```
    {Pre1} code of conf_call {Post1}
```

That means the validity of `Post1` relies on the validity of `Pre1`, which gives the conditions of use of the function `conf_call`.

The loop invariant of `conf_iterate` ensures that the precondition of `conf_call` holds for all iterations of the loop:

```
        {Inv2} jump to entry point
           of conf_call {Pre1}
```

So, the respect of the conditions of use of the function `conf_call` relies on the correctness of the loop invariant.

Like in a proof by induction on the number of loop iterations, the proof of the invariant is decomposed into two proof obligations (PO):

1. one PO to show the invariant is stated;
2. one PO to show the invariant is maintained.

The precondition of conf_iterate guarantees the truth of Inv_2 the first time execution reaches the loop body:

```
{Pre2} code from the beginning of
conf_iterate to the while-loop {Inv2}
```

That is the base case. Then, the invariant has to be maintained by the execution of the loop body. The inductive case {Inv2} loop body {Inv2} shows that if Inv_2 holds after k-1 iterations, then it also holds after k iterations. It can be decomposed in three Hoare triples:

1. {Inv2} jump to the entry point of conf_call {Pre1}
2. {Pre1} code of conf_call {Post1}
3. {Post1} return from conf_call ; increment loop counter {Inv2}

So, the dependence loop of Figure 5 is, in our case, sound. The proof of the postconditions of conf_call is valid for all values $k \geq 0$ of the underlying synchronous clock.

4. Limitations and complementary approaches

Typical control/command programs use a lot of numerical operators. Unfortunately, CAVEAT provides very limited support for floating-point computations. It cannot be used so far to derive sound proofs of properties of numerical operators. That is the reason why we are considering improving the tool to fully support floating-point numbers, with the semantics of mathematical real numbers only (precision issues will not be addressed). This will make it possible to analyse C functions performing floating-point computations, provided some complementary numerical precision analysis is carried out. This analysis will be performed with the FLUCTUAT ([7]) abstract interpretation ([4]) based static analyser, a dedicated tool for studying the propagation of rounding errors in floating-point computations.

On the other hand, most operators can only be accurate with respect well-defined limited input ranges. For instance, multiplicative operators may yield floating-point overflows when used on incorrect inputs. As a consequence, we need to prove the complete program free from such run-time errors, as well as compute maximal input ranges for all operators, for our proof-based approach to be sound. The Astrée ([5]) abstract interpretation based static analyser has been shown to meet this ambitious objective on real-world control/command programs ([6]).

Moreover, all these analysis techniques work on C source code. They cannot replace testing completely, unless the subsequent compilation process is also proved correct. We are therefore working on a translation validation technology: prove that the source and the compiled program have the same semantics ([8]).

Finally, it has to be proved that every instance of each temporal operator is actually used in a synchronous execution model, ie that the program can always be run completely on every tick of the synchronous clock. We thus use another static analysis tool to compute a safe and precise upper bound of its worst-case execution time ([9]).

5. Conclusion and future work

The experiments described in this paper show that the CAVEAT static analyser can be used to prove multi-cycle properties on basic operators of real-world embedded control/command programs. Moreover, this work has been an opportunity to sketch a proof methodology that makes it possible for non-expert engineers from industry to perform this formal verification activity in a rather straightforward way.

On the other hand, some work remains to be done for this proof technique to yield sound results on all types of operators: we need several complementary static analysis techniques to join forces. However, most necessary technologies are already available, and the rest will soon be. We will then be in a position to replace all (unsound) testing with (sound) static analysis for most basic operators.

6. References

1 Hoare, C.A.R. An axiomatic basis for computer programming. Commun. ACM 12(10), pages 576-580, 1969.

2 Patrick Baudin, Anne Pacalet, Jacques Raguideau, Dominique Schoen, Nicky Williams. Caveat : A tool

for software validation. In Proceedings of the Int. Conference on Dependable Systems and Network (DSN), IEEE Computer Society, pages 537-537, 2002.

3 Stéphane Duprat, Jean Souyris, Denis Favre-Felix. Formal verification workbench for airbus avionics software. In Proceedings of ERTS 2006, SIA, 2006.

4 Patrick Cousot & Radhia Cousot. Basic Concepts of Abstract Interpretation. In Building the Information Society, R. Jacquard (Ed.), Kluwer Academic Publishers, pp. 359--366, 2004.

5 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival. The ASTREE analyser. In ESOP 2005 -- The European Symposium on Programming, M. Sagiv (editor), Lecture Notes in Computer Science 3444, pp. 21--30, 2--10 April 2005, Edinburgh, (c) Springer.

6 David Delmas and Jean Souyris. Astrée: from Research to Industry. In Static Analysis, SAS 2007, Lecture Notes in Computer Science. Springer-Verlag, 2007.

7 Eric Goubault, Matthieu Martel, and Sylvie Putot, Static Analysis-Based Validation of Floating-Point Computations, Proceedings of Dagstuhl Seminar Numerical Software with Result Verification 2003, LNCS volume 2991, pp 306-313.

8 Xavier Rival. Symbolic Transfer Function-based Approaches to Certified Compilation. In 31st Symposium on Principles of Programming Languages (POPL'2004), Venice, Jan. 2004 ACM.

9 Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the worst-case execution time of an avionics program by abstract interpretation. In Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, pages 21-24, 2005.