An experimental Study using ACSL and Frama-C to formulate and verify Low-Level Requirements from a DO-178C compliant Avionics Project

Frank Dordowsky
ESG Elektroniksystem- und Logistik GmbH, 82256 Fürstenfeldbruck, Germany
frank.dordowsky@esg.de

Safety critical avionics software is a natural application area for formal verification. This is reflected in the formal method's inclusion into the certification guideline DO-178C and its formal methods supplement DO-333. Airbus and Dassault-Aviation, for example, have conducted studies in using formal verification. A large German national research project, Verisoft XT, also examined the application of formal methods in the avionics domain.

However, formal methods are not yet mainstream, and it is questionable if formal verification, especially formal deduction, can be integrated into the software development processes of a resource constrained small or medium enterprise (SME). ESG, a Munich based medium sized company, has conducted a small experimental study on the application of formal verification on a small portion of a real avionics project. The low level specification of a software function was formalized with ACSL, and the corresponding source code was partially verified using Frama-C and the WP plugin, with Alt-Ergo as automated prover.

We established a couple of criteria which a method should meet to be fit for purpose for industrial use in SME, and evaluated these criteria with the experience gathered by using ACSL with Frama-C on a real world example. The paper reports on the results of this study but also highlights some issues regarding the method in general which, in our view, will typically arise when using the method in the domain of embedded real-time programming.

1 Introduction

Recent advances in automated theorem provers have brought formal methods from purely academic exercises close to industrial use. Airborne software has early been recognized as a suitable candidate for the application of formal methods [14, 21]. Airbus, for example, has examined the formal method Caveat to replace unit tests [15]. Since Airbus is a trend setter in the avionics industry at least in Europe, one can expect the obligation to use formal methods for the development of highly safety critical airborne software in future. Moreover, standard organizations start to incorporate formal methods into their regulations as one can already see with the formal methods supplement DO-333 [12] of the new version of the avionics software certification guidelines, DO-178C [11]. Another well-known example is the Common Criteria for Information Technology Security Evaluation that demand the use of formal methods for the highest Evaluation Assurance Levels EAL 6 and EAL 7 [9]. It is probably safe to assume that more and more development standards will mandate the application of formal methods in the future. This will force also small and medium enterprises (SME) that develop safety critical software to consider the adoption of formal methods in the long run.

ESG as a medium sized company that develops airborne system solutions has therefore launched a small experimental study to examine if formal verification can be integrated into its development pro-

C. Dubois, P. Masci, D. Méry (Eds): 2nd International Workshop
on Formal Integrated Development Environment (F-IDE 2015)
EPTCS 187, 2015, pp. 28–41, doi:10.4204/EPTCS.187.3

cesses for airborne software, and to identify the prerequisites for an adoption of formal methods in future DO-178C compliant software projects.

In Germany, the Verisoft XT project [6] examined the application of formal methods in an industrial context. ESG participated in a work package of that project that examined the application of formal methods in the avionics domain [7]. This experimental study is a continuation of ESG's work in that project.

ACSL is an acronym for "ANSI C Specification Language" provided for the open source tool platform Frama-C [2, 10]. It is a behavioral specification language that can express properties of C source code including pre-conditions and post-conditions of C functions using first order logic. The ACSL specifications are provided as annotations to the source code. The WP plug-in of the framework allows a deductive verification of the source code against the formal annotations in ACSL [3].

DO-178C defines *Low-Level Requirements (LLR)* as software requirements from which source code can be directly implemented without further information. In the software projects we have been involved in, these requirements were natural language annotations to the subroutines or function declarations provided by the interfaces of the software modules. It is therefore natural to consider ACSL as a candidate to formally express the low-level requirements.

Several so-called DO-178C *objectives* are associated with low-level requirements that must be fulfilled for acceptance of the software by certification authorities. One such objective is the verifiability of low-level requirements. This objective is indeed achieved when a formal notation is used. Another objective is the compliance of the source code with the low-level requirements, which is usually shown by code review. These time consuming reviews may be replaced by automated formal verification. Blasum et al [7] discuss these and other DO-178C objectives for the formal methods of the Verisoft XT project.

The major goal of the study was to check if formal notations can be used in a real project to formally express real world low level requirements, i.e. to see if a certain formal method is fit for use in industrial practice in general and for ESG in particular. A secondary goal was to examine if available tools can verify such annotated code and even be able to find bugs not yet discovered by testing, which would prove one of the claimed advantages of formal methods over testing.

The first step was to establish criteria which a formal notation and its supporting tools should meet in order to be ready for industrial use in an SME setting. In a second step, we selected a function from a real avionics project and formalized the natural language specifications of the associated C functions into ACSL specifications. It was in this step that we encountered the first obstacles although the selected function was rather trivial.

These obstacles are, in our view, quite typical for embedded real-time programming. The problems were solved with support of Frama-C experts via the Frama-C mailing list. However, the solutions are not fully satisfactory which may be inherent to the approach, as it will be discussed later.

In a next step, we attempted to verify the source code against the ACSL specification while instrumenting the source code with assertions in order to guide the prover. Not all verification attempts were successful. We were only able to explain some of the failed attempts before running out of time[1]. In a final step, we compared our experience in using tool and method against the formerly established criteria.

[1] Ironically, the failed proof attempts that we were not able to explain were those where the prover timed out.

2 Industrial Fit for Purpose

The main goal of the study was to check if formal specification and verification is already suitable to be used in an industrial rather than academic context, in a small to medium sized enterprise. In order to decide on this question, a number of criteria must be fulfilled:

- Notation and method should be within the normal range of experience of an average software engineer, i.e. it should be usable for engineers without a PhD in formal logic.
- The complexity of the formal language should have the same level as a programming language such as Ada or C. The learning curve should be moderate.
- Training courses should be available.
- Information sources such as text books, guidelines, etc. should be available.
- The tools used for specification and verification should be mature and stable.
- The tool should provide feedback to the user in case a proof fails, ideally by showing the places where the proof fails together with a counter example.

For avionics projects that are going to be developed in accordance with DO-178C, additional criteria derived from DO-333 objectives such as method soundness, coverage criteria or tool qualification have to be considered. Such considerations were not in the scope of this study. For a discussion of these criteria for VCC [20], a tool similar to Frama-C, see [7].

3 Frama-C, WP Plugin and Alt-Ergo

Avionics software is usually real-time, embedded, i.e. hardware related, resource constrained software that often implements complex arithmetic algorithms. One will find type casts, especially between integer and hardware addresses, and low level pointer handling, as well as floating point calculations.

Embedded projects often do not use an operating system so that the developer must directly access and manipulate the hardware.

The strict real-time requirements sometimes lead to the application of the most efficient programming constructs that may be difficult to verify with formal methods.

A formal notation and its supporting tools must address these constraints. This is the case with the open source tool platform Frama-C together with its WP plugin. Frama-C is organized into a plug-in architecture with a common kernel that allows the plug-ins to interact with each other using ACSL as lingua franca.

The WP plugin was selected because it well supports the aforementioned properties of avionics software. WP uses memory and arithmetic models in order to model memory management of dynamic structures (pointers) and properties of integer and real variables [3].

4 Function Selection

The examples in this study have been taken from the control software of a sensor that ESG has developed as a central component of a pilot assistance system for a military helicopter. The sensor control software has been developed in accordance with DO-178B, level D and was accepted by the German military certification authority WTD 61/ML in 2014.

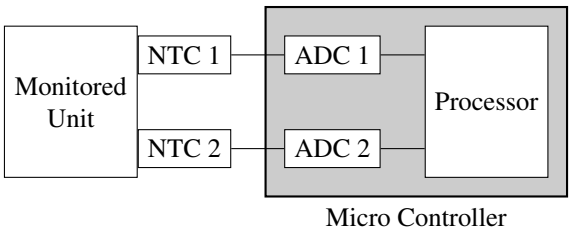

Figure 1: Temperature Monitoring Hardware

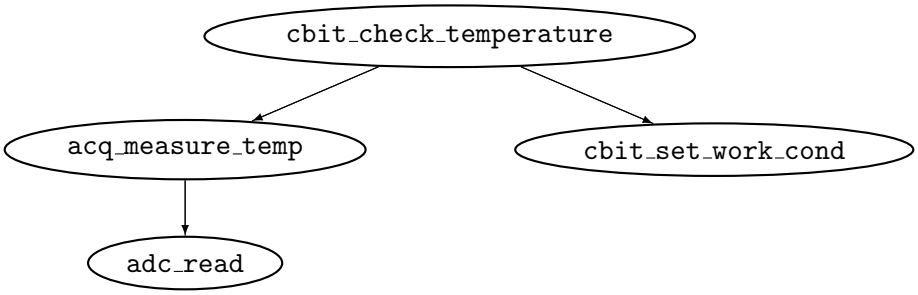

Figure 2: Temperature Monitoring Call Hierarchy

Monitoring the environmental conditions, especially the temperature, is important to the correct operation of the sensor. Therefore, two temperature sensors (NTC1 and NTC2 in figure 1) are installed within the equipment for redundant temperature measurements.

The temperature monitoring function is part of the sensor's control program that is running on the processor shown in figure 1. It calculates the mean temperature of both readings, as long as they do not differ more than a certain amount (declared as positive integer `TEMP_FAIL`). It accepts `MAX_TEMP_ERR_CNT` consecutive differences larger than `TEMP_FAIL` before returning the error `EC_TEMP`.

Figure 2 shows the temperature monitoring call hierarchy: the function `cbit_check_temperature` is executed every 500 ms. This function calls `acq_measure_temp` that returns the temperature readings of the two temperature sensors. To do so, the latter function calls `adc_read` to obtain the voltage levels from the Analog-Digital converters (ADCs, see Fig. 1) connected to the temperature sensors.

If the temperature reading is outside operational limits, this fact is reported as a *working condition* in a module state variable that is set by the function `cbit_set_working_cond`.

This set of functions has been selected for the following reasons:

- The functions are rather simple but already required some of the major concepts of ACSL for their formalization.
- The selected functions have been taken from a real project, it is real code and real specifications with all the flaws that typically occur in a time constrained industrial project.
- The functions cover a standard problem (monitoring a certain value obtained from an analogue-to-digital (AD) converter) so it does not reveal intellectual property. They are well suited for public discussion.
- The example could well be isolated: the source files contain only the selected C functions without failing compilation and analysis.
- The whole set of functions encompasses purely algorithmic as well as hardware related functions.

5 Formalization and Verification of LLR

The analysis approach is the same for all functions: at first, the original specification of the C function is formalized using ACSL behavior specifications. In a subsequent step, verification of the source code is attempted to show by proof that the implementation follows the specification. This is actually an iterative approach which often needs adding instrumentation (assertions, loop invariants) to the source code.

To illustrate the approach, we use the most simple function `cbit_set_work_cond` that simply writes to a module variable, i.e. a persistent variable only known within the module `cbit`. This variable is used in other parts of the program but is read and write protected by setter and getter functions.

The update of the state variable is guarded – the function takes a new value of the working condition as a parameter `new_cond` and writes that to the module variable `WorkCondition` unless this module variable has been set previously. The only exception is the special parameter value `NCD_NO_COMMAND` that can overwrite all settings. A natural language formulation of the low level requirements would read as follows:

1. If the value of the input parameter `new_cond` is equal to `NCD_NO_COMMAND`, then the module variable `WorkCondition` is set to this value `NCD_NO_COMMAND`.
2. If the current content of the module variable `WorkCondition` is equal to `NCD_IDLE_EMIT`, the module variable `WorkCondition` is set to the value of the input parameter `new_cond`.
3. If the value of the module variable `WorkCondition` is not equal to `NCD_IDLE_EMIT` and the value of the input parameter `new_cond` is not equal to `NCD_NO_COMMAND`, then the current value of `WorkCondition` is not changed.

The attempt to formalize these LLR into ACSL encounters the first obstacle: the internal module variable – a static variable of file scope – is not visible in the header file where the formalized specification is located. The solution was to introduce a ghost variable that represents the internal data, which in effect is uncovering the internal variable, i.e. turning parts of the module inside out. The ghost variable for the internal state is located in the header file:

```
//@ ghost uint16_t gWorkCond = NCD_IDLE_EMIT;
```

The behavioral specification, i.e. the translation of the LLR above, is now straight forward:

```
/*@
  @ behavior NoCommand:
  @  assumes new_cond == NCD_NO_COMMAND;
  @  ensures gWorkCond == new_cond;
  @ behavior ModifyWC:
  @  assumes gWorkCond == NCD_IDLE_EMIT;
  @  assumes new_cond != NCD_NO_COMMAND;
  @  ensures gWorkCond == new_cond;
  @ behavior KeepWC:
  @  assumes ((gWorkCond != NCD_IDLE_EMIT) &&
  @           (new_cond != NCD_NO_COMMAND));
  @  ensures gWorkCond == \old(gWorkCond);
  @ complete behaviors;
  @ disjoint behaviors;
  @*/
void cbit_set_work_cond(uint16_t new_cond);
```

The three behavior specifications above directly correspond to the natural language requirements so that they could be replaced by the formalized requirements. The `behavior` clause can be named as shown in the example which facilitates traceability to higher level requirements, an objective that is strongly emphasized in aviation standards. The `assumes` clauses represent the conditional part of the natural language requirements and are evaluated at the beginning of the execution of the function. Moreover, with `complete` and `disjoint` clauses one can automatically verify completeness and consistency of the formal specification.

Note that `assigns` clauses have been omitted from the specification above although this is discouraged by the ACSL reference manual ([4], section 2.3.5). However, adding `assigns` clauses generates verification conditions that cannot be proved. This is due to the fact that the function modifies a state variable of file scope (`WorkCondition`) which cannot be listed in the `assigns` clause because it is not visible there. Several solutions for this problem are under discussion on the tool's mailing list at this time of writing.

For the verification, the ghost state variable must be aligned with the internal module variable using assertions, as shown below:

```
void cbit_set_work_cond(uint16_t new_cond)
{
  //@ assert gWorkCond == WorkCondition;
    if ((WorkCondition == NCD_IDLE_EMIT) || (new_cond == NCD_NO_COMMAND))
    {
        WorkCondition = new_cond;
        //@ ghost gWorkCond = WorkCondition;
    }
}
```

The assertion in line 3 is necessary to inform the prover that the internal state is always equal to the ghost state. It is not possible to prove it.

The WP plugin generates a verification condition for every assertion. The verification conditions are similar to test cases in traditional verification so that a verification condition that cannot be proved is identical to a test case that has failed. Therefore, justification is needed for the assertion in line 3 to fulfill objective FM2 of table FM.C-7 of DO-333 that demands that formal analysis results are correct and discrepancies are explained.

Representing internal state with a ghost variable is not optimal as indicated above. The function `cbit_set_work_cond` and its counterpart `cbit_get_work_cond` would have been better specified by using algebraic specification techniques in the way it is shown in [8] for the stack example. The approach described there transforms the axioms into additional, ACSL annotated C code. This essentially means that C code is used as low level requirements which is not acceptable[2]. The inclusion of algebraic specification techniques into ACSL would probably be the most elegant solution for this problem.

6 Obstacles

The previous section has already discussed one of the obstacles we encountered when applying a formal notation to the specification of LLRs, which is the need to address internal state. There were additional issues which will be described in the following subsections.

[2]EASA is concerned of even using pseudocode as LLR [1]

6.1 Specifying Behavior across multiple Invocations

With ACSL, one can only specify a single execution of a C function. However, the result of a function call sometimes depends on prior executions of this or even other functions. For example, the function `cbit_check_temperature` tolerates two consecutive discrepancies in the readings of the two temperature sensors before indicating an error condition. This is usually implemented with internal state variables (`err_cnt` in our example), which is a static variable in the definition of the function, not visible to the outside. This again cannot be addressed in the specification located in the header file. As a solution, a ghost variable `gerrcnt` has been introduced into the specification, but as in the previous section, the ghost variable must be updated accordingly using assertions within the body of the function.

This solution is also not optimal because it is rather close to the implementation and discloses internals of the implementation.

A better way in this example is to specify the intended behavior with a state automaton. The Frama-C plugin Aoraï [18] can be used for this purpose. It translates automaton specifications formulated in YA or LTL into ACSL annotations (and additional annotated C code) that can subsequently be verified with the WP plugin. However, one needs to learn an additional specification language to use this approach. We were not able to accomplish this in the time available for the study.

During verification attempt we detected an inaccuracy in the natural language specification concerning the number of consecutive temperature discrepancies. The implementation is correct, but the specification text is misleading.

6.2 Input Values obtained from Calls to Subroutines

As shown in Fig. 2, function `cbit_check_temperature` calls `acq_temp_measure` to obtain temperature readings as input, and `acq_temp_measure` calls `adc_read` to get the voltage levels from the ADCs as input. This pattern of calling subroutines to obtain input was used a lot in the real world example. Again, the return values of subroutines are not visible at specification level.

Ghost variables that represent return values of subroutines were introduced as a work-around. These ghost variables (`T1` and `T2`) must be aligned with the values of the subroutine call:

```
acq_measure_temp(&temp1, &temp2);
//@ ghost T1 = temp1;
//@ ghost T2 = temp2;
```

Although `T1` and `T2` act as input, their values are obtained during execution and nothing is known at pre-state. It is therefore not possible to express pre-conditions in the form of `assumes` clauses with these input variables.

We introduced predicates to replace the `assumes` clauses, as shown in the following code snippet:

```
@ predicate A_TempReadFailTrans
@    (integer t1, integer t2, integer cnt) =
@    (\abs(t1 - t2) > TEMP_FAIL) && ( cnt <= 2);
@
@ predicate A_TempReadFailPerm
@    (integer t1, integer t2, integer cnt) =
@    (\abs(t1 - t2) > TEMP_FAIL) && ( cnt > 2);
```

These predicates are now used as replacement for the `assumes` clause as shown in the following specification of the behaviour in case of discrepancies of temperature readings:

```
@ ...
@ behavior TempReadFailTrans:
@  ensures A_TempReadFailTrans(T1,T2,\at(gerrcnt,Pre)) ==>
@            (gModuleTemp == \old(gModuleTemp));
@  ensures A_TempReadFailTrans(T1,T2,\at(gerrcnt,Pre)) ==>
@            (gerrcnt == \old(gerrcnt) + 1);
@  ensures  A_TempReadFailTrans(T1,T2,\at(gerrcnt,Pre)) ==>
@             (\result == EC_NO_ERROR);
@
@ behavior TempReadFailPerm:
@  ensures A_TempReadFailPerm(T1,T2,\at(gerrcnt,Pre)) ==>
@            (gModuleTemp == 0);
@  ensures A_TempReadFailPerm(T1,T2,\at(gerrcnt,Pre)) ==>
@            (gerrcnt == \old(gerrcnt));
@  ensures A_TempReadFailPerm(T1,T2,\at(gerrcnt,Pre)) ==>
@            (\result == EC_TEMP);
@
@ behavior TempOK:
@  ensures A_TempReadOK(T1,T2) ==>
@            (gModuleTemp == temp_average(T1,T2));
@  ensures A_TempReadOK(T1,T2) ==> (\result == EC_NO_ERROR);
@  ensures A_TempReadOK(T1,T2) ==> (gerrcnt == 0);
@  ...
```

As one can see in the example above, we used a naming convention to indicate that certain predicates replace `assumes` clauses. However, this approach is not very elegant and has additional disadvantages that are discussed in the next subsection.

The verification attempt revealed another weakness in the natural language specification – it is ambiguous with respect to the calculated module temperature in case of permanent temperature discrepancy. Here again is the implementation correct but the specification text is misleading.

6.3 Behavior Specifications

Without `assumes` clauses it is no longer possible to use `complete` and `disjoint` clauses, i.e. it is not possible anymore to use the tool to simply prove completeness and consistency (i.e. "disjointedness") of the specification. As a work-around, one can formulate completeness and consistency properties with `ensures` clauses. Completeness is formulated as follows:

```
@ ensures A_TempReadFailTrans(T1,T2,\at(gerrcnt,Pre)) ||
@    A_TempReadFailPerm(T1,T2,\at(gerrcnt,Pre)) ||
@    A_TempReadOK(T1,T2);
```

Consistency is expressed as

```
@ ensures ! (( A_TempReadFailTrans(T1,T2,\at(gerrcnt,Pre)) &&
@     A_TempReadFailPerm(T1,T2,\at(gerrcnt,Pre))) ||
@    (A_TempReadFailTrans(T1,T2,\at(gerrcnt,Pre)) &&
@     A_TempReadOK(T1,T2)) ||
@    (A_TempReadFailPerm(T1,T2,\at(gerrcnt,Pre)) &&
@     A_TempReadOK(T1,T2)));
```

For consistency specifications with many predicates this easily becomes complicated and nontransparent.

When taking this problem and the discussion of the previous subsection into account, it is better to avoid input values obtained from subroutine calls and to add this as a rule to the design standards. In our example, the calling function of `cbit_check_temperature` would call `adc_read`, `acq_measure_temp`, `cbit_check_temperature`, and `cbit_set_work_cond` in a row. This would also flatten the call hierarchy which has additional advantages as explained in the next subsection.

There is a subtle flaw in the original natural language specification of `cbit_check_temperature` as well as in its formal counterpart. The function sets the module variable `WorkCondition` but this is not specified for all cases. It is implicitly assumed that it is not modified in these cases (which is correct), but formally an implementation is free to modify the working condition by any value. Such omissions can become critical if specifications are part of interface contracts. The method and the notation itself cannot prevent such specification errors. The problem is addressed by DO-333 in objective FM.5-8 of table FM.C-7 (Verification Coverage of Software Structure is achieved). This objective refers to section FM.6.7.1.3 (Completeness of the Set of Requirements) that states that for all outputs, the required input conditions must have been specified. The output that must be considered here is the internal module variable `WorkCondition`.

6.4 Specification Proliferation

The original description of `cbit_check_temperature` states that, if the module temperature is outside the range between `TEMP_MIN` and `TEMP_MAX`, the module variable `WorkCondition` shall be set to `NCD_TEMP_LOW` or `NCD_TEMP_HIGH` respectively by calling `cbit_set_work_cond` (see section 5).

This was formalized as follows:

```
@  ensures (A_TempReadOK(T1,T2) && TempTooCold(T1,T2)) ==>
@              (gWorkCond == NCD_TEMP_LOW);
```

This cannot be proved because `cbit_set_work_cond` only overwrites the working condition if the current value of the working condition is equal to `NCD_IDLE_EMIT` (see section 5). The corrected specification (which can be proved) is

```
@  ensures (A_TempReadOK(T1,T2) && TempTooCold(T1,T2) &&
@           gWorkCond == NCD_IDLE_EMIT) ==>
@                (gWorkCond == NCD_TEMP_LOW);
```

This specification repeats parts of the specification of `cbit_set_work_cond`. It is an example of *specification proliferation* where higher level functions partially repeat and aggregate those of lower level functions. It leads to redundancy in the specifications which in turn leads to a higher maintenance effort, and it counteracts to a certain extent the well established information hiding heuristic.

One option to alleviate the specification proliferation is keeping the call hierarchy as flat as possible (e.g. by avoiding input values from subroutine calls). However, this might not be possible for large and complex programs. Another option is to move all algorithmic complexity to the lowest levels of the call hierarchy and use formal specifications only at these levels. The higher level functions should have simple logic (at best, only call sequences) that can be easily verified by code review.

Another alternative is to introduce predicates for the expressions shared in the specifications. A similar approach using "specification macros" has been used in the Verisoft project for recurring annotations [5]. However, this technique cannot be demonstrated with the small example used in this study.

6.5 WP Plugin does not support Math Functions

The function acq_measure_temp converts the digitized voltage readings of the temperature dependent resistors (NTC1 and NTC2 in Fig. 1) into temperature values. In our first specification attempt we formulated the post-condition on the output variables temp1 and temp2 with a logical function that used the exponential function exp as shown in the code extract below:

```
...
//@ ghost uint16_t D1, D2;
...
/*@ logic real R(integer T) =
  @       10000.0 *\exp(3988.0*(1.0/(T+273.15)-1.0/298.15));
  @
  @ logic real U(integer T) = (5.0*R(T))/(R(T)+5360.0);
  @
  @ logic integer D(integer T) = \floor(250.0*U(T)+0.5);
  @*/
...
/*@
  @ ,,,
  @ ensures D(*temp1) >= D1 > D(*temp1 + 1);
  @ ensures D(*temp2) >= D2 > D(*temp2 + 1);
  @ ...
  @*/
void acq_measure_temp(uint16_t* temp1, uint16_t* temp2);
```

The logical function $D(T)$ expresses the relation between the temperature and the digitized voltage reading. Is is defined by the temperature characteristic of the NTCs and the electrical circuit design. The ghost variables D1 and D2 represent the digitized voltage readings returned by function adc_read – we have used here the same approach as in section 6.2. The ensures clauses use the logical function to constrain the values of the output variables temp1 and temp2.

We had to learn that the WP plugin in the version used for the study did not recognize standard math functions. It is possible to extent the WP plugin to incorporate own definitions of these functions. This is not a trivial task and very likely out of scope for SMEs. However, a specification of math functions would be very useful across many formal specification projects. Section 3.2 of the ACSL manual [4] announces a library for logic specifications of math functions which would be very helpful for requirements specifications as intended here.

In a second attempt we defined the logical function $D(T)$ as a piece-wise linear approximation with ghost arrays containing the sampling points. This was accepted by the WP plugin, but the verification attempt failed due to timeouts. We were not able to explain the timeouts in the time available for the study.

Note that the implementation of the function acq_measure_temp does not use math functions but iterates through a look-up table of 100 pre-calculated values.

6.6 Compiler specific Language Extensions

The source code of the real world example uses compiler specific language extensions for easier access to hardware registers. These language extensions are obviously not known in standard C and therefore not in the WP plugin. There are three possible solutions to this problem:

Table 1: Verification Status of Temperature Monitoring Functions

Function	Proof Obligations scheduled	valid	unknown	timed out
`cbit_set_work_cond`	6	5	1	0
`cbit_check_temperature`	22	17	5	0
`acq_measure_temp`	237	221	1	15
`adc_read`	-	-	-	-

1. Ban compiler specifics by coding standard.
2. Define semantics of compiler specifics to make it known to WP plugin. This is very laborious and has not been tried.
3. Only use compiler specifics in very small routines at the lowest layer and review these manually.

The last option is probably the most practical solution, and should be enforced by corresponding design and coding rules.

6.7 Modelling Hardware for Hardware Access Functions

One property of avionics software is that it needs hardware related programming (section 3). Specification of hardware access functions requires modelling of hardware in ACSL. The hardware interacts with the external world which operates independently of the software so that the hardware modifies the content of program variables in a way that is not visible in program statements.

ACSL offers so called *volatile ghost variables* to model side effects such as hardware interaction but this concept was not implemented in the Fluorine version of Frama-C that was used for the study.

The solution to this obstacle is the same as in section 6.6: all hardware access shall be concentrated into small subroutines which is common programming practice anyway. These hardware access functions are formally specified like the other functions, but then reviewed manually for compliance with the LLR instead of using automated proof.

The micro controller that was used in the project provided up to sixteen ADC channels onboard. We modelled the set of ADC channels as a ghost array and used this model to specify and partially verify that the correct ADC channels were used for temperature monitoring.

7 Verification Summary

Table 1 provides an overview over the verification attempt of all four functions considered in this study.

The WP plugin could not be executed on `adc_read` because of the non-standard C statements in the source, see section 6.7. We were able to explain all proof obligations for which a proof attempt failed except for the 15 proof obligations of `acq_measure_temp` that timed out. We were not able to conclude the analysis of the problem due to lack of time. It is not uncommon to use alternative provers (which is supported in Frama-C), but this requires additional effort to understand, install, configure and use these provers.

8 Results

This section matches the experience made in conducting this experiment against the criteria established in section 2.

Familiarity with Notation and Method. We had some exposure to formal methods from past experience with algebraic specifications and with the formal notation Z ([16]) as well as his participation in the research project Verisoft XT, but did not have working knowledge of ACSL and the Frama-C tool suite.

The ACSL notation is quite close to the C programming language syntax, a couple of additional language constructs must be familiarized. However, even the simple example that had been chosen for this study required some more sophisticated concepts such as ghost variables, logic functions, axioms and labels. As a consequence, the specifications are not simple to read for the untrained.

One particular difficulty is to find the correct and most efficient loop invariant, which however is a common problem to formal proof.

It should be noted that the amount of instrumentation in form of assertions and loop invariants is in the same order of magnitude as the size of the source code, or even exceeds it. This is to be expected and in line with DO-178 objectives where all source code must be traceable to requirements.

Complexity of the Formal Language. The language is very close to C syntax and is therefore familiar to C programmers. However, for more sophisticated specification tasks, the available ACSL constructs need a much deeper knowledge. Some specification tasks require learning of additional languages (e.g. YA or LTL for Aoraï).

Training. We are only aware of training sessions at conferences, and of courses and project consultancy that had been offered by the Fraunhofer FOKUS institute. Also the recently founded company TrustinSoft offers consultancy for Frama-C.

Information Sources. A few online resources are available, mostly tutorials and papers. The most efficient source of information is the Frama-C mailing list, accessible via the Frama-C Website [13] where the members of the Frama-C community provide fast and accurate advice.

Tool Maturity. We had difficulties installing the tool set on our Linux distribution (Archlinux 3.14.4-1) and on Cygwin over Microsoft Windows 7. With assistance of the tools mailing list and the forum of the Linux distribution we managed to install a command line version of Frama-C with the WP plugin and the Alt-Ergo proof engine on the Linux system. Although problems early on in the tool's usage can have detrimental effects on an organization's acceptance of the tool, this is considered a minor problem.

The WP plugin should be extended to include all ACSL features and support of mathematical functions.

User Guidance provided by Tool. The tool, even the command line version, highlights the goals that cannot be proved. It is possible to record the verification conditions that the tool generates. However, these are formulated in an intermediate language as input to the prover which is quite different to the C or even ACSL syntax. Analysis of verification conditions requires deep knowledge of automated prover technologies.

We were not able to prove all generated proof obligations and could not explain all discrepancies in the time that was allocated for this study. It may be possible that the Frama-C GUI together with interactive provers provide more debugging support – which we could not test due to the aforementioned installation problems.

9 Conclusion

During analysis several smaller deficiencies were discovered in the low-level requirements. The formalization of natural language requirements into ACSL is an excellent tool to "debug" specifications and to improve their quality. Moreover, the formalization helps to achieve many of the DO-178C objectives related to LLR.

At this time of writing, we recommend to use ACSL and Frama-C only in highly safety critical applications and even there only in areas that need extra scrutiny. And even so, the team must be supported by an experienced consultant for both tool and method who can provide immediate assistance in case specification or verification problems occur. We also suggest to collect and publish a set of specification patterns or best practices similar to those for Z [17, 19].

If formal specification is used, then additional rules must be added to the software design and coding standards in order to facilitate formal verification. Such rules would include the reduction of the call hierarchy, the reduction of internal state where possible and the increase of parameter passing as main method of data flow. The decision to use ACSL for specification and verification will shape the software design and coding practice.

Since formal methods have made their way into aviation standards and guidelines it is to be expected that they will be mandated in certain areas in the future, if not by the certification authorities, then by major aerospace companies like Airbus and Dassault. Since the WP plugin did not fully implement the ACSL language standard at the time of the experiment, we also recommend to observe the further development and to repeat this study in one or two years.

References

[1] (2012): *EASA Certification Memo CM-SWCEH-002 Software Aspects of Certification*. Technical Report EASA CM-SWCEH002 Issue 01 Revision 1, European Aviation Safety Agency.

[2] *ANSI/ISO C Specification Language*. Available at `http://frama-c.com/acsl.html`.

[3] Patrick Baudin, Loic Correnson & Zaynah Dargaye (2013): *WP Plug-in Manual. Version 0.7 for Fluorine-20130601*. CEA LIST.

[4] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy & Virgile Prevosto (2013): *ACSL: ANSI/ISO C Specification Language. Version 1.7*. Technical Report, CEA LIST, Software Reliability Laboratory. Available at `http://frama-c.com/download/acsl.pdf`.

[5] C. Baumann, T. Bormer, H. Blasum & S. Tverdyshev (2011): *Proving Memory Separation in a Microkernel by Code Level Verification*. In: *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pp. 25–32, DOI:10.1109/ISORCW.2011.14.

[6] Christoph Baumann, Bernhard Beckert, Holger Blasum & Thorsten Bormer (2009): *Better Avionics Software Reliability by Code Verification – A Glance at Code Verification Methodology in the Verisoft XT Project*. In: *Embedded World 2009 Conference*, Franzis Verlag, Nuremberg, Germany.

[7] Holger Blasum, Frank Dordowsky, Bruno Langenstein & Andreas Nonnengart (2012): *DO-178C Compliance of Verisoft Formal Methods*. In: *Proceedings of the Embedded Real Time Software and Systems Conference, 1. - 3. February, Toulouse*.

[8] Jochen Burghardt, Jens Gerlach amd Liangliang Gu, Kerstin Hartig, Hans Pohl, Juan Soto & Kim Vollinger (2011): *ACSL By Example. Towards a Verified C Standard Library*. Technical Report, Fraunhofer FIRST.

[9] (2007): *Common Criteria for Information Technology Security Evaluation. Part 3: Security Assurance Components*.

[10] Loïc Correnson, Pascal Cuoq, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles & Boris Yakobowski (2013): *Frama-C User Manual. Release Fluorine-20130601*. Available at `http://frama-c.com/download/frama-c-user-manual.pdf`.

[11] (2011): *RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification*.

[12] (2011): *RTCA DO-333 Formal Methods Supplement to DO-178C and DO-278A*.

[13] *Frama-C Software Analyzers*. Available at `http://frama-c.com/`.

[14] John Rushby (1993): *Formal Methods and the Certification of Critical Systems*. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA.

[15] Jean Souyris, Virginie Wiels, David Delmas & Hervé Delseny (2009): *Formal Verification of Avionics Software Products*. In Ana Cavalcanti & Dennis Dams, editors: *FM 2009: Formal Methods*, *Lecture Notes in Computer Science* 5850, Springer Berlin / Heidelberg, pp. 532–546, DOI:10.1007/978-3-642-05089-3_34.

[16] J. M. Spivey (1998): *The Z Notation: A Reference Manual*, 2nd edition edition. Prentice Hall International (UK) Ltd.

[17] Susan Stepney, Fiona Polack & Ian Toyn (2003): *A Z Patterns Catalogue I: Specification and Refactorings*. Technical Report YCS-2003-349, Department of Computer Science, University of York.

[18] Nicolas Stouls & Virgile Prevosto (2015): *Aorai Plugin Tutorial*. Technical Report, INRIA. Available at `http://frama-c.com/download/frama-c-aorai-manual.pdf`.

[19] Samuel H. Valentine, Susan Stepney & Ian Toyn (2004): *A Z Patterns Catalogue II: Definitions and Laws*. Technical Report YCS-2004-383, Department of Computer Science, University of York.

[20] *VCC Website*. Available at `http://vcc.codeplex.com/`.

[21] L.M.G. de Vries (1996): *Applying Formal Methods in the DO-178B Certification Process*. Technical Report NLR TP 95547, National Aerospace Laboratory NLR, Amsterdam, The Netherlands.