Verification and validation of B models

Benoît Lucet
Systerel

November 19, 2013

Abstract

This document describes the verification and validation processes applicable to B models. It supports B as a promising method in the context of openETCS, given the maturity and level of rigor associated with it. After an introductory section where we discuss the subject of validation, section 2 presents the different verification processes and discusses their advantages and drawbacks, while section 3 presents the tools available to perform such verification. Finally, appendix A presents the application of the verification process to an existing B example: the procedure on-sight.

Contents

1 Introduction

A B model is a textual and formal specification covering the functional behaviour of a safety critical software. It is usually written based on a high-level specification (informal or formal specification, for example SysML or a natural language). It is gradually refined, starting at the top with an abstract notation and ending at the bottom with sequential instructions — which are then auomatically translated in a target language such as C or Ada.
Thus, we define three objects of verification and validation: the specification, the B model and the generated source code.

Validation consists of:

- guaranteeing the functional adequacy between the specification and the model (this can be achieved, for example, through review and proofreading),
- building a test environment around the generated source code and test it.

Hence, this document will mainly focus on verification, i.e. the methods and tools required to assure that B method is a consistent way of producing critical software.

2 Verification processes applicable to a B model

In this section, we demonstrate the suitability of the B method towards the problematics encountered in the openETCS project by introducing the different verification processes applicable to a B model.
Each of the verification process is presented and its contributions to the system security and consistency are discussed.

2.1 Type checking

Static type checking (TC) is a basic form of program verification that ensures type safety of the model. It is the first verification — after lexical and syntactic analysis — to be performed on a B model, thus allowing an early detection of problems. It is also a pre-requisite for the higher-level verifications.
Strong typing ensures a consistent use of data and is essential to writing correct formulae (predicates or expressions).

The type checking process consists of two main activities: data typing and type verification.
Data typing is the activity of assigning a type to newly-encountered data in a predicate or a substitution[1]. It is based on an inference mechanism, which is

[1]In B, a substitution is comparable to a set of instructions that modify variables. For more information, see [1].

able to deduce the type of a newly-encountered variable from the type of the other variables intervening in the predicate or the substitution, and specific inference rules.
On the other hand, *type verification* is the activity of verifying typing rules between already-typed variables. These rules are specific to their applying predicates, expressions or substitutions.

2.2 B0-check

The B0 verification has the specific purpose of checking the respect of the rules that the B model has to conform to in order to generate the translation to C or ADA. These rules are called implementability rules and must be respected in order for the translation to process properly. They also ensure that the resulting code is executable and respects a set of properties.

2.3 Well-definedness

An expression is well-defined (WD) if its associated value or interpretation exists and is unique, thus avoiding ambiguity.
Examples of ill-defined formulae include division by zero, function application to an argument out of the domain, function application of a relation etc.

Well-definedness checking is thus an extra verification that helps strenghten the model.

2.4 Model checking

Model checking is a static semantic check that searches for invariant or assertion violations and deadlock states.
This type of verification animates the model, modifying the current state of the machine, starting from the initialization. Operation calls are simulated and modify the internal state which is then checked for various properties. Most of the time, an invariant or assertion violation is looked for.

This verification process, as opposed to the ones previously introduced, considers the semantics of the model and aims at verifying properties dynamically. However, it has its limitations:

- unability to run through all of the states and transitions,
- difficulty to deal with infinite sets.

This means that potential erroneous states can be missed, and that this verification process is not sufficient to ensure *correctness*, though satisfying as an additional verification tool.

2.5 Constraint-based checking

Constraint-based checking (CBC) is the process of finding a given valid state, for which an operation call leads to an erroneous state. This is done by constraint solving instead of — as seen for model checking — running through states from the initialization.
This technique will usually provide more counter-examples than model-checking, because it ignores the initialization constraints and can thus reach a wider range of states.

2.6 Formal proof

A proof obligation is a mathematical lemma generated by the proof obligation generator (POG). It corresponds to a consistency property of the model, that has to be demonstrated. A fully-proved model is said to be *correct*, in the sense that every property (invariant, assertion) expressed is proved to hold for every state of the program. If a proof obligation is not provable, it means that the B model is inconsistent and must be corrected. In fact, the goal of any B development is to obtain a proved model.
In contrast to model-checking, formal proof does not require to make assumptions about the size of the system (number of transitions). It is reliable and powerful, but it needs to be taken into account that:

- some proofs can be very difficult to solve,
- the model needs to be written as to make it the simplest to prove, which demands experience and skill.

A proved model will always meet the safety and security qualifications ; however that doesn't mean it will behave in regards to the specification! This is the domain of validation, as discussed in the introduction.

3 Tools for verification

3.1 Existing tools

In this section, we present the existing tools suitable for the verification processes defined in section 2.

3.1.1 Atelier B

Atelier B[2] is the main development tool associated with the B method and is produced and distributed by ClearSy. It provides most of the needed tools.

B compiler
The B compiler performs syntax analysis, type checking, identifier scope resolution and B0-checking. It is part of Atelier B as an open source tool.

[2]See `http://www.atelierb.eu/en`

Proof obligation generator
Atelier B's POG is currently the only known fully operational POG for B, and is free of charge — although proprietary software, which means closed-source. ClearSy is currently working on a new proof obligation generator ; whether it will be open source or not is to be determined.

Prover, proof assistant, user-defined rules
Atelier B provides a free of charge — although not open source — prover which discharges proof obligations. Depending on the complexity of the model, a varying proportion of the proof obligations is discharged automatically.
For the remaining proof obligations, Atelier B provides an interactive proof assistant allowing the user to guide the prover in discharging the PO. The user may define theories (or rules) which have in turn to be proved. The user-defined rules are organized in a database.

Atelier B translators
Translators are an essential component of the industrial success of B. The translators take the B0 implementations as input and produce a target source code, typically Ada or C/C++, ready to be compiled or integrated in an environment. ClearSy provides an open source translator, but it does not reach the T3 level of qualification[3].

3.1.2 ProB

ProB is an animator and model checker for B models distributed under the EPL license (open source) and mainly developed by Formal Mind[4].
It performs model checking as well as constraint-based checking and searches for a range of errors, with customizable search options and various graphical views. ProB also handles automatic coverage reports generation.
ProB is a mature tool and is being used by several industrials such as Siemens and Alstom. This makes it a precious tool for the verification processes described above.

3.2 Tool qualification

Atelier B has been used for many years to develop railway critical software. It is, for this exact reason, *qualified* by the main actors of the railway domain: SNCF, RATP, Alstom, Siemens etc.

The CENELEC norm defines qualification levels for verification tools. Annex A 5 of the norm specifies several verification techniques and for each of them, a recommendation level (mandatory, higly recommended, recommended). Below are listed the different techniques and measures along with their recommendation levels:

[3]For additional information on qualification, see subsection 3.2 or the CENELEC norm.
[4]See `http://www.formalmind.com`

1. Formal Proof (HR)
2. Static Analysis (HR)
3. Dynamic Analysis and Testing (HR)
4. Metrics (R)
5. Traceability (M)
6. Software Error Effect Analysis (HR)
7. Test Coverage for code (HR)
8. Functional/ Black-box Testing (M)
9. Performance testing (HR)
10. Interface testing (HR)

Table 1 shows, for each of the verification processes presented in 2 (specification and source code were added), the corresponding item in the CENELEC norm annex table.

	spec	TC	B0C	MC	CBC	proof	source code
A 5.1						✓	
A 5.2		✓	✓		✓		
A 5.3				✓			
A 5.4							
A 5.5	✓						
A 5.6							
A 5.7							
A 5.8							✓
A 5.9							
A 5.10							

Table 1: Correspondence between CENELEC norm recommendations and the presented verification processes

3.3 Conclusion on tools

Table 2 summarizes the presentation of the tools in subsections 3.1.1 and 3.1.2. Atelier B and ProB are both mature tools that have proved their worth. They are the core tools for validation processes of B models. However, key components of Atelier B are not open source and this issue is not completely compensated by ProB's model checking and CBC.
An ongoing research project named BWare[5] and conducted by ClearSy, Inria,

[5]See `http://bware.lri.fr/index.php/BWare_project`

LRI and others aims at providing a framework from proof obligation generation to proof discharge by the means of SMT solvers. This promising project started in September 2012 and is funded for a period of four years. It opens perspectives for the near future in terms of open source B model verification.

	TC	B0	model check.	CBC	proof
B Compiler	✓	✓			
POG and provers					✓
ProB			✓	✓	

Table 2: Comparison of the tools available for B verification processes

4 Conclusion

The B method, along with its verification processes and tools, meets the goals and activities of the openETCS project in terms of quality, rigor, safety and credibility.
There is yet to develop open-source POG and build a framework for proving, but this is compensated by the fact that work on the subject is ongoing, and ProB is an effective tool for verification.

A Application: Procedure On-Sight

The Procedure On-Sight, as described in *System Requirements Specification, Chapter 5*, has been modelled in B[6] to show the feasibility of the task and the credibility of the method. This appendix briefly presents the model, then applies the verification processes to this example.

A.1 Presentation of the B model

As shown in figure 1, the model is split into three main processing modules, one of which corresponds to the actual on-sight procedure, and the two others being used as data conditioning:

- `os_mode_level`: determines the ETCS level and the mode. Contains the on-sight procedure algorithm,
- `os_consist`: checks data consistency, provides adaptation to the current ETCS level (BTM or radio),
- `os_train_info`: elaborates the location and the speed of the train.

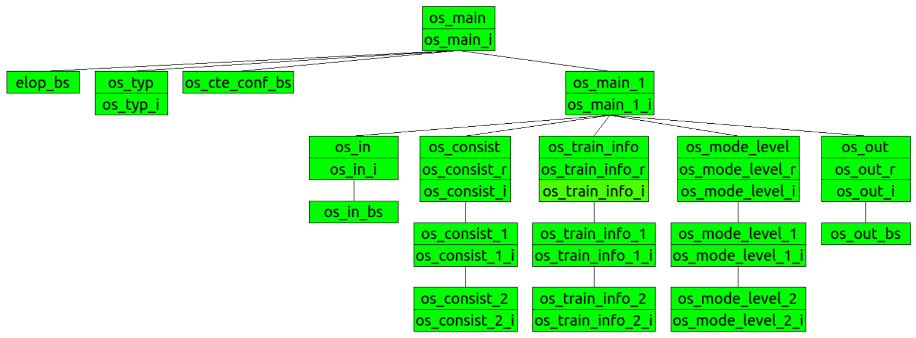


Figure 1: Architecture of the B model for the Procedure On-Sight example

These three modules are imported by the main sequencer, `os_main_1`, which calls their respective operations. The main sequencer also imports the input module `os_in`, and the output module `os_out`.
The typing machine, `os_typ`, and the constant machine, `os_cte_conf_bs`, are both imported by `os_main`, the entry point of the software, which also imports `os_main_1`.

[6]The model is available at `github.com/openETCS/validation/tree/master/VnVUserStories/VnVUserStorySysterel/02-DAS2V/c-ClassicalBModel/ProcedureOnSight`.

This “IMPORTS”-based vertical layout is complemented by a horizontal aspect: the “SEES” clause, which enables a component to access another component’s data. It is possible for a component to see the components to its left, but not to its right. Thus, a cycle-free graph is maintained.

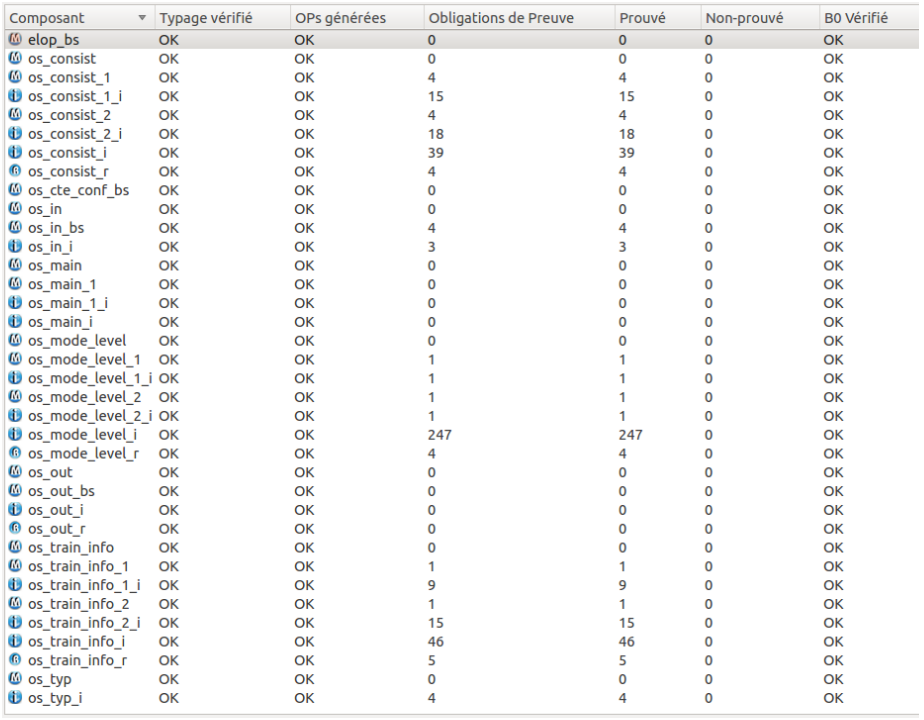
A.2 Model checking results

ProB has been used on the example model and has shown through model checking that no invariant was violated, and no deadlock state was found. However, for some machines, only a minority of states and nodes have been visited (because of infinite sets in particular) and thus no formal conclusion can be drawn. Additionally, constraint-based checking has also been run and stated, for every operation of every abstract machine, the non-violation of the invariant.

A.3 Formal proof results

A.3.1 Project status

Project status illustrated in figure 2 shows the fully-proved model in Atelier B.



Composant	Typage vérifié	OPs générées	Obligations de Preuve	Prouvé	Non-prouvé	B0 Vérifié
elop_bs	OK	OK	0	0	0	OK
os_consist	OK	OK	0	0	0	OK
os_consist_1	OK	OK	4	4	0	OK
os_consist_1_i	OK	OK	15	15	0	OK
os_consist_2	OK	OK	4	4	0	OK
os_consist_2_i	OK	OK	18	18	0	OK
os_consist_i	OK	OK	39	39	0	OK
os_consist_r	OK	OK	4	4	0	OK
os_cte_conf_bs	OK	OK	0	0	0	OK
os_in	OK	OK	0	0	0	OK
os_in_bs	OK	OK	4	4	0	OK
os_in_i	OK	OK	3	3	0	OK
os_main	OK	OK	0	0	0	OK
os_main_1	OK	OK	0	0	0	OK
os_main_1_i	OK	OK	0	0	0	OK
os_main_i	OK	OK	0	0	0	OK
os_mode_level	OK	OK	0	0	0	OK
os_mode_level_1	OK	OK	1	1	0	OK
os_mode_level_1_i	OK	OK	1	1	0	OK
os_mode_level_2	OK	OK	1	1	0	OK
os_mode_level_2_i	OK	OK	1	1	0	OK
os_mode_level_i	OK	OK	247	247	0	OK
os_mode_level_r	OK	OK	4	4	0	OK
os_out	OK	OK	0	0	0	OK
os_out_bs	OK	OK	0	0	0	OK
os_out_i	OK	OK	0	0	0	OK
os_out_r	OK	OK	0	0	0	OK
os_train_info	OK	OK	0	0	0	OK
os_train_info_1	OK	OK	1	1	0	OK
os_train_info_1_i	OK	OK	9	9	0	OK
os_train_info_2	OK	OK	1	1	0	OK
os_train_info_2_i	OK	OK	15	15	0	OK
os_train_info_i	OK	OK	46	46	0	OK
os_train_info_r	OK	OK	5	5	0	OK
os_typ	OK	OK	0	0	0	OK
os_typ_i	OK	OK	4	4	0	OK

Figure 2: Overview of the B model in Atelier B, showing type check, B0 check and proof status

This model was proved almost entirely automatically, using the provers with force 0 and force 1. Proving the model results in the certainty of its correctness. In this case, only typing invariants and constraints were expressed, because strong links between the variables do not exist in the example.

A.3.2 User-defined theories

When automatic proof fails, the user must provide a manual proof and uses theories for this purpose. Theories are rules that are used to discharge specific goals.
In this example, the only module for which interactive proof was required is `os_consist_i`. Below is presented a very simple theory (among several others) used for the proof of this component:

$$a < 0 \wedge 0 \leq b \wedge 0 < c \Rightarrow a \leq \frac{b}{c} \tag{User theory 1}$$

This theory is automatically verified by Atelier B and therefore ensures full consistency of the proof.

References

[1] Jean-Raymond Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge, 2005.