OETCS/WP2/D2.4.0 – 00/00

openETCS

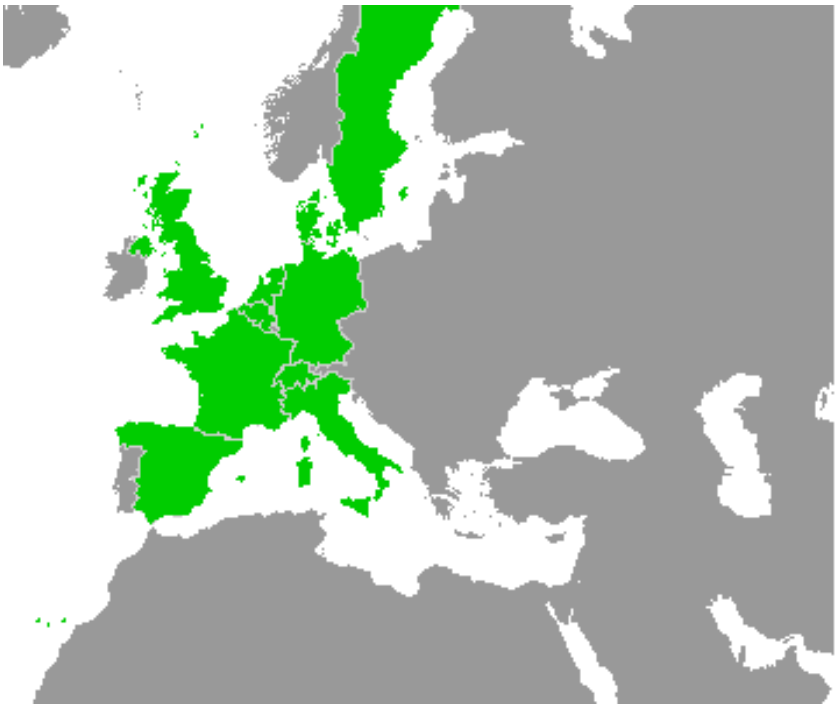
ITEA2 Project
2012 – 2015

Work-Package 2: “Definition”

OpenETCS methods

Definition of the methods used to perform the formal description

Marielle Petit-Doche and David Mentré February 2013



This page is intentionally left blank

OpenETCS methods

Definition of the methods used to perform the formal description

Marielle Petit-Doche

Systerel

David Mentré

Mitsubishi Electric R&D Centre Europe

Definition

Prepared for ITEA2 openETCS consortium
Europa

Abstract: This document give a description of the process to be applied in the OpenETCS project. It gives a description of the activities to specify and design a critical system in a first part. The second part presents an abstract description of the case study issued from subset 26.

Detailed process in regards of state of the art results and Cenelec standard requirements (detailed design plan) - For each step of the process defined in T.2.2.1 : - objectives of the step - Input/output - proposed techniques - rules on languages and tools

Table of Contents

Figures and Tables **Figures**

Tables

1 Introduction

The purpose of this document is to describe, for the OpenETCS project, the means and methods used to perform the formal description.

However the benchmark activities are not yet achieved in WP7, such the definition of the methods can not yet be done.

For the intermediate version of the document, we propose a description of the benefits of formal methods in the design, development, verification and validation of critical systems.

Open Issue. Is the detailed description of the formal methods to used during the openETCS project necessary ? If yes, can we postpone the final version of the document after the choice of the methods ? How the document can then be compatible with the deliverable of WP7 ?

Otherwise what is expected from this document ?

2 Reference documents

- CENELEC EN 50126-1 — 01/2000 — *Railways applications — The specification and demonstration of Reliability, Availability, Maintenability and Safety (RAMS) — Part 1: Basic requirements and generic process*
- CENELEC EN 50128 — 10/2011 — *Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems*
- CENELEC EN 50129 — 05/2003 — *Railway applications — Communication, signalling and processing systems — Safety related electronic systems for signalling*
- FPP — *Project Outline Full Project Proposal Annex OpenETCS* – v2.2
- SUBSET-026 3.3.0 — *System Requirement Specification*
- SUBSET-076-x 2.3.y — Test related ERTMS documentation
- SUBSET-088 2.3.0 — *ETCS Application Levels 1 & 2 - Safety Analysis*
- SUBSET-091 2.5.0 — *Safety Requirements for the Technical Interoperability of ETCS in Levels 1 & 2*
- CCS TSI — *CCS TSI for HS and CR transeuropean rail has been adopted by a Commission Decision 2012/88/EU on the 25th January 2012*

3 Conventions

The requirements are prefixed by “R-zz-x-y”, and are written in a roman typeface, where “R” stands for “Requirement”, “zz” identifies the source document,“x” is the version number and“y” is the identifier of the requirement. All the text written in italics is not a requirement: it may be a note, an open issue, an explanation of the requirements, or an example.

The placeholder “ %%xxx%% ” is used to indicates that a paragraph or section is not finished, to be defined or to be confirmed.

4 Glossary

API Application Programming Interface

FME(C)A Failure Mode Effect (and Criticity) Analysis

I/O Input/Output

OBU OnBoard Unit

QA Quality Analysis

RBC Radio Block Center

RTM RunTime Model

SIL Safety Integrity Level

THR Tolerable Hazard Rate

V&V Verification & Validation

5 Short introduction on formal approaches to design and validate critical systems

5.1 What is a formal approach?

A *formal* approach is a way to describe system or software that builds upon (i) rigorous syntax and (ii) rigorous semantics.

The *syntax* defines how the system or software is described. It is usually made through a grammar and a set of additional constraints. It can be textual or graphical.

The *semantics* roughly describes how the system or software evolves along time. It is defined using a mathematical model, i.e. use of mathematical objects attached to each element of the syntax and mathematical rules that define how those objects evolve. The mathematical models used can be very different from a formal approach to another one. For example B Method uses the Generalized Substitutions, SCADE relies on the Synchronous language Lustre, etc.

A *semi-formal* approach is one where the syntax is precisely defined but the semantics is not precisely defined, usually through some English text. Typical semi-formal approaches are Matlab language or SysML/UML formalisms.

A semi-formal approach can become formal if its semantics is rigorously defined through a mathematical model.

5.2 When formal approaches are recommended according to CENELEC standard?

The use of formal approaches is *Highly Recommended* for SIL3 and SIL4 software according to CENELEC EN 50128:2011.

5.3 Which constraints are required on the use of formal approaches?

Each formal approach has some restriction on the kind of software or system it can be applied on. Moreover, each formal approach is specialized in the verification of some kind of property. Therefore a formal approach should be chosen in accordance to verification objectives.

Moreover, using a formal approach can impact the overall system building process. For example software developed using the B Method follow a specific process and imposes a very specific architecture, very different from designing C software. In the same way, use of formal approach can impose specific resource needs at different phases of project lifetime. For example, more work on the requirement analysis and formalization phase.

Last but not least, as a formal approach brings its benefits only inside a given boundary, the process should be designed to keep those benefits outside those boundaries. For example, code compilation of a verified source code should be done in such a way as to ensure that the verified properties are kept in the compiled code.

5.4 Which are the benefits to use formal approaches?

Several benefits are expected from the use of formal approaches.

The first benefit is to enhance the understanding of the formalized system or software. By using a non ambiguous notation, the designer is forced to clarify his mind. Very often, several design issues or defects are found at this step.

The second benefit is to enable the verification of some properties in an exhaustive way. Therefore avoidance of certain kinds of bugs can be guaranteed. Of course, such guarantee can only be obtained if the formal method is used along some specific way and on a well delimited part of the software and system (for example one cannot guarantee properties on variables outside program boundary).

The third benefit is to allow Correct by Construction software or system building. By verification properties along the construction cycle of a system or software, one can ensure that some formalized requirements are fulfilled in the final software. For example, one can ensure that some variables stay in well defined boundaries.

6 Formal approaches for the design and development of a system

Very roughly, from an engineering point of view three kinds of formal methods can be used for the design and development of a system:

- Pre/Post-condition approach;
- Model checking of concurrent and synchronous languages;
- Static analysis of software code.

6.1 Contract based approach

Contract based approaches are based on software (or model) annotation. The software is usually considered state based, i.e. made of a state stored in a set of typed variables. Software is divided in a set of operations (aka procedures, functions, methods, ...). To each operation is associated a pre-condition, i.e. a set of conditions that should be guaranteed at operation entrance by the caller of the operation. To each operation is also associated a post-condition, that the operation should fulfill, provided the pre-condition is assumed. In other words, the called operation should ensure the post-condition. The pre and post-conditions are usually expressed using first order logic (and, or, implication, for all and exists quantifiers, etc.).

In the contract based approach, if all the pre and post-conditions are fulfilled for all possible execution, then we can guarantee that all the operations work well together.

Those kind of approaches are known to be scalable, at the price of sometimes a lot of manual work to properly annotate the software or prove the annotations are correct.

Example of such approaches are B Method, Event-B, GNATprove/SPARK on Ada language or Frama-C on C language.

6.2 Model checking of concurrent and synchronous languages

In this approach, a model is based using various textual or graphical formalisms: state based model (like State Machines), data flow equations or Petri Nets.

In a second step, a property is formalized over this model, usually using temporal logic. A temporal logic is usually a first order logic augmented with operators expressing the relationship between events: in the next event, a property is true until another property is true, etc.

Then, model checking techniques (symbolic model checking, exhaustive state enumeration, ...) are applied to check that the expressed property is valid over the model, for all possible executions.

Compared to previous contract based approach, model checking allows to verify more complex properties along the life time of the system. For example, one can express that “something good” will occur in the future after a certain event.

However model checking suffers from state explosion problem: if the model is not properly designed, it can have too many states and make the exhaustive verification impossible.

Example of such approaches and tools are SCADE, Petri Nets, NuSMV, UPPAAL or SPIN.

6.3 Static analysis of software code

Static analysis techniques are inspired by abstract interpretation techniques proposed by Cousot and Cousot in 1977. The main idea is to transform the domain of concrete program variables into a simpler, abstract domain. Then the analysis is done, for all possible execution paths, within this abstract domain. And finally the result of the analysis is put back on the original concrete variables.

Constructing an abstract interpretation is done using specific mathematical approaches (mainly Galoi connections) that ensure that the result of the analysis is sound: if an issue is found by the analysis in the abstract domain, it exists in the concrete domains of the variables, i.e. in the real program.

On the contrary, completeness of the approach is difficult to ensure: due to abstraction, the analysis can make some approximation. In such case the result of the analysis is meaningless: the analysis cannot tell if a verified property is valid or not.

The main advantage of static analysis is that it works on actual, concrete software code, with minimal annotations. It thus integrates quite easily with existing development process, along testing phase for example. And it is highly automated, requesting minimal user intervention.

The drawback of this approach is that it is restricted to certain kind of properties (overflow, underflow, out-of-bound accesses, division by zero). Moreover it does not apply well to all kind of programs.

Example of tool applying such approach are Polyspace, Astrée or Frama-C (with Value analysis plug-in).

7 Formal approaches for V&V of a critical system

The various approaches previously presented can be used to check various kind of properties:

- safety properties: ensure the system is safe;
- functional properties: ensure the system works as expected regarding its functional behavior;
- non-functional properties: ensure the system works has expected regarding its speed, capacity, ...

Due to the cost and complexity of formal analysis, use of formal methods in the railway domain is usually focused on ensuring only safety properties. We only consider them is the remaining of this section.

7.1 Formal approaches for verification

Ensuring safety properties using formal methods starts in a similar way to classical approaches. A safety analysis will produce the properties that should be ensured to guarantee safe operation of the system.

Usually such safety properties are high level properties (e.g. “two trains do not collide”). In order to be amenable to formal verification, they should be translated into properties related to the system state (e.g. “there exists two free blocks between any occupied blocks”, ...). Several system-related safety properties can be associated to a single high-level safety property.

Then those properties are checked to be valid, i.e. in any system state a safety property is always true. This can be done with the various approaches presented previously.

If Correct by Construction approach is used, some traditional verification activities like unit or integration tests can be avoided because they are ensured by the formal approaches.

7.2 Validation of formal approaches

Proving that a safety property is always valid on a system model does not ensure the property is valid in the real life. Discrepancies can occur between the system model and the real system. Moreover, errors can occur during the formalization of the high-level safety property into a set of system-related properties.

Therefore a validation activity is needed. Validation checks that formalization of properties is correct, as well as all related assumptions.

This is done using non formal techniques:

- Review;

- Simulation and animation;
- Test.

7.3 Formal methods for safety

Comment. proof of safety requirements, static analysis, safety analysis, traceability,...

8 Guidelines on the approaches used for OpenETCS

8.1 Definition of an OpenETCS methodology

R-WP2/D2.3.0-X-1 The model formalism shall be easily understandable by the domain experts.

R-WP2/D2.3.0-X-2 The safety properties should be provided in a declarative, simple and formal language.

R-WP2/D2.3.0-X-3 The formal model shall be understandable by or exportable to many tools (SCADE, Simulink, B tools, OpenETCS tool chain...)

R-WP2/D2.3.0-X-4 Formal specifications should be able to formalize:

R-WP2/D2.3.0-X-4.1 State machines,

R-WP2/D2.3.0-X-4.2 Time-outs,

R-WP2/D2.3.0-X-4.3 Truth tables,

R-WP2/D2.3.0-X-4.4 Arithmetics,

R-WP2/D2.3.0-X-4.5 Braking curves,

R-WP2/D2.3.0-X-4.6 Logical statements.

Comment. This requirement does not state that all these objects need to be first order objects *of the language. It only state that it should be possible (easy?) to formalize and manipulate them.*

It is to be noted that if (for example) braking curves are objects of the language, it shall be proved that they are sound, and that the code generation for these objects is also sound.

R-WP2/D2.3.0-X-5 The formal model shall be executable.

R-WP2/D2.3.0-X-5.1 The formal model shall be executable in debug mode (step-by-step), allowing inspection of states, variables and I/O.

R-WP2/D2.3.0-X-5.2 The environment shall be emulated by high level construction of the inputs.

Justification. "High level" means that it will not be necessary to define bitwise the inputs at each cycle. On the contrary, some motorization will be available to define the behavior of the inputs.

R-WP2/D2.3.0-X-6 It shall be possible to assert logical properties on the model (*i.e. invariants*).

R-WP2/D2.3.0-X-6.1 It shall be able to check the conformance of these properties at runtime.

R-WP2/D2.3.0-X-6.2 It shall be able to prove the conformance of the model to these properties.

8.2 Use of tools

R-WP2/D2.3.0-X-7 The tool chain shall be sufficiently robust to allow large software management.

R-WP2/D2.3.0-X-7.1 It shall allow modularity at any level (proof, model, software).

R-WP2/D2.3.0-X-7.2 It shall allow the management of documentation within the same tool.

R-WP2/D2.3.0-X-7.3 It shall allow distributed software development.

R-WP2/D2.3.0-X-7.4 It shall include an *issue-tracking system*, in order to allow change management and errors/bugs management.

R-WP2/D2.3.0-X-7.5 It shall allow to document/track the differences between the model and the ERTMS reference.

*Justification. In case where errors are found in the specification, or reducting choices are to be made in the model (*e.g. *in case of non-determinism).*

R-WP2/D2.3.0-X-7.6 It shall allow concurrent version development, or be compatible with tools allowing concurrent version development.

R-WP2/D2.3.0-X-7.7 In particular, it shall be made easy [1] to track the differences between two releases of a model and to manage conflicts.

R-WP2/D2.3.0-X-7.8 In particular it shall allow to track the roles and responsabilities of each participant on a configuration item, at each step of the project lifecycle.

[1] Especially in the case of a graphical language.

R-WP2/D2.3.0-X-7.9 In particular, version management shall allow to track version of the safety properties together with the model.

R-WP2/D2.3.0-X-8 The tool chain shall allow traceability between the documentation (in particular the specification) and the models and safety properties.

R-WP2/D2.3.0-X-9 The tool chain shall allow traceability between the different layers of model and safety properties.

R-WP2/D2.3.0-X-10 The tools used in the tool chain shall be able to cooperate, *i.e.* the outputs of one tool will be suitable to be used as the inputs of the other tool.

R-WP2/D2.3.0-X-11 The tool chain shall conform to 50128 requirements, for the corresponding SIL and tool class.

R-WP2/D2.3.0-X-11.1 For T2 and T3 tools[2], the choice of tools shall be justified, and the justification shall include how the tool's failures are covered, avoided or taken into account (ref. to EN 50128 6.7.4.2).

R-WP2/D2.3.0-X-11.2 All T2 and T3 tools must be provided with their user manuals.

R-WP2/D2.3.0-X-11.3 For all T3 tool, the proof of correctness or the measure taken to guarantee the correctness of the output w.r.t. their specification and the inputs shall be provided.

R-WP2/D2.3.0-X-11.3.1 . . . for data transformation,

R-WP2/D2.3.0-X-11.3.2 . . . for software transformation (*e.g.* translation, compilation. . .).

R-WP2/D2.3.0-X-12 The tool chain shall allow to write and store *test cases* and *use cases* for the model.

R-WP2/D2.3.0-X-12.1 Version management will allow to map test cases version to model versions.

R-WP2/D2.3.0-X-13 The tool chain shall allow to generate test cases for the model.

Open Issue. Is it really necessary? If we have formal proofs on the models, the tests should stay at a functional level. Therefore generated test cases should not be interesting in this context.

Open Issue. TBD requirements on the prover. Should it verify De Bruijn's criterion[3]*. Should at least the proof tree be exportable and checkable in another tool? (if the proof tree itself is mandatory).*

[2] T2: Tools contributing to the test or verification of the code or design *e.g.* static analyzers, test generators. . .)
T3: tools contributing directly or indirectly to the final code or data (*e.g.* compilers, code translator. . .)

[3] *I.e.* to be able to produce a proof tree that could be verified by a simple proof checker.

