Formal Specification and Automated Verification of Railway Software with Frama-C

Virgile Prévosto, Jochen Burghardt, Jens Gerlach, Kerstin Hartig, Hans-Werner Pohl, Kim Völlinger

▶ **To cite this version:**

Virgile Prévosto, Jochen Burghardt, Jens Gerlach, Kerstin Hartig, Hans-Werner Pohl, et al.. Formal Specification and Automated Verification of Railway Software with Frama-C. IEEE International Conference on Industrial Informatics - INDIN, Jul 2013, Bochum, France. cea-01835639

HAL Id: cea-01835639

https://hal-cea.archives-ouvertes.fr/cea-01835639

Submitted on 11 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Specification and Automated Verification of Railway Software with Frama-C

Virgile Prevosto
CEA, LIST
Software Safety Laboratory
91 191 Gif sur Yvette Cedex, France
Email: virgile.prevosto@cea.fr

Jochen Burghardt, Jens Gerlach, Kerstin Hartig,
Hans Pohl, Kim Voellinger
Fraunhofer FOKUS
Email: jens.gerlach@fokus.fraunhofer.de

***Abstract*—This paper presents the use of the Frama-C toolkit for the formal verification of a model of train-controlling software against the requirements of the CENELEC norm EN 50128. We also compare our formal approach with traditional unit testing.**

I. Introduction

Embedded systems increasingly rely on software to accomplish a wide variety of tasks. In some cases, *e.g.* train-controlling software, where a bug could have very serious consequences, it is crucial to provide strong guarantees that the code will always behave as intended. To this end, many industrial domains have established a norm, such as EN 50128 [1] for railways, that specifies how such an assessment must be done. Testing has always played an important role in validating software, and it is still the case nowadays. However, the number of test cases (hence the cost of running them) needed to cover all possible behaviors of the program grows very quickly with the size and the complexity of the code. Formal methods provide a way to cope with this issue by reasoning abstractly on the source code itself and establishing formally that it is conforming to its specification.

Formal techniques such as deductive verification have long been seen as a mainly academic exercise lacking industrial-grade tools except for some isolated experiments such as the B toolkit [2]. They have gained some traction in the past few years, with *e.g.* VCC [3], Boogie [4], Caveat [5], and Frama-C [6]. Recent evolution of the norms (for instance the 2011 version of EN 50128) reflect this fact by recommending, or even mandating, the use of such formal techniques, at least for the most critical part of embedded software.

In this paper, we extend our previous work [7] on formally verifying various safety-relevant functionalities in railway software by the results of a case study [8] where we compare our *unit proofs* with traditional *unit tests*. The software modules themselves have been written in C, while the requirements have been formalized through ACSL [9], the ANSI/ISO C Specification Language. C code and ACSL specifications have then been analyzed with the Frama-C tool set, more specifically the Jessie and WP plug-ins.

The remaining of this paper is structured as follows. Section II presents Frama-C and ACSL. We then describe one of the requirements investigated in section III and show how it can be formally specified in section IV. We also propose an implementation in section V. Set-up of the verification and testing frameworks are presented in section VI, while the results of their application are discussed in section VII. Finally, we draw the conclusions from this experiment in section VIII.

II. Frama-C and ACSL

Frama-C [10] is a toolkit dedicated to the analysis of C programs. It comes with a formal specification language ACSL [9], [11], which let the user state the properties that a given code is supposed to meet.

ACSL is based on the notion of *function contract*. In its basic form, such a contract is composed of the following parts.

- Precondition of the function, that is the properties that it *requires* from its caller.
- Postcondition, that is the properties that the function *ensures* when it returns normally.
- The set of locations that the function might modify (thereby ensuring that any other location is left untouched by the evaluation of the function).

Pre- and post-conditions are expressed as first-order formulas, that can refer to some built-in predicates dealing with pointers and memory accesses. Users can also define their own predicates and use them in further ACSL annotations (see section IV-A).

As an example, the following C function is expected to return the maximum value found in the array given as argument together with its length.

```
/*@
requires \valid(a+(0 .. len-1));
assigns \nothing;
ensures
  \forall integer i;
     0<=i<l ==> a[i] <= \result;
ensures
  \exists integer i;
     0<=i<l && a[i] == \result;
*/
int max_array(int* a, int len);
```

`max_array` requires to be given a valid memory block `a` of at least `len` elements. The **`assigns`** clause guarantees that it won't modify the global memory state. The first post-condition states that the value returned by `max_array` is greater than

any element of the array, while the second one states that the result must in addition be present in the array (otherwise, returning `INT_MAX` would be a valid implementation of the specification).

Given a function contract and a corresponding implementation, Frama-C then offers a variety of plug-ins that can be used to verify that the implementation fulfills its contract. We briefly describe here Jessie [12] and WP [13], the two plug-ins that have been used during this case study. Both plug-ins perform *deductive verification*, a technique based on Hoare logic [14]. From each contract and implementation, the plug-ins generate a set of *proof obligations*, whose verification ensures that the code is correct with respect to its specification. These proof obligations are first-order logic formulas, that can be discharged by automated theorem provers (ATP), or interactive proof assistants. Both plug-ins can use several ATP as back-end. Indeed, since each ATP comes with its own strengths and weaknesses, it is usually useful to use them in combination. ATP used in this case study are Alt-ergo [15], CVC3 [16], Simplify [17] and Yices [18]. Previous experiments [11] have shown that these ATP provide a good rate of success.

One key aspect of deductive verification concerns the handling of loops. Basically, for each loop in the program, the user must provide a set of **loop invariant**s, that is properties that are true when the program reaches the beginning of the loop, and are preserved by each loop step. From that, we can then inductively prove that these properties are true for any number of steps. In particular, they hold at the end of the loop[1] regardless of how many times we pass through it. Loop invariants thus allow one to abstract away the effects of the whole loop. Moreover, the invariants are the only properties that are known about the global state after the loop. It is thus important that they are precise enough to have the post-condition be deduced from them. For instance, if the implementation of `max_array` visits each cell and keeps track in `max` of the maximal value seen so far, an appropriate invariant would be the following.

```
/*@
 loop invariant 0<=i<=l;
 loop invariant
   \forall integer j;
     0<=j<i ==> a[j] <= max;
 loop invariant
   \exists integer j;
     0<=j<i && a[j] == max;
 loop assigns i,max;
*/
for(int i=0; i<l; i++) { ... }
return max;
```

We first state the bounds of the main index of the loop. Then, we specify the value of `max` with respect to the values seen so far in the array. In the end, we'll have `i==l`, and our post-conditions will directly follow from these invariants. Finally the **loop assigns** states that the loop only modifies `i` and `max` (and not *e.g.* the content of `a`).

[1] Termination is handled by specific annotations outside of the scope of this paper.

III. Requirements and Modelling

A vigilance device is a safety device that operates in case of incapacitation of the engine driver. Various control elements, such as push-buttons and pedals, belong to a vigilance device. On the one hand, any of these control elements is required to be reapplied at a certain timed interval. On the other hand, control elements are not permitted to be held for longer than a specified time. If the control elements are not applied correctly, the vigilance device must initiate a forced brake.

The requirement reads as follows:

> "The vigilance device is only activated while the locomotive is not at a standstill. If the engine driver continuously applies at least one of the vigilance device control elements (in active control stand) for more than 35 seconds, then the forced brake applies automatically. If the engine driver does not apply at least one of the vigilance device control elements (in active control stand) within a timed interval of 8 seconds, then the forced brake applies automatically."[19]

Some notions mentioned in this requirement need to be clarified further.

- A locomotive is at a standstill if and only if its velocity is less than 3 km/h. The negation of standstill is motion, which is defined by a velocity equal or greater than 3 km/h.
- Active control stand means the key switch is not set to "0"-position, which means the driver's control console is "on".
- A forced brake is automatically invoked by monitoring devices (such as the vigilance device), whereas the emergency brake is invoked by a human. However, both have the same effect: all brakes of the train apply automatically until the train is at a standstill and the trigger event is cleared.

Figure 1 depicts the UML state diagram for the vigilance device described above.

Initially, the state *vigilance device deactivated* is entered and kept as long as the locomotive is at a standstill. As soon as the locomotive is not at a standstill anymore, the transition *activate* leads to the state *vigilance device activated* which means the vigilance device is *waiting to be applied*. No measure needs to be taken as long as the events *push* and *release* occur alternately. If the device is staying in state *waiting to be applied* for 8s the action *apply forced brake* will be triggered. Similarly, if it is staying in state *applied* for 35s the action *apply forced brake* will be triggered as well. These transitions describe the event of incapacitation of the engine driver and the locomotive enters the state *braking*. Of course, the vigilance device just deactivates as soon as the locomotive is at a standstill again.

IV. Formal Specification with ACSL

For our case study we consider user-defined data types containing the core data of the locomotive and the vigilance

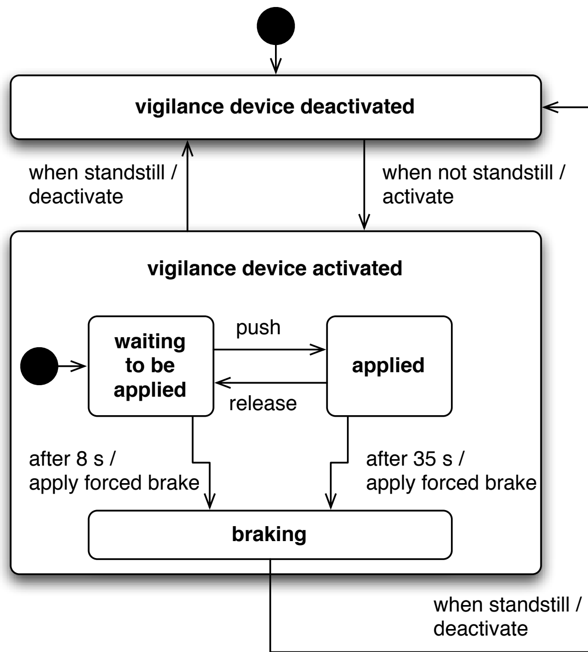

Fig. 1. Statechart of a vigilance device.

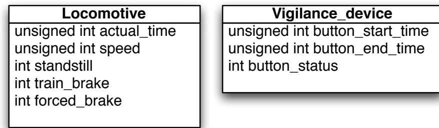

Fig. 2. Data Structures.

device. Figure 2 depicts the UML model of the data structures `Locomotive` and `Vigilance_device`.

In `Locomotive` the variable `actual_time` represents a relative-time counter expressed in milliseconds (msecs). The variable `speed` measures the velocity of the train in km/h. The variables `standstill`, `train_brake` and `forced_brake` are represented by an integer, but contain the Boolean values true or false, where `1` denotes `TRUE` and `0` denotes `FALSE`, as customary in C [20, §7.16].

The structure `Vigilance_device` contains the variables `button_start_time` and `button_end_time` describing the moments a vigilance device pedal or button was invoked and released the last time. The variable `button_status` represents a Boolean variable, which is true if any vigilance device pedal or button is currently invoked.

A. Domain-specific Predicates

We start our specification by defining some predicates that will allow us to express succintly the functions contracts. First, our informal requirements state that some integer variables may take only Boolean values, *i.e.*, `1`, represented as `TRUE`, and `0`, represented as `FALSE`. However, since a C integer may take more than these two values, we have to formalize this additional constraint. To achieve that, we can define an ACSL predicate as follows:

```
/*@ predicate true_or_false(int a) =
      (a == FALSE) || (a == TRUE); */
```

Then, we can state that there are relations between the fields of `Vigilance_device`. This is expressed in the predicate `vigilance_invariant` that is shown below. First, it is required that the status may only be true or false as it is a Boolean value. Therefore, we use our formerly defined predicate `true_or_false`. Furthermore, there exists an equivalence between the status of the buttons and the start and end time of their invocation. More precisely, the value of the status is true if and only if, the moment of releasing a button or pedal is after it was pushed last time. This predicate is in fact a *type invariant* of the `Vigilance_device` structure, that must be enforced at every program point.

```
/*@
 predicate vigilance_invariant
   (Vigilance_device* vig) =
   true_or_false(vig->button_status) &&
     (vig->button_status <==>
      vig->button_end_time
        < vig->button_start_time);
 */
```

Similarly, we can also formulate a type invariant `locomotive_invariant` for the user-defined data type `Locomotive`.

Below we can see an example of a predicate that checks whether the limit for holding a vigilance pedal or button has expired. According to the informal specification, this predicate remains true if any vigilance pedal or button is currently invoked (`status` must be true) and the difference between the actual time and the moment the button invocation was started exceeds thirty-five sec., represented by `MAX_VIGILANCE_BUTTON_HOLD`.

```
/*@
  predicate vig_button_hold_expired{L}
  (Vigilance_device* vig, Locomotive* loc) =
    vig->button_status &&
    (loc->actual_time - vig->button_start_time
                > MAX_VIGILANCE_BUTTON_HOLD);
 */
```

Note that we don't have to add here that `vig->button_status` is a Boolean, as this point is already part of `vigilance_invariant`, that all well-formed `Vigilance_device` in the program will be required to meet.

We can similarly formulate a predicate `vig_button_break_expired` that checks whether the pause limit between the invocation of any vigilance pedal or button has expired.

B. Function Contracts

Once our predicates are defined, we can provide a specification for the function `process_vigilance_device`. This function checks whether the vigilance device is processed correctly, and initiates forced brake if this is not the case. More precisely, it verifies that the vigilance buttons or pedals are neither held nor paused for too long. The ACSL function contract is shown below. This contract is structured in general pre- and post-conditions and two **behavior**s, which specify

two specific cases. On the contrary to **requires** clauses, **assumes** clauses are not necessarily fulfilled for each call. Namely, they guard the rest of the corresponding **behavior**, in the sense that the related clauses must only be verified in the case where the **assumes** clause holds.

```
/*@
  requires \valid(vig) && \valid(loc);
  requires locomotive_invariant(loc)
           && vigilance_invariant(vig);
  requires
    actual_time_is_latest_time(vig, loc);

  ensures locomotive_invariant(loc)
          && vigilance_invariant(vig);
  ensures
    actual_time_is_latest_time(vig, loc);

  behavior forced_brake_initiate:
   assumes !loc->standstill &&
    (vig_button_hold_expired(vig, loc)
     || vig_button_break_expired(vig, loc));
   assigns
     loc->forced_brake && loc->train_brake;
   ensures
     loc->forced_brake && loc->train_brake;

  behavior no_forced_brake_necessary:
    assumes loc->standstill
     || (!loc->standstill &&
         !vig_button_hold_expired(vig, loc) &&
         !vig_button_break_expired(vig, loc));
    assigns \nothing;

  complete behaviors;
  disjoint behaviors;
 */
void process_vigilance_device
     (Vigilance_device* vig, Locomotive* loc);
```

The function takes as parameters pointers to the state of the device and of the locomotive. Those pointers must be dereferenceable. We thus use the built-in **\valid** predicate to state this requirement. We also express that our user-defined predicates `vigilance_invariant` and `locomotive_invariant` must hold as pre- and postconditions. Additionally, we expect a formerly defined predicate `actual_time_is_latest_time` to hold as pre- and as postcondition, which means no time variable may have values describing the future relative to the actual time.

At the end of that contract we can specify that the described behaviors are supposed to be **complete** – meaning that they cover all possible cases in which the function can be called – and **disjoint** – meaning that we cannot be in both behaviors at the same time. These properties can be checked by looking at the **assumes** clauses of the behavior, as completeness means that the disjunction of the **assumes** always hold, while disjointedness means that their pairwise conjunction is always false. Inclusion of these last clauses provides an additional check on the specification itself.

V. Implementation

Below, the implementation of the function formally specified in the former section is depicted. Given the very precise formal specification, the implementation of this function is straightforward.

```
void process_vigilance_device(
  Vigilance_device* vig,
  Locomotive* loc)
{
  if (!loc->standstill) {
    if (check_vig_button_break_expired(
          vig, loc)
    {
      loc->train_brake = TRUE;
      loc->forced_brake = TRUE;
    }
    else if (check_vig_button_hold_expired(
               vig, loc))
    {
      loc->train_brake = TRUE;
      loc->forced_brake = TRUE;
    }
  }
}
```

This function calls two auxiliary functions:

`check_vig_button_hold_expired`

`check_vig_button_break_expired`

Each of those must be formally specified as well. This example shows, that the programmer needs to have knowledge about the specification language as well. This process is comparable to the fact, that the programmer must also be able to write unit tests.

VI. Unit testing

Now that we have a specification and a corresponding implementation, we want to check whether the implementation is correct with respect to its specification. As mentioned above, the Jessie and WP plug-ins of Frama-C have been used to generate proof obligations, that were then passed to ATP, namely Alt-ergo, CVC3, Simplify and Yices.

In parallel to this formal verification activity, unit tests have been derived from the specifications. The reason for the additional testing effort is of course, that unit testing is well-established in the software quality assurance process of safety critical systems. If one wishes to introduce formal methods on the level of software units (components), then one must provide convincing arguments on benefits compared to unit testing.

However, even if all tests, that have been derived from the requirements, have successfully passed, testing may not be finished because full *requirement coverage* does not guarantee full *code coverage*. Therefore, we have checked the code coverage obtained by functional testing using a code coverage analyzer and a suitable code coverage criterion. When necessary, we devised additional test cases until a full code coverage had been reached. Whenever a new test case was added, we provided arguments that this test case can be justified by the specification.

In order to measure the code coverage of our unit tests we have applied the TestCocoon tool. TestCocoon has been

released as open source software, but is no longer maintained. There is however a commercial successor, namely Squish Coco [21] which is compatible with TestCocoon.

The CoverageScanner function of TestCocoon can instrument the code for three different code coverage criteria: branch coverage, decision coverage and condition coverage. Since we want to validate our implementation according to the standard EN 50128, we have to answer the following question. How are these criteria related with those from the standard?

According to the standard, 100% branch coverage is reached by checking both branches of every decision. This matches exactly the decision coverage criterion from TestCocoon. In addition, TestCocoon always checks that all statements are exercised even though another criterion is specified. Checking if all statements are exercised is the statement coverage criterion from the standard, but is called branch coverage criterion for TestCocoon.

TestCocoon's strongest criterion is the condition coverage criterion, described in the manual as follows: "*reaching a coverage of 100% is more difficult with a condition coverage than with a decision coverage and than branch coverage*". TestCocoon's condition coverage matches the compound condition criterion from the standard. According to EN50128 standard, complete compound condition coverage is reached when "*every condition in a compound conditional branch (i.e linked AND/OR) is exercised*". Gaining complete condition coverage with TestCocoon means that every condition in a compound condition has to be evaluated at least once as false and once as true, whereas the compound condition must have been true and false itself. As mentioned above, every statement has to be exercised in addition.

Since TestCocoon's condition coverage criterion is its strongest and exceeds the compound condition coverage criterion from the standard, we have used it to check the code coverage reached by our test cases.

VII. Results of Verification and Testing

In addition to the vigilance device presented above, two other safety-relevant functionalities have been studied using the same approach. First, Speed Control is meant to check that the train's speed is always below the authorized limit and automatically slow it down if necessary. Second, the Blocking Device is responsible for powering up the locomotive only if an appropriate sequence of commands are given. We summarize in this section the results of the formal specification, formal verification and test of these three devices.

For each algorithm, the number of generated verification conditions (VC) are listed, as well as the percentage of proven verification conditions for each prover. Table I depicts the results of the individual provers for the three examined functionalities. For each algorithm there is at least one prover that is able to prove all verification conditions. CVC3 stood out in comparison to the other provers, since those had a higher percentage of unproven verification conditions.

The results of the code coverage analysis of the unit tests for the three components is shown in Table II.

		Percentage of Proved VCs			
Functionality	**# VC**	**Alt-Ergo**	**CVC3**	**Simplify**	**Yices**
Speed Control	23	96 %	100 %	96 %	100 %
Vigilance Device	62	97 %	100 %	97 %	94 %
Blocking Device	95	97 %	100 %	97 %	93 %

TABLE I. Verification results for each functionality.

Component	**Code Coverage**
Speed Control	100%
Vigilance Device	100%
Blocking Device	100%

TABLE II. Results of code coverage analysis.

Concerning quantitative aspects, Table III compares the size of the formal specifications, implementations, and unit tests.

Component	**Implementation**	**Specification**	**Testing**
Speed Control	12	30	54
Vigilance Device	22	46	94
Blocking Device	36	57	131

TABLE III. Number of lines per component and method.

As can be seen, the number of lines needed for specification is always larger than for implementation, but smaller than for testing. The measure of code lines employed here should not be confused with a measure of labor cost in person months. The latter is, however, difficult to collect in an objective way, as it depends in practice *e.g.* on the degree of familiarity of the developer with the respective method.

Of course, the suggested linearity and loop-class/slope dependency would need confirmation based on a broader set of programs and should include a varied group of stakeholders, including developers and verification experts from industry.

VIII. Conclusion

In this case study, we have applied both deductive verification and traditional unit testing to three selected safety-critical subsystems of a diesel locomotive control system. We have demonstrated the feasibility of formal specification with ACSL and verification of embedded software from the railway domain using Frama-C/WP.

Apart from the successful verification, the process of formal specification helped us to detect inaccuracies in the informal requirements: In their original version, it was not clear that the vigilance device shall be active only when the train moves. We reported this issue to the author of the requirement who subsequently provided a clarification. This shows that formal methods are capable of "enabling precise communication between engineers"[22, p. 1]. In general, we found graphic formal models (like the state charts shown in Sect. III) to be a good starting point both for deriving tests and specifications.

On the other hand, in the original version of the implementation of the blocking device, the time-out counter was not reset on a transition from state "*unblocked while motion*" to "*unblocked while standstill*". This bug was not detected by verification due to lack of a corresponding formal requirement. The process of formal specification helped us to detect ambiguities of informal requirements whereas a mature

process of deriving test cases can provide safeguards against incomplete specifications.

Although EN 50128 “highly-recommends” formal verification of SIL 3/4 software, it still mandates functional testing as the primary source of evidence [1, Table A.5]. This leads to uncompensated effort increases by formal verification in the railway domain. We suggest to discuss an approach similar to that in the aerospace domain, where formal methods nowadays may replace certain test activities. Like in [22], certain overall integration tests would remain indispensable, and caution should be exercised as to define suitable formal-specification completeness criteria, similar to test coverage criteria.

Acknowledgment

The work described in this paper has been partially funded by the ANR/BMBF PICF project Device-Soft and the ITEA2 project Open-ETCS.

We also express our gratitude to Dipl.-Ing. Jürgen Busse, an assessor for railway software, for sharing his expertise about the assessment process of railway software and formulating the informal safety requirements on which this work is based.

Finally, we’d also like to thank the anonymous referees for their helpful remarks on an draft version of this paper.

References

[1] CENELEC, European Committee for Electrotechnical Standardization, “EN 50128: Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems,” Tech. Rep., Jun. 2011.

[2] J.-R. Abrial, *The B Book - Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[3] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte, “VCC: Contract-based Modular Verification of Concurrent C,” in *International Conference on Software Engineering, ICSE*, 2009.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, “Boogie: A Modular Reusable Verifier for Object-Oriented Programs,” in *FMCO*, ser. LNCS, vol. 4111, 2005.

[5] F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, J. Raguideau, and D. Schoen, “Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach,” in *the Wold Congress on Formal Methods in the Development of Computing Systems (FM’99)*, 1999, pp. 1798–1815. [Online]. Available: http://dl.acm.org/citation.cfm?id=647545.730777

[6] Frama-C, “Frama-C homepage,” http://frama-c.com/.

[7] K. Hartig, J. Gerlach, J. Soto, and J. Busse, “Formal Specification and Automated Verification of Safety-Critical Requirements of a Railway Vehicle with Frama-C/Jessie.” in *FORMS/FORMAT*, E. Schnieder and G. Tarnai, Eds. Springer, 2010, pp. 145–153. [Online]. Available: http://dblp.uni-trier.de/db/conf/forms/forms2010.html#HartigGSB10

[8] J. Burghardt, J. Gerlach, K. Hartig, H. Pohl, and K. Völlinger, “Formal Specification and Automated Verification of Railway Software with Frama-C,” Fraunhofer FOKUS, Tech. Rep., Jun. 2012.

[9] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language, v1.6*, Sep. 2012, http://frama-c.com/acsl.html.

[10] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-C: A software analysis perspective,” in *Software Engineering and Formal Methods, SEFM*, ser. LNCS, vol. 7504, Oct. 2012.

[11] J. Burghardt, J. Gerlach, K. Hartig, H. Pohl, and J. S. K. Völlinger, *ACSL by Example*, version 7.1.0 (for Frama-C Nitrogen).

[12] Y. Moy and C. Marché, *Jessie Plugin Tutorial*.

[13] L. Correnson and Z. Dargaye, *WP Plug-in Manual, version 0.5*, Jan. 2012.

[14] C. A. R. Hoare, “An axiomatic basis for computer programming,” vol. 12, no. 10, pp. 576–580 and 583, Oct. 1969.

[15] Alt-ergo, “Alt-ergo homepage,” http://alt-ergo.lri.fr/.

[16] C. Barrett and C. Tinelli, “CVC3 homepage,” http://www.cs.nyu.edu/acsys/cvc3/.

[17] Simplify, “Simplify homepage,” http://freshmeat.net/projects/simplifyprover/.

[18] SRI International, “Yices homepage,” http://yices.csl.sri.com/.

[19] J. Busse, “Sicherheitsanforderungsspezifikation Diesellok,” Institut für Bahntechnik GmbH (IfB), Technical Note 2009-408300-26, Oct. 2009, unpublished.

[20] *9899:TC3: Programming Languages—C*, ISO/IEC JTC1/SC22/WG14, 2007, www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[21] “Squish Coco, Code Coverage Measurement for C/C++ or C#,” http://doc.froglogic.com/squish-coco/2.0/squishcoco.pdf, 2012/04/20.

[22] RTCA SC-205, “Formal Methods Supplement to DO-178C and DO-278A (DO-333),” Radio Technical Commission for Aeronautics (RTCA Inc.), Tech. Rep., Dec. 2011.