OETCS/WP2/D2.4 – 00.07

openETCS

ITEA2 Project
2012 – 2015

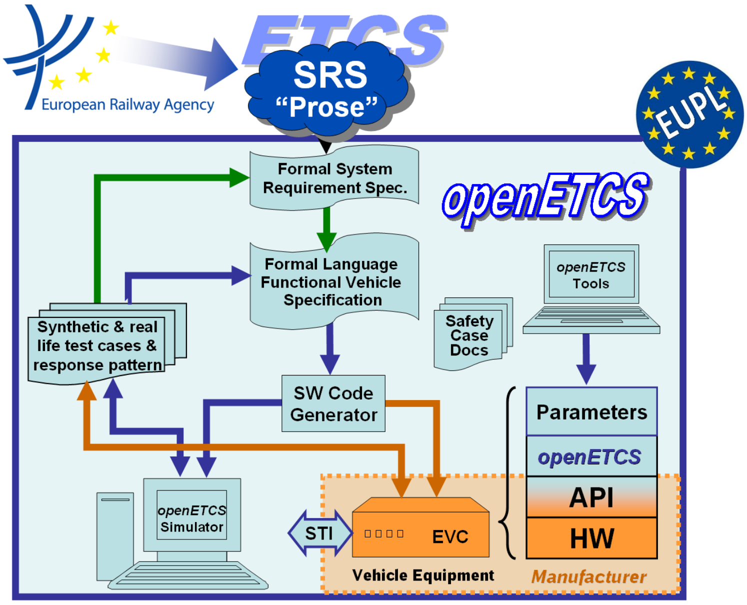
Work-Package 2: “Definition”

OpenETCS methods

Definition of the methods used to perform the formal description

Marielle Petit-Doche, David Mentré and Mathias Güdemann

January 2014



Funded by:

Federal Ministry of Education and Research

Région de Bruxelles-Capitale

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

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

Mathias Güdemann

Systerel

Definition

Prepared for ITEA2 openETCS consortium
Europa

Abstract: This document give first an introduction to formal methods. In a second part, it proposes the method to follow during the openETCs project according to the methodology selection.

Table of Contents

Figures and Tables **Figures**

Tables

Document information	
Work Package	WP2
Deliverable ID or doc. ref.	D2.4
Document title	Definition of the methods used to perform the formal description
Document version	00.07
Document authors (org.)	Marielle Petit-Doche (Systerel) David Mentré (Mitsubishi Electric R&D Centre Europe) Mathias Güdemann (Systerel)

Review information	
Last version reviewed	00.04
Main reviewers	S. Baro (SNCF) P. Mahlmann (DB), B. Hekele (DB) H. Hungar (DLR), M. Behrens (DLR) J. Welte (TU-BS)

Approbation			
	Name	Role	Date
Written by	Marielle Petit-Doche David Mentré Matthias Güdemann	T2.4 Sub-Task Leader	January 2014
Approved by	Gilles Dalmas	WP2 leader	

Document evolution			
Version	Date	Author(s)	Justification
00.01	03/05/2013	M. Petit-Doche	Incorporation of section on formal methods written by David Mentré
00.02	2013-05-16	D. Mentré	Incorporation of formal reviews in the document
00.03	03/05/2013	M. Petit-Doche	Remaining remarks of formal reviews
00.04	26/09/2013	M. Petit-Doche, D. Mentré	First version of sections 7, 8 and 9
00.05	16/12/2013	M. Petit-Doche	Section 7
00.06	03/01/2014	Uwe Steinke	Section 7.4 and 7.5
00.07	09/01/2014	M. Petit-Doche	Issues #33,

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 approaches in the design, development, verification and validation of critical systems in sections 4, 5 and 6.

For the final version, the document proposes the method to follow during the OpenETCS project according to the OpenETCS process and requirements defined in D2.3 and D2.6. Section 7 gives some guidelines for design and VnV and some conventions. Section 8 describes how to use SysML in the OpenETCS process.

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* (this version is related to version 2.3.y of SUBSET-026)
- 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*
- D1.3 – Project Quality Assurance Plan
- D2.1 – Report on existing methodologies
- D2.2 – Report on CENELEC standards
- D2.3 – Definition of the overall process for the formal description of ETCS and the rail system it works in
- D2.6 – Requirements for OpenETCS

3 Glossary

API Application Programming Interface

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

FIS Functional Interface Specification

HW Hardware

I/O Input/Output

OBU On-Board Unit

PHA Preliminary Hazard Analysis

QA Quality Analysis

RBC Radio Block Center

RTM RunTime Model

SIL Safety Integrity Level

SRS System Requirement Specification

SSHA Sub-System Hazard Analysis

SSRS Sub-System Requirement Specification

SW Software

THR Tolerable Hazard Rate

V&V Verification & Validation

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

4.1 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 description is built and valid. It is usually made through a grammar and a set of additional constraints. It can be textual or graphical.

The *semantics* gives a meaning to each object found in the system or software description. This meaning is given 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 interacts with other objects. The mathematical models used can be very different from one formal approach to another one. For example the B Method uses the Generalized Substitutions, SCADE relies on the Synchronous language Lustre, etc. One should notice that being able to compile or run a language is not enough to give it some semantics, as this semantics is hidden within the execution/compilation steps. An explicit document should be provided. This document can be informal (e.g. the B-Book) or formal (BiCoq formalization of B Method in Coq formal language).

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 the Matlab language or the SysML/UML formalisms.

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

4.2 When are formal approaches recommended according to CENELEC standard?

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

4.3 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 to. Moreover, each formal approach is specialized in the verification of some kind of property. Therefore a formal approach should be chosen in accordance to the verification objectives.

Moreover, using a formal approach can impact the overall system building process. For example software developed using the B Method follows a specific process and imposes a very specific architecture, very different from designing C software. In the same way, the usage of a formal approach can impose specific resource needs at different phases of the 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 development process should be designed to transfer these benefits beyond 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.

4.4 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, and in general, fixing errors at this step is much less costly than in later development phases.

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 approach 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 verifying 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.

The fourth benefit is the ability to easily extend the formalized system or software, by updating the formal description. After such an update, applying the formal verification allows to know precisely which parts are no longer valid and focus development effort on them, without the need to re-verify parts not impacted by the change.

4.5 How to use formal approaches?

In the design and development of a system using an approach based on formal languages, there are two orthogonal aspects to consider: at which stage (or stages) in the development cycle the formal approach will be used and how it will be used, i.e., choice of approach, technical realization.

In the development cycle, there are three main stages where a formal approach can be applied:

- Formalization of Requirements
- Design Support
- Implementation Verification

4.5.1 Formalization of Requirements

In the System Development Phase and Software Requirements Phase, a formal approach applied to initial requirements can bring clarifications, by enforcing a non-ambiguous meaning for all parties. In case making such a formalization of requirements is difficult, it usually triggers further clarification efforts between involved parties.

4.5.2 Design Support

In the Design and Architecture Phase, a formal approach can support the system design and architecture design. In this phase, systematic errors can be detected which can be very difficult and costly or even impossible to fix later.

In combination with a refinement based correct by construction approach, it is possible to have high level properties on the whole system which are refined to sub-properties on the different parts of the system architecture while designing the system. An example of such an approach is the Event-B method.

4.5.3 Implementation Verification

In the later phases of the development process, formal approaches can deal with formal reasoning over the actual functional system source code. Depending on the method, this code can be generated from a formal model, derived via a refinement based approach or written manually, annotated with formal properties.

Code generation from a higher level model is in particular interesting, if the generator is qualified and code generation can reduce the required testing of code. A refinement based approach will iteratively add detail to a high level description until a detail level is reached which can be implemented in programming languages, here often translation, i.e., side-by-side creation of refined model and source code is used. And finally it is possible to manually write code which is annotated with properties that can be verified formally (see also Section 5.1. An example for a code generation based approach is SCADE, the B method is based on refinement and formal proof and Frama-C, GNATprove or SPARK are based on source code annotation.

5 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:

- Contract based approaches;
- Model checking of concurrent and synchronous languages;
- Static analysis of software code.

5.1 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 a pre-condition is associated, i.e., a set of conditions that should be guaranteed at operation entrance by the caller of the operation. To each operation there is also a post-condition associated 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, . . .).

In the contract based approach, if all the pre and post-conditions are fulfilled for all possible executions, 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.

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

5.2 Model checking of concurrent and synchronous languages

In this approach, models are based on 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.

On the other hand, model checking requires a finite state space. In general systems represent an infinite state space and therefore a finite abstraction must be derived for model checking. 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. To limit this problem , model-checking approaches are often links with abstraction and decomposition techniques.

Example of such approaches and tools are Design Verifier (used in SCADE), Petri Nets, NuSMV, UPPAAL or SPIN.

5.3 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 Galois 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 (ie. ensuring that any issue of the concrete domains are covered by an issue of the abstract domain) is difficult to ensure: due to abstraction, the analysis can make some approximation. In such cases 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).

6 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.

6.1 Formal approaches for verification

Ensuring safety properties using formal approches 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 partitioned into properties related to the system state (e.g., “there exists two free blocks between any occupied blocks”, ...) and properties for specific system parts. Several system-related safety properties can be associated to a single high-level safety property. In general it should be verified that as a whole the partial properties imply the high level properties.

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 a Correct by Construction or refinement based approach is used, some traditional verification activities like unit or integration tests can be avoided because they are ensured by the formal approaches. In this case the high level property is refined side by side with the system model, iteratively proving the correctness of the refinement steps.

6.2 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.

6.3 Formal approaches for safety

Formal approaches are usually used to support safety analysis to automatize some reasoning in view to obtain an exhaustive analysis. One possibility is to use dedicated language as AltaRica, or formal language and model-checking to inject fault in a model and analyse the effects. This can allowed to support FMEA or Fault-Tree generation.

A second possibility is to use formal methods to focus on the verification and validation of safety requirements on a formal model during system design.

7 Guidelines on the approaches used for OpenETCS

According to the WP7 decision meeting, the 4th of July, in Paris, SysML, supported by the Papyrus tool, has been chosen to cover the highest level of modelling.

The choice of the approaches for the lower levels of modelling is not yet fixed.

This section gives a proposal on how to use the selected approaches to produce from the input documents (ERA documentation and complements) to a SIL4 code. this section provides also the common structure and convention for modelling, verification and validation activities whatever was the approaches used: definition of the useful data and naming convention.

7.1 Sum up of chosen approaches and artifacts

The figure 1 gives the main step of the design approach and main used and produced artifacts. All is detailed in the following sections.

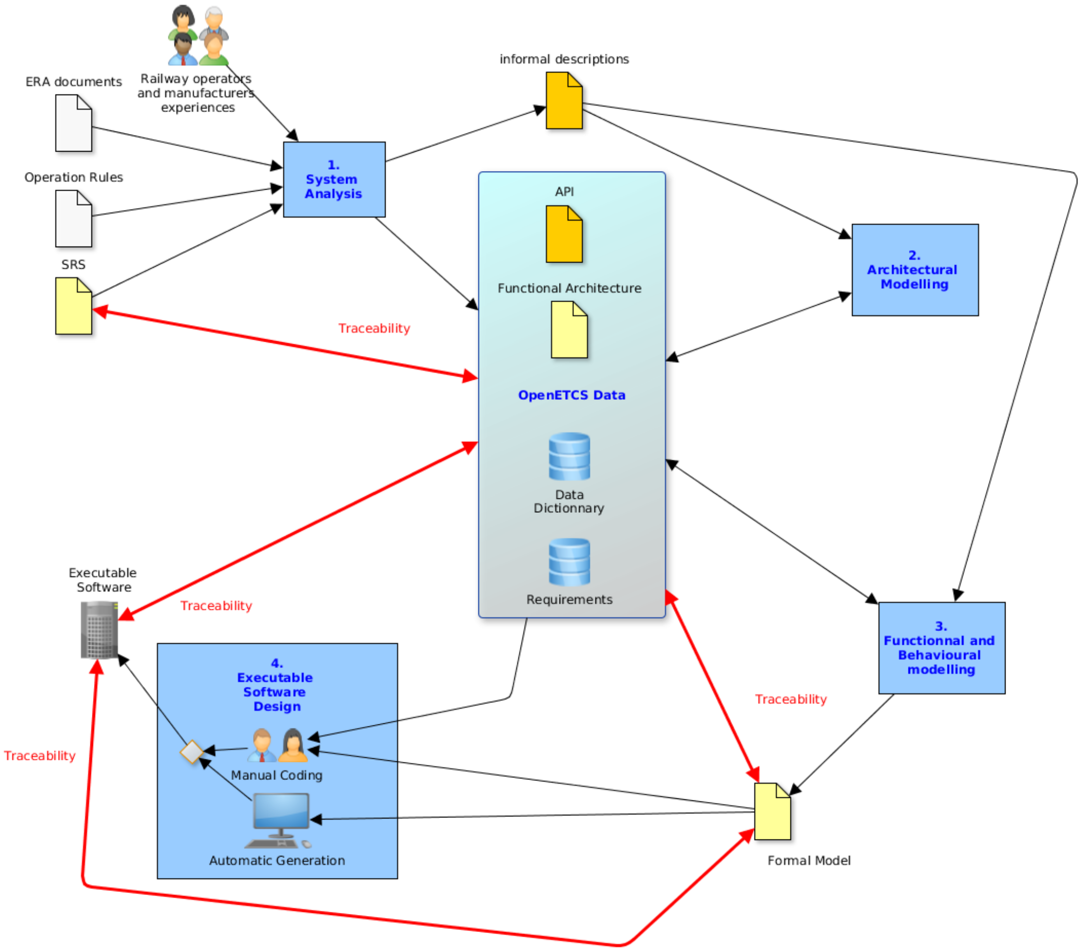


Figure 1. Main steps of the design approach

7.1.1 System analysis

Aim

The objectives of this step is to clarify the scope of the study and to provide a detailed description of the chosen solution to cover the requirements of the main input document of the project [SRS-Subset 26 v3.3.0].

Thus the tasks of this step are:

- to define the scope of the model (more or less the OBU kernel);
- to provide an architecture for this model (functional and software);
- to lift ambiguities;
- to detect errors and inconsistencies.

Input artifacts

The input documents and elements of this activity are also inputs of the project as it is its first activity.

- SRS Subset 26 v3.3.0 is the reference document and can be viewed as an user need document.
- All the documents provided by ERA according to the directive [CCS TSI] available `http://www.era.europa.eu/Core-Activities/ERTMS/Pages/Set-of-specifications-2.aspx`.
- Experience of the railway operators partners
- Experience of the railway manufacturers partners
- National and Operation Rules of the operators

Output artifacts

The output shall provide a clear view of the system to design and will be composed of:

- a set of informal descriptions (scope of the process, description of the functions, behaviour of the system, ...)
- an API to describe the environment of the system to design, its interfaces and its dynamic implementation
- a functional architecture which identifies the functions to design and the interaction between these functions and with the environment
- a data dictionary with description of the input and output variables of the system and all the internal variables needed to describe the system and the interactions between the functions
- all the requirements, allocated to function or data description, in natural language. The requirements from the SRS can be possibly split and rewritten in order to restrict their scope to the functions or to match the objects of the data dictionary. New requirements can be defined to describe specification choices or to clarify the behaviour of a function (for example according to the experience from a partner). Traceability issue is mandatory and shall be taken into account early.

Means

The first element needed is a way to manage and organize informal descriptions (including text, pictures, tables,...).

Functional architecture can be easily defined with BDD and IBD diagrams (see 8). Data dictionary and requirement sets shall be tool supported in order to link their contents to the functional architecture and to be reused during the modelling, verification, validation and safety activities.

As functional architecture, data dictionary and set of requirements shall be linked together, and shall be defined to be used during all the project. Thus section 7.2 give some specification of the items to define and some naming conventions.

7.1.2 Architectural modelling

Aim

The objectives of this step is to design the software organisation of the function to implement and to clarify the interaction with the hardware and the environment of the software.

Input artifacts

The input documents and elements of this activity are:

- a set of informal descriptions (scope of the process, description of the functions, behaviour of the system, ...)
- the API
- the functional architecture
- the data dictionary
- the requirements, allocated to function or data description, in natural language.

Output artifacts

The output shall provide :

- the input API updated and completed
- the input functional architecture updated and completed
- the input data dictionary updated and completed
- a set of new requirement which described the design choices made during this step
- traceability with system analysis output

Means

The software architecture shall be derived from the functional architecture provided during system analysis step. By the way the existing SysMl model can be completed here to describe the software architecture, then derived to a SCADE model (for exemple by using Scade system), as SCADE is the method firstly proposed to design the software.

7.1.3 Functional and behavioural modelling

Aim

The objectives of this step is

Input artifacts

The input documents and elements of this activity are:

- a set of informal descriptions (scope of the process, description of the functions, behaviour of the system, ...)
- the updated API
- the updated functional architecture
- the updated data dictionary
- the updated set of requirements, allocated to function or data description, in natural language.

Output artifacts

The output shall provide:

- a formal model of on-board unit
- the updated data dictionary
- the updated set of requirements, formalized or justified
- traceability with system analysis and architecture modelling output

Means

One of the objectives of the project is to provide a formal model of the on-board unit functionalities. The task consist to give a formal model of the functions and the requirements. First decision proposed to use the SCADE approach for this activity. Thus the task consist, from the SCADE architectural model, to complete the behaviour of the functions in each node of the model.

7.1.4 Executable software

Aim

The objectives of this step is to provide a C code which implements the applicative part of the On-board Unit as defined during the system analysis. Target of the executable code is not identify to allow its reuse in different environment. However, constraints specify in the API document shall be covered.

Input artifacts

The input documents and elements of this activity are :

- Scade Model
- Software Specification Document
- API

Output artifacts

The output shall provide an executive software:

- C code of the applicative part of the on-board unit
- Supporting documentation: at least Software Deployement Manual and Release Notes

Means

The C code can be generated from the Scade model for most of the parts. For interfaces components or for components which the code can be optimized c code shall be automatically written.

In both cases, the production of C code shall follow the constraints defined in EN50128 for SIL4 software design.

7.2 Data Model

This section describes all the data shared by the different activities during the project. These data shall be managed in a common repository which shall be the reference for specification, modelling and VnV activities.

Thanks to the choice of Eclipse platform, use of technologies available on Eclipse and XML files are the best candidates to store these items.

In the following, we give the specification of these items and the links between them. The specification of this data-model is based on an Ecore model. However it can be implemented with any means or tools, depending the decision of the project partners.

3 groups of data are defined: *Variable*, *Function* and *Requirement*; for each group, a set of *attributes* are defined to specify the group. These attributes are specified by a name, a cardinality and a type which can be a well know type (as boolean, string, integer,...) or a type defined explicitly for this model.

All these items share common attributes described in the following section. The *Features* are defined to organize the items in group. Finally the links between the groups are specified in the last section.

7.2.1 Common attributes

All the items are *named* and have mandatory common attributes:

name defined as a string and unique for each item, naming convention are defined in 7.3;

safety a boolean tag to qualified the information as safety or not ;

definition to describe in a clear way the items, at least by a textual description. Links to picture or table can be added.

Some *issues* can be associated to each items. These issues are characterized by:

description a textual explanation;

closed a boolean tag to qualified the state of the issue;

owner the person in charge of the issue.

Comment. TODO: do we need more information on issue , as author, ... ?

Do we need to link to github issue tracker ?

The figure 2 gives the Ecore model of these elements.

7.2.2 Variables

First of all we have to define the types used to specify the variables.

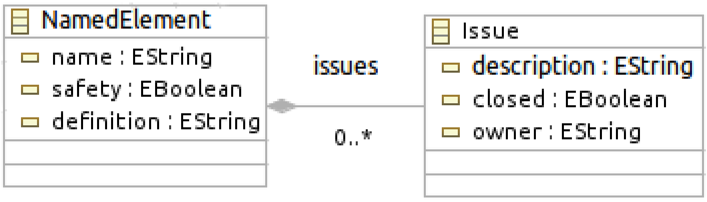


Figure 2. Common attributes for all items

In order of possible, we shall use simple and well defined, as boolean, string, integer,... (see the Iso standard for C language `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`). In the Ecore model these types appeared with the prefix "E".

Besides some new types can be added to define a complex or structured type or an enumeration.

These types are defined as *VariableType* with the following attributes:

parentType optional, in the case the type is a subtype of a already defined type (for example "Distance" can be considered as subtype of "Integer");

minimalValue optional, in the case the type is a range of integer, the minimal value;

maximalValue optional, in the case the type is a range of integer, the maximal value;

resolution optional, in the case the type is a range of integer, resolution to take into account;

The figure 3 gives the Ecore model of these elements.

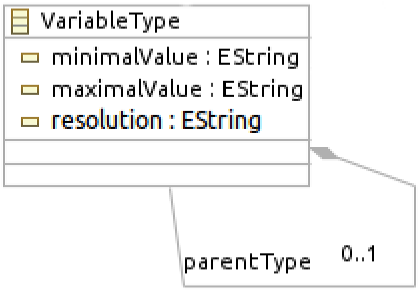


Figure 3. Type definition

The *Variable* is then defined with the following attributes:

type the variableType associated, unique;

definitionRequirements at least one requirement to define the variable;

constant a boolean tag to qualified constants;

specialValue optional, several special value can be defined;

store optional, a variable can be an element of a more complex or structured variable.

The figure 4 gives the Ecore model of these elements.

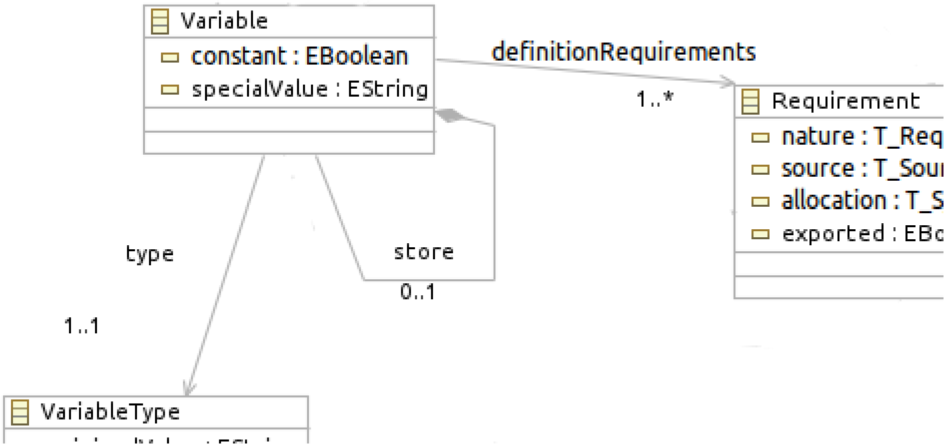


Figure 4. Variable definition

7.2.3 Functions

A *Function* is defined with the following characteristics:

allocation the subsystem on which is allocated the function (i.e. Kernel, DMI, Odometry,..);

requirement at least one requirement to describe the function;

input optional, the input variables of the function;

output optional, the output variables of the function;

internal optional, the internal variables of the function;

subFunction optional, the sub-functions which allow to structure and detailed the function.

The figure 5 gives the Ecore model of these elements.

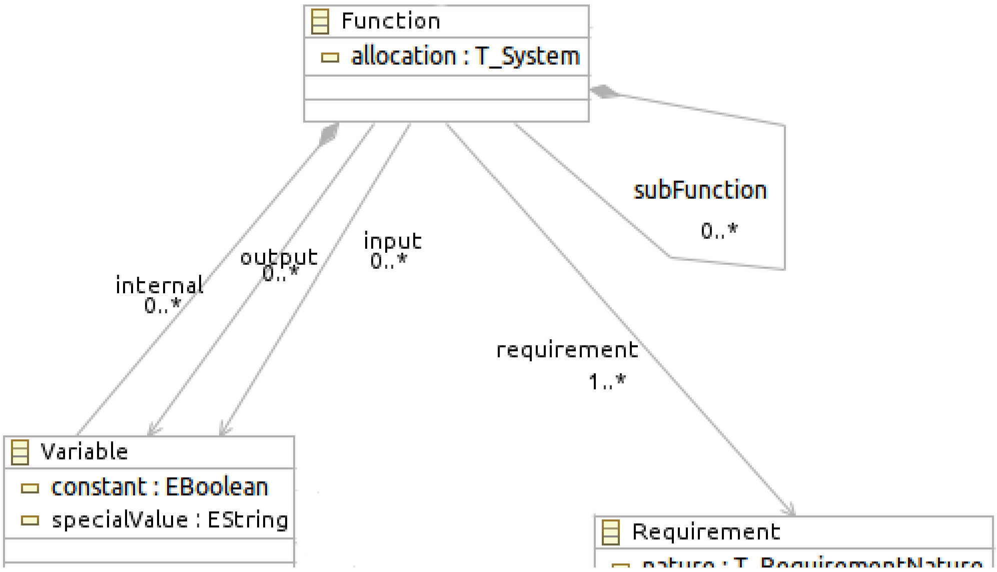


Figure 5. Function definition

7.2.4 Requirements

A *requirement* is defined by the following attributes:

nature class of property defined by the requirement: Structural, functional, definition,...

source the document or the artifact where the requirement is defined for the first time (i.e. SRS, SystemAnalysis,...);

allocation the subsystem on which is allocated the function (i.e. Kernel, DMI, Odometry,..);

exported a boolean tag to defined if the requirement shall be exported to another sub-system or function;

subRequirement optional, the sub-requirements in which the requirements is divided.

Thus the following enumerate sets shall be defined (their contents are not given in an exhaustive way here):

T_SourceDocument set of source document for defining a data, this can be an input document (SRS, a subset,...) or a document provided during the process (system analysis, model description,...)

T_RequirementNature what describe the requirement ? Structural, Functional or Definition

T_System on what subsystem is allocated the data ? Kernel, DMI, BIU,...

The figure 6 gives the Ecore model of these elements.

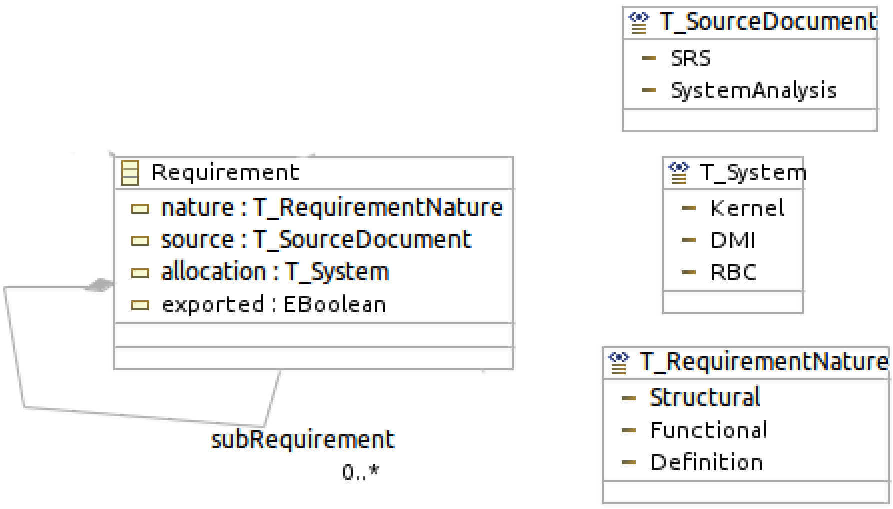


Figure 6. Requirement definition

7.2.5 Feature

A *feature* is a mean to structure the analyse of the system, it is the starting element of an analysis, and contents informal information:

description an informal and textual description of the functionality to analyse, which can contain some links to pictures or tables;

subFunctions the functions which are defined to describe the functionality.

The figure 7 gives the Ecore model of these elements.

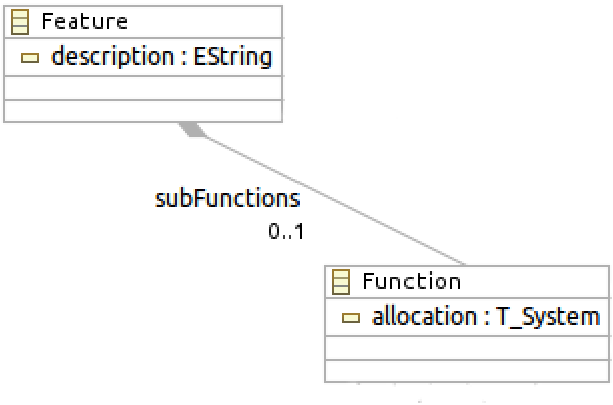


Figure 7. Feature definition

7.2.6 Links

The figure 8 gives the Ecore model of all the data model as describe above and can be read as an UML diagram.

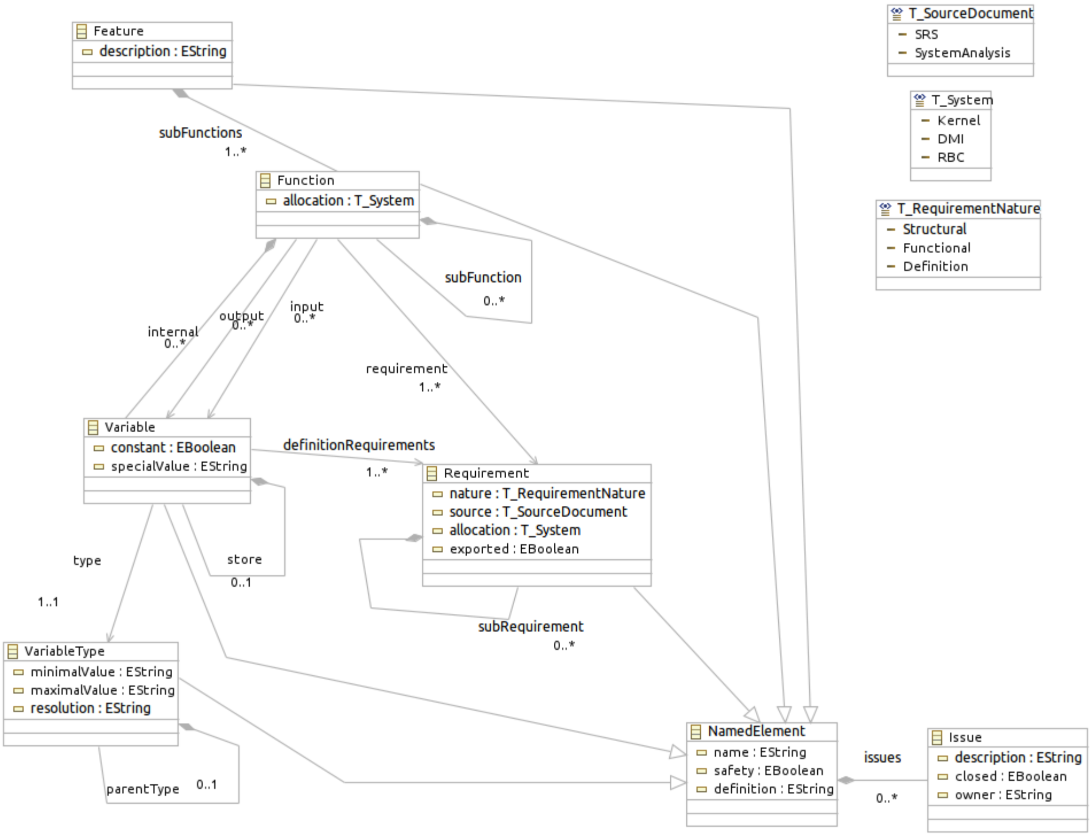


Figure 8. General Ecore model of the data dictionnary

7.3 Name convention

All the items described in section 7.2 shall be named in a unique way. This section gives some naming conventions to facilitate modelling and VnV.

In a general way, the name shall be clear, concise to facilitate the understanding of the models. UPPER case will be used for names issued form official documents as SRS, mixed of lower and UPPER case for others.

To avoid trouble during translation of models or code generation, we assume some languages can be not case sensitive: a same word can not be used twice with different cases. By the same way, classical keywords of programming language shall be avoid (for example for C language "if", "then", "void",...).

7.3.1 VariableTypes naming convention

The naming of the type should consist at least on an object.

The name shall be prefixed by "typ_".

7.3.2 Variable naming convention

The naming of the variables should consist of at least a combination of a verb and an object using the passive form (for examples: orderReceivedFromTrackside for a structure, endOfMissionIsExecuted for a boolean)

For the internal variables, i.e. those defined during the design activities to describe the state of the system and the exchanges between functions, the name shall be prefixed by "int_".

For the external variables, name shall be as close as possible to those already defined in the documentation. Typically, the naming convention of SRS §7 and 8 shall be followed for the interface with trackside (see § 7.3.2.11):

- A_ Acceleration
- D_ distance
- G_ Gradient
- L_ length
- M_ Miscellaneous
- N_ Number
- NC_ class number
- NID_ identity number
- Q_ Qualifier
- T_ time/date
- V_ Speed
- X_ Text

For the other external variables, the name will be prefixed by the short name of the interface:

- DMI_
- ODO_ for odometry
- BTM_
- RIU_
- ...

7.3.3 Feature and Function naming convention

Features and functions are named using an active form consisting of at least a combination of a verb and an object (for example: initiateTerminatingASession).

A short name, formed from first letters of each word, is defined for each feature to facilitate the naming of sub-functions or requirements (for Example MORC for ManageOfRadioCommunication).

7.3.4 Requirement naming convention

A requirement shall be identified in a unique way by is name and the source document with it version.

The naming convention of requirement depends on the source document or activity as described in the following table:

Source	Name	Description
Subset 26 v3.3.0	SRS_*number_letter*	where *number* is the number of the section as it appears in the SRS, and *letter*, optional, is used in the case the requirement is divided in sub-requirements (for example : 3.5.3.1, 3.5.3.4_a)
Subset X v Y	SubsetX_Y_*number_letter*	where X is the number of the subset, Y its version, *number* is the number of the section as it appears in the SRS, and *letter*, optional, is used in the case the requirement is divided in sub-requirements (for example : Subset_34_3.0.0_ 2.2.2.1
System Analysis	SA_*short_number*	where *short* is the shortname of the feature or the allocated subsystem, *number* is a unique number for this short name (for example SA_MORC_4 or SA_DIM_18)
Software Modeling	SW_*short_number*	where *short* is the shortname of the feature or the allocated subsystem, *number* is a unique number for this short name

7.4 Use of the ETCS Language

The ETCS Language, i. e. the definition of ETCS data types, packets and messages is specified in Subset 026, chapter 7 and 8.

These definitions reflect the data formats at external interfaces, especially at the air gap between track and train. The data formats have been defined to save bits during their transmission between balises, loop, RBC and train. The various different bit lengths lead to field borders on bit boundaries in data structures and are very unhandy to be used in modelling and implementation languages.

In addition, the ETCS language terminology talks about “variables” instead of “types”. For clarification: all “variable”, “packet” and “message” types are in fact data type definitions and should not be misunderstood as global variables. To unburden the OBU software modelling and implementation from the inconvenience of arbitrary bit lengths and bit boundaries, the following provisions should be made.

7.4.1 Transformations between native ETCS language and OBU internal data types

The ETCS language data types as specified in Subset 026, chapter 7 and 8, require an OBU software internal equivalent supported by the openETCS modeling and implementation languages.

Native ETCS language data types must be converted into OBU internal data types at the external interfaces of the OBU software. This enables the internal OBU functions to deal with internal data types only.

7.4.2 Mapping of basic data types: Native ETCS language - OBU internal data types

The transformations between ETCS data types and the corresponding OBU internal data types are

Native ETCS language types	OBU internal data types
integer 2 – 31 bit	integer32
enumeration	enumeration
integer 1 bit	boolean (if character is boolean)
integer 1 bit	enumeration (if character is enum)
physical dimensions w. int resolution	integer or fixed point
structures	structures with fields on at least byte boundaries
dynamic iterations of structures	Iterations unrolled and transformed into static arrays of structures of constant maximum size (constant defined in a central place, minimum = 0)

7.4.3 Mapping of type names

To avoid confusion by different names for internal and external data types and preserve the relationship between both, types defined in the ETCS language shall preserve their names when transformed into their corresponding OBU internal data type. The distinction between both shall be made via different name spaces and prefixes only.

7.4.4 Use of ETCS language type definitions whenever possible

In principle, data type used internally within the ETCS OBU software can be defined freely. On the other hand, the distributed development in the openETCS team could be improved by common understanding and agreement to artefacts defined by the ETCS language.

Therefore, self defined data types shall be based upon ETCS language elements (transformed to the internal equivalent) whenever possible. Type definitions in parallel to existing ETCS language terms shall be avoided.

7.4.5 Uniform scaled (physical) dimensions

Several elements of the ETCS language use combinations of scaling factors (example: Q_SCALE) and values (example: D_LINK) to quantify (physical) dimensions. Self-defined OBU internal types shall be based upon a unique scaling factor instead (example: all lengths and distances in cm).

7.4.6 Replacing dynamic data structures with static type definitions

Several structured types of the ETCS language use iterations of substructure loops of arbitrary lengths; an actual iterator value then indicates the actual length of the loop. Since such dynamic allocations are not suited for dependable and safety-related software, the dynamic structures shall be replaced by static structure arrays of a constant maximum length. “dataValid”-Flags (see next sectionh) then shall designate array elements filled with data.

Default values should be assigned to “empty” variables additionally.

7.5 Basic Concepts

7.5.1 "‘dataValid"’-Flags and "‘eventValid"’-Strobes

Since resource allocations like memory space for variables must not be performed dynamically, all resources must be allocated statically and variables like structures and arrays must be of fixed size. This requires a method to differentiate the array elements or structures filled with valuable data from "‘empty"’ ones. This shall be achieved by adding a boolean “dataValid” flag to the structures and array elements. Data flows with event character (example: a just received balise datagram) shall in the same way make use of such “valid” strobes within the data flow type definitions: a “eventValid” strobe set to true then designates the presence of an event, set to false designates the absence of an event.

Default values should be assigned to “empty” variables additionally.

7.5.2 Time Stamps

All data received via an external or internal interface (like sensors, balise antennas, ...) shall be marked with the actual system time at the earliest moment of reception.

This allows to eliminate the impact of system internal propagation delays and can be used consistency checks or for diagnosis purposes, if neccessary.

7.5.3 Execution on a Synchronously Clocked Platform

The openETCS software has to be modeled and implemented to be executed on a synchronously clocked platform or runtime system with the following characteristics.

A clock cycle is divided into exactly 3 phases:

1. Sample and hold of all inputs (input phase)
2. Calculate all outputs (execution phase)
3. Emit all outputs (output phase)

The execution order of subfunctions results from the input-output relationships between the subfunctions. There are no racing conditions between subfunctions.

Whithin one clock cycle, the system time (as all other inputs to the OBU software) is kept unchanged for the OBU software. The processing time for all 3 phases is thought to be 0 (practically, it will be > 0, but this is irrelevant as long as the processing time does not exceed the clock period).

8 SysML approach with Papyrus

8.1 Introduction to SysML with Papyrus

SysML is a graphical modeling language based on UML2 with extensions to:

- Supports the specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities.
- Supports model and data interchange via XMI.

OMG specification[1] describe a visual modeling language that provides both Semantics and Notation but *do not describe neither a tool nor a method.* SysML is based on *BLocks* representing concepts -e.g. pieces types, functions- and how these are related (links, compositions), internally arranged (using Parts typed by other blocks), or acting (describing internal behaviors). The aim is to describe a system -possibly build by composing subsystems- and also it surrounding, requirements, behaviors and so on[2].

Papyrus is an Eclipse Open source project[3] held by CEA List the propose a modeler and toolkit for UML and SysML model creation and use[4]. Development team focus on bringing the tool uncolored by any methodology in order to respect the ideal of OMG specification and leave to users the widest choice of usage.

8.2 SysML in the project

The proposal is to use SysML for the highest level of modelling:

Conception and design Aim of the SysML model is to manage the gap between prose system analysis and formal model dedicated to the design of on board unit. Thus the SysML model will provide a high level model associated to the system analysis, the SSRS and the API:

- Structure of the system, with physical and functional architecture will be described with *Block Definition Diagrams* and *Package Diagrams*
- Logical interfaces between subsystems and between functions will be described with *Internal Block Diagrams*
- Data definition will be defined with *Block Definition Diagrams*
- Requirements will be defined and allocated with *Requirements Diagrams*
- High level Behaviour description will be described with *State Machine Diagrams* and *Sequence Diagrams*

Safety analysis SysML model will provide an organic and a functional architecture of the system

- Functional Breakdown Structure will be defined with *Block Definition Diagrams* and *Internal Block Diagrams*

[1] UML Specification website: http://www.omg.org/spec/UML/,
SysML Specification website: http://www.omg.org/spec/SysML/

[2] Open ECTS Braunschweig workshop tutorial on SysML:
https://github.com/openETCS/toolchain/blob/master/ToolDescription/Papyrus/Tuto_SysML_basic_short_2013_10_8_Braunschweig.pdf

[3] Eclipse Papyrus project website

[4] Open ECTS Braunschweig workshop tutorial on Papyrus:
https://github.com/openETCS/toolchain/blob/master/ToolDescription/Papyrus/Tuto_Papyrus_basic_short_2013_10_8_Braunschweig.pdf

Verification and Validation SysML model will provide expected behavior of the system during execution:

- Test Cases and execution traces will be defined with *Sequence Diagrams*

This list can be completed, after the results of benchmark of secondary tools, either by defining use of some diagrams for other activities (for example for safety or VnV) or by defining methods and tools to use to complete SysML (for example for requirements management or database definition).

It is considered to use SysML/UML stereotype mechanism to refine the semantics of the defined model, especially for *Conception and design* phase where for example some blocks could be tagged as Function, System or Data. See issue number 25 of requirements repository for details[5]. Currently no specific mechanism has been defined.

8.3 Selection of used diagrams

SysML diagrams that can be used for modeling are:

- *Package Diagram* : Organization description
- *Block Definition Diagram* (BDD): Structure description
- *Internal Block Diagram* (IBD): Structure description
- *Sequence Diagram* : Specification on sequence of functions, timing constraints, Test case, execution trace, counter example, etc.
- *State Machine Diagram* : Behaviour description
- *Requirement Diagram* : Requirements description

The following other SysML diagrams *cannot* be used:

- *Parametric Diagram*
- *Activity Diagram*
- *Use Case Diagram*

8.3.1 Remarks for all diagrams

Following elements are allowed on all kinds of diagrams:

- Comment Note

[5] `https://github.com/openETCS/requirements/issues/25`

8.3.2 Notes on selected diagrams

The diagrams and diagram elements of SysML have been chosen because (1) they are supported by Papyrus v0.10.0 used in Eclipse Kepler release and because (2) they are the minimal subset of SysML needed for a proper system description and V&V activities.

In the future, we can decide to augment the chosen SysML subset because Papyrus offers new capabilities and because new elements are needed. This can been done at project level following a procedure which is not yet decided.

8.4 Restrictions on Package Diagram

Note: The naming of the nodes and paths of this section and the following one is the same as Annex A of book “A Practical Guide to SysML”.

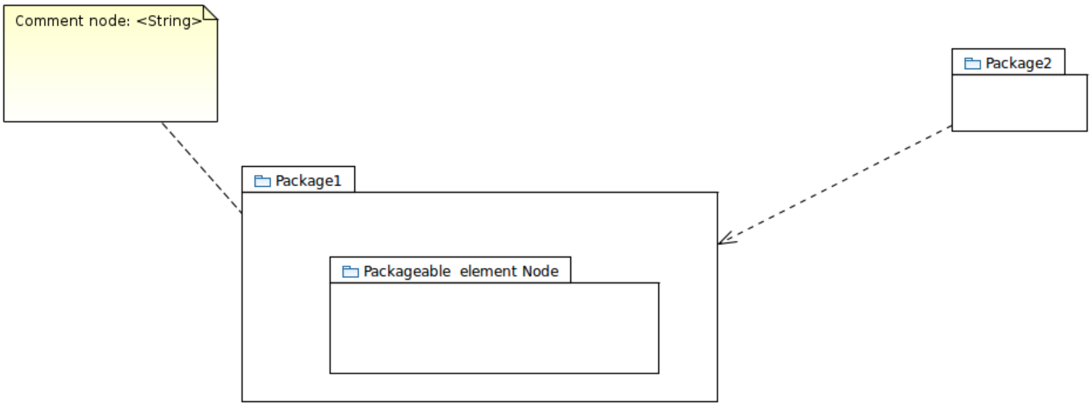


Figure 9. Package Diagram

The following Package Diagram nodes and paths can be used for modelling:

- Comment Note
- Package Node
- Packageable Element Node
- Import Path

The following items *cannot* be used:

- Model Node
- View Node
- Viewpoint Node
- Containenement Path
- Dependency Path
- Conform Path
- Metamodel Node

- Metaclass Node
- Model Library Node
- Stereotype Node
- Profile Node
- Generalization Path
- Extension Path
- Associatin Path
- Reference Path
- Profile Application Path

8.5 Restrictions on Block Definition Diagram

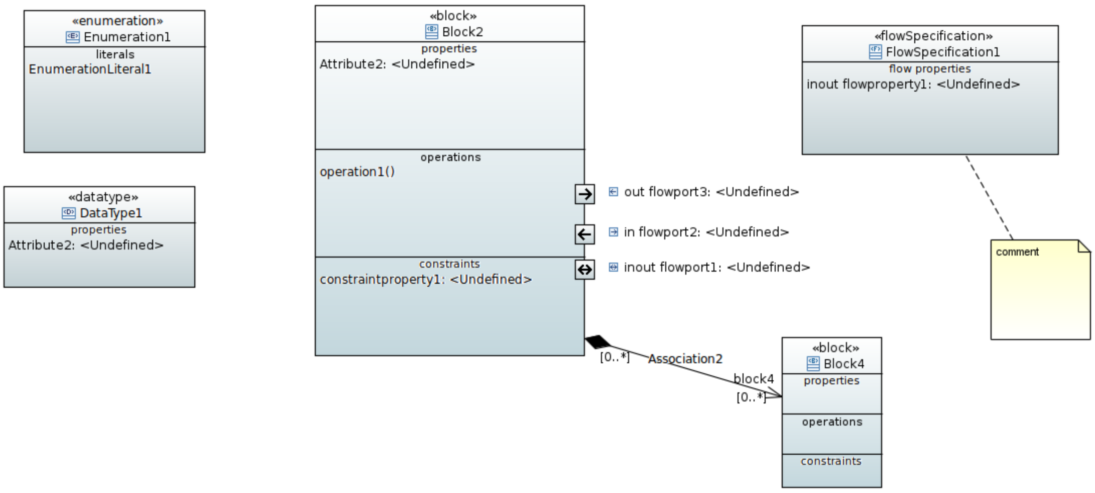


Figure 10. Block Description Diagram

The following Block Definition Diagram nodes and paths can be used for modelling:

- Block Node
- Enumeration Node and enumeration litteral
- Block Node with “datatype” Stereotype
- Composite Association Path
- Flow Specification Node and Flow Properties
- Primitive Type Node
- Property Node
- Constraint Node
- Requirement Node

The following items *cannot* be used:

- Quantity Kind and Unit Nodes
- Value Type Node
- Actor Node
- Interface Block Node (SysML 1.3)
- Interface Node
- Signal Node
- Interface Compartments for Block Node
- Reference Association Path
- Association Block Path and Node
- Generalization Path
- Full Port Node
- Proxy Port Node
- Proxy Port Node With Interfaces
- Port Compartments for Block Node
- Nonatomic Flow Port Node
- Block Node with Constraint Compartment
- Constraint Block Node
- Activity Node
- Activity Composition Path
- Object Node Composition Path
- Instance Specification Node
- Association Instance Specification (Link) Path

8.6 Restrictions on Internal Block Diagram

The following Internal Block Diagram nodes and paths can be used for modelling:

- Part Node Part
- Connector Path
- Atomic Flow Port Node

The following items *cannot* be used:

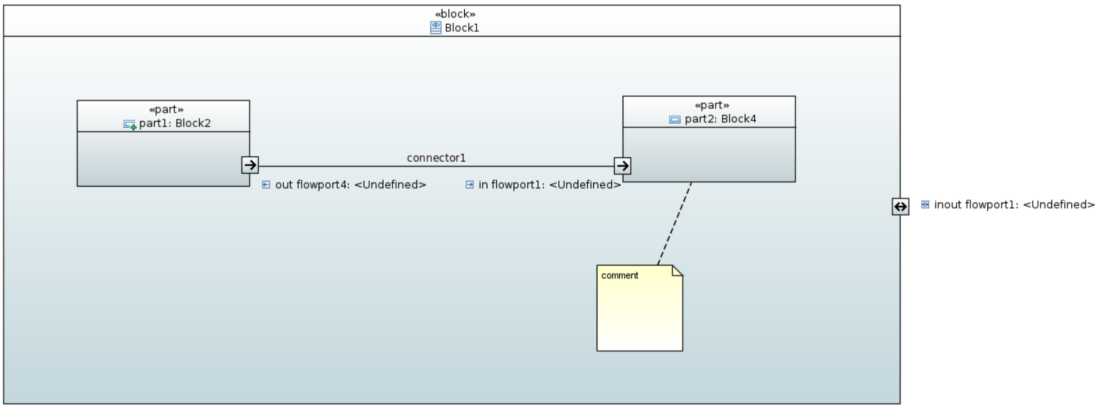

«block»
Block1
«part»
part1: Block2
«part»
part2: Block4
connector1
out flowport4: <Undefined>
in flowport1: <Undefined>
inout flowport1: <Undefined>
comment

Figure 11. Internal Block Diagram

- Actor Part Node
- Reference Node
- Participant Property Node
- Value Property Node
- Connector Property Path and Node
- Item Flow Node

8.7 Restrictions on Sequence Diagram

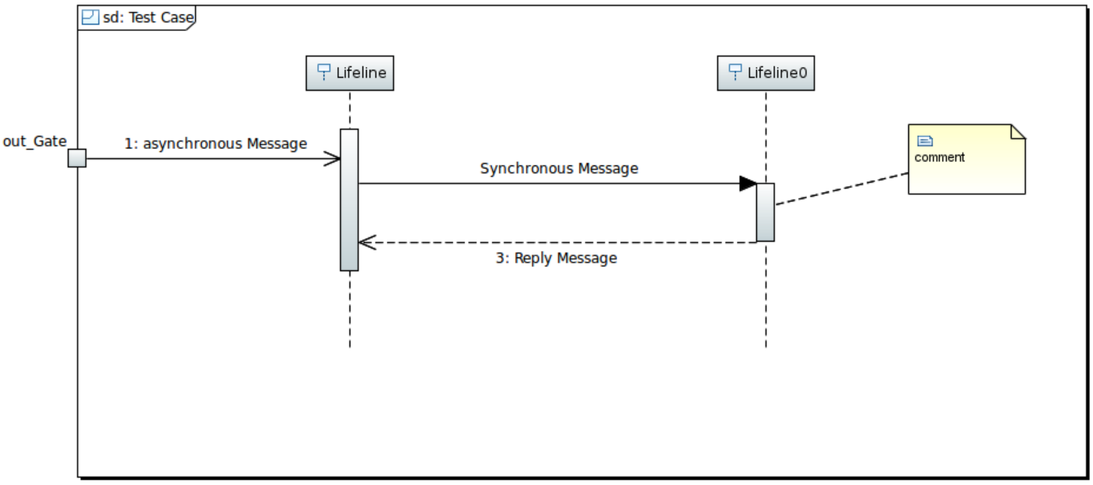


Figure 12. Sequence Diagram

The following Sequence Diagram nodes and paths can be used for modelling:

- Lifeline Node
- Synchronous Message
- Asynchronous Message
- Reply Message

- Duration Observation Symbol &d
- Duration Constraint Symbol {d}
- Time Observation Symbol @t
- Time Constraint Symbol {t}

The following items *cannot* be used:

- State Invariant Symbol
- Single-compartment Fragment Node
- Multi-compartment Fragment Node
- Filtering Fragment Node
- Interaction Use Node
- Lost Message Path
- Found Message Path
- Activation Node
- Create Message Path
- Destroy Event Node
- Coregion Symbol

8.8 Restrictions on State Machine Diagram

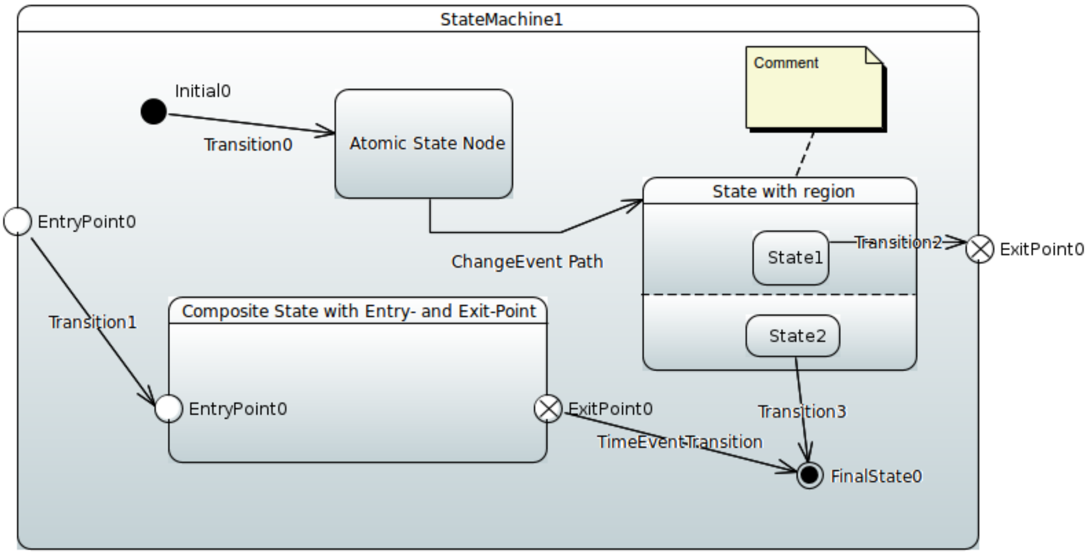


Figure 13. State Machine Diagram

The following State Machine Diagram nodes and paths can be used for modelling:

- State Machine with Entry- and Exit-Point Pseudostate Nodes
- Atomic State Node
- Composite State with Entry- and Exit-Point Pseudostate Nodes
- Composite State Node with Multiple Region
- Initial Pseudostate Node
- Final State Node
- Time Event Transition Path
- Change Event Transition Path
- Constraint Node (for transition guard)
- Constraint Path between state and constraint
- Constraint Path between transition and constraint
- Requirement Node (to link some diagram parts to a requirement)
- Realize Path between state and requirement
- Realize Path between constraint and requirement

Simple and composite states may contain internal activities compartments:

- Entry actions may contain simple C-Assignments, timer resets or operation calls;
- Exit actions may contain simple C-Assignments, timer resets or operation calls;
- Do actions may contain simple C-Assignments, timer resets or operation calls.

The following items *cannot* be used:

- Sub-State Machine Node with Connection Points
- Terminate Pseudostate Node
- Choice Pseudostate Node
- Junction Pseudostate Node
- Trigger Node
- Action Node
- Send Signal Node
- Join Pseudostate Node
- Frok Pseudostate Node
- History Pseudostate Node
- Signal Event Transition Path
- Call Event Transition Path

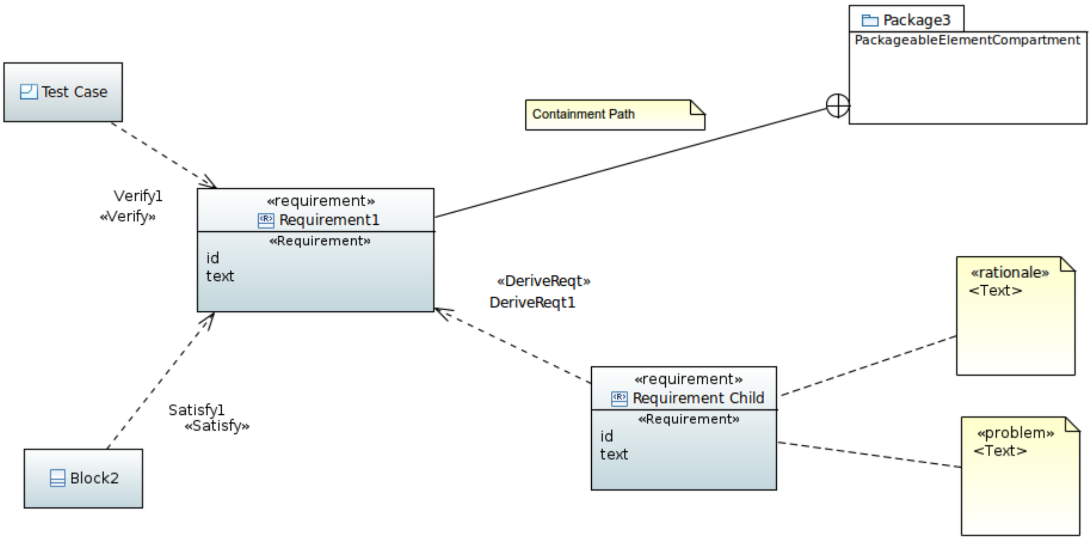


Figure 14. Requirement Diagram

8.9 Restrictions on Requirement Diagram

The following Requirement Diagram nodes and paths can be used for modelling:

- Requirement Node
- Package Node
- Containment Path
- Derivation Path
- Satisfaction Path
- Verification Path
- Rationale Callout
- Problem Callout

The following items *cannot* be used:

- Requirement Related-Type Node
- Trace Compartment
- Test Case Node
- Refinement Path
- Trace Path
- Copy Path
- Trace Callout
- Derivation Callout

- Verification Callout
- Statisfaction Callout
- Refinement Callout
- Master Requirement Callout

8.10 Model patterns

For *Modelling* activity, the modeler shall only use:

- *Package Diagram*
- *Block Definition Diagram* (BDD)
- *Internal Block Diagram* (IBD)
- *State Machine Diagram*
- *Requirement Diagram*

For *V&V* activities, one shall only use:

- *Package Diagram*
- *Block Definition Diagram* (BDD)
- *Internal Block Diagram* (IBD)
- *Sequence Diagram*
- *Requirement Diagram*

The following guidelines should be followed:

- Each block must either have an associated state machine diagram (with the same name).
- Guard conditions contain Boolean conditions over variables and timer;
- Actions are sequences of C assignments, timer resets or operation calls;
- Initial state of a State Machine has only one out transition without guard condition or action.

Comment. FIXME: Example of patterns to use.

9 Approaches for low level design

Comment. to complete later: SCADE ? OS approaches ?

9.1 SCADE

9.2 C Code

9.3 Alternative open-source approaches

Comment. to complete later depending the choices

References