Using Formal Proof and B Method at System Level for Industrial Projects

Denis Sabatier(✉)

ClearSy, Aix-en-Provence, France
denis.sabatier@clearsy.com

Abstract. Since several years, ClearSy has driven large projects about using formal proofs at system level in the railway domain. The fundamental goal in these projects is to extract the rigorous reasoning establishing that the considered system ensures its requested properties, and to assert that this reasoning is correct and fully expressed. In this paper, we give feedback about the methodology used in all these projects, about the differences made by whether the concerned system is currently under design or already existing and about the benefits obtained. The formal proofs are performed using Event-B, with the Atelier-B toolkit.

Keywords: System level proof · Formal methods · Event-B · Atelier-B

1 Introduction

Since several years, ClearSy has driven large projects about using formal proofs at system level for railway systems in the railway domain. The fundamental goal in these projects is to *extract the rigorous reasoning* establishing that the considered system ensures its requested properties, and to *assert that this reasoning is correct* and fully expressed. At system level, this rigorous reasoning involves the properties of different kind of subsystems (from computer subsystems to operational procedures), that the formal proof shall all encompass.

It may seem that such a top-level reasoning should be very complicated, involving all details of the complex system: in reality it is often quite simple. In the case of a CBTC system (most of our system level proofs are about CBTCs), the requested properties are ensured because equipped trains determine their position correctly, because protection envelopes are determined using these correct positions, because the interlocking does so that those envelopes remain within locked routes and because equipped trains remain before their given limits. Well, it does not completely fit in a single sentence of course, but nevertheless it is fortunately simple enough to be expressed, at least independently from technical details below like track data format.

2 Role and Benefits

So contrary (and in complement to) many other methods, we start from the top level before details appear. What are the expected benefits? The system level "reasons why" properties are part of the domain knowledge. They are, however, known by domain

T. Lecomte et al. (Eds.): RSSRail 2016, LNCS 9707, pp. 20–31, 2016.
DOI: 10.1007/978-3-319-33951-1_2

experts in terms of complex design solutions, so the experts may not easily give the true explanations without going through all the design details. This drives the focus away from what actually ensures the properties and levels out all details as if they were of equal importance. Extracting these "reasons why" in a formal way forces one to formulate a self-sufficient and provable set of properties and assumptions: unnecessary details are eliminated thanks to assumptions optimization, and the proof ensures that all that is necessary has indeed been included.

Having a clear definition of these properties and assumptions will certainly not find hidden bugs in low level design: providing proofs at high level does not suppress the necessity to cover all the project's stages. At least we can expect that a clearer definition of what each sub-part should ensure (highlighted for important properties alone) will draw the testing /verifications toward important bugs or avoid them in the first place thanks to proper developers' focus. More precisely, we can summarize the benefits in two categories:

- Benefits thanks to a sufficient set of clear properties requested from developed sub-systems: better focus on those properties, either during development /testing (if these tasks occur during or later system level proofs) or for post verifications;
- Benefits thanks to a sufficient set of clear properties requested from context sub-systems: usually such properties or assumptions about context concern very different domains. In an industrial project each partner or company is eager to focus on its domain of expertise, even when doing so means using unclear or not well defined assumptions about context sub-systems. Important pitfalls can hide there, and they could remain undetected until the final integration phase. The system level formal proof typically helps to detect concerns that might fall in such "responsibility holes" otherwise.

3 Projects

The first project of that sort done by ClearSy is the system formal verification for the CBTC of New York subway line 7 (Flushing line system proof, see [4, 8]). New York City Transit (NYCT) has awarded THALES Toronto for the design and fitting of this CBTC (awarded in 2010, revenue service scheduled in 2016). This CBTC system is composed of an on-board computer fitted in renewed R142 cars and of field and office equipment (zone controllers and central supervision). It interfaces to the existing interlocking system, adapted with specific modifications. This system level formal verification with proofs lasted from November 2010 until December 2012 and the workload was several man years.

The second project was also for NYCT. Reusing the models developed for the Flushing line, the goal was to provide system level proofs for all CBTC complying with NYCT's Interoperability Interface Specifications (I2S). In fact, I2S based CBTCs are divided in subsystems with predefined roles: by clearly defining the required properties of each subsystem and the context assumptions a formal proof of system level properties is possible, even without knowing each possible vendor's design. ClearSy obtained such

a proof; the project lasted from November 2013 to July 2015 with a workload less than half that of Flushing, thanks to reuse.

Two other projects of similar sizes and topics are currently going on concerning French railways: one for SNCF has started last summer (2015), the second one for the Paris metro operator RATP is about to start.

4 Functional or Safety Properties

Should the system proof include functional properties or be restricted to safety properties only? This question may seem straightforward, because proving as many things as possible seems natural. A closer look reveals that the proof of functional properties is often expensive or with limited benefits, even if proving only safety properties means that the proof will not give any guarantee about the system being functional at all.

Here is why functional properties are less accessible to proofs: the mechanism of a formal demonstration consists in formulating target properties, gathering assumptions and finding a logical path (i.e. using only known rules) from assumptions to target properties. But functional properties are defined in terms of performances (delays, capacity) and scenarios (typical situations where performance should be reached; in degraded situations maximum performance is not expected). So the assumptions here would be the scenarios themselves, together with all the mechanisms involved; any proof there becomes very similar to a simulation on a particular case.

Conversely safety properties should be kept whatever the scenario, using minimal assumptions about mechanisms: there a proof gives all its value by replacing testing or simulation across an unbounded set of scenarios and situations. In our case of system level proofs, we thus target global safety properties only. Of course there are cases where the notions of "safety" and "functional" merge (think for example of systems aboard planes), where keeping something functional becomes vital. Then the necessary redundancies multiply the scenarios, bringing back the value of a proof for such cases.

In railway systems and in particular in CBTCs, the safety properties remain well separated from functional ones. Typical target safety properties are:

- Impossibility of collision between trains,
- Impossibility of derailment over an unlocked switch,
- Impossibility of over-speeding.

These safety properties are found in all kind of railway systems, whether they are CTBC projects or signaling system projects. In ClearSy's system level proof projects, the top level reasoning is thus very similar despite the fact that the projects are from different contexts and different designs; however similarities decrease rapidly at more detailed levels, where the chosen design has a stronger importance.

This difference between safety properties and functional properties gives an important clue about the role of a proof in a large system project, in particular the role of a system level proof: it addresses the global safety properties that should hold whatever the scenarios and whatever the conditions. The more a property is related only to specific scenarios, the less the benefits of a proof: for such properties simulation and testing apply.

5 Methodology

5.1 Overview and Experiences

ClearSy's methodology to obtain such system level proofs is divided in two main steps:

1. Write documents explaining how the system ensures the desired properties. We call this natural language "proof", with quotes because this step is not yet a formal proof.
2. Write event-B models such that the proof performed is the formal equivalent of the natural language proof.

The first step is based on the fact that we do not use the formal method to understand why a property is ensured, but to validate that it is really ensured once the "why" is understood. We want to avoid mixing formal notation issues with domain issues, which are paramount. The second step is of course necessary to obtain a true formal proof and the correctness guarantee that comes with it. These two steps are duplicated for all topics, starting from top level properties and repeated for sub-properties down to the chosen level of details, in a hierarchical manner.

The first step "natural language proof" is deeply impacted by whether the system is already designed, ranging from a new system currently in its first stages to an existing system with legacy design. There is a paradox here: the job of the formal proof team is easier if the design is stable and well known, but then the benefits of the proof are reduced, because if pitfalls exist they will not be easily corrected and if there are no pitfalls the whole proof work seems useless. Conversely, if the design is not yet decided at all, there may be very little to prove.

In the Flushing NYCT project, the design was well known but currently under modification: THALES designers were currently adapting their CBTC design to NYCT's requirements. ClearSy had extensive contacts with the designers, so the design was easily accessible with comprehensive explanations. In the I2S project, the situation was more difficult because we had to rely only on the part of the design imposed by the interface specification: for the vendor specific part, we had to assume that the vendor's internal design would correctly establish the sub-properties taken as assumptions in the proof. ClearSy so added documents called "proof requests" (not meaning "formal" proof request) to the I2S explaining required properties and clarifying context assumptions under which these properties should hold. Besides, the design fixed with the interface was not to be changed at all, and the "reasons why" of this design was not so easily accessible. In subsequent projects we also encounter the case of a design still in its very first phases: because proving all possible solutions is not feasible, the initial task is then reduced to defining the notions in preparation for future proofs.

5.2 The Natural Language Proof

In all cases, the necessary elements to extract for the "natural language proof" are:

- Well defined target properties;
- A set of fully realistic assumptions about concerned sub-systems and context;
- An understanding of how these assumptions ensure the properties.

Defining target properties is not difficult at the system's top level because it's directly linked to what the system should obviously ensure. It is conversely *very difficult* to find the true properties from the sub-systems and the context constituting the set of assumptions cited above: the role of each subsystem is known by each domain's experts in terms of internal design details, never (or practically never) as abstract properties. Actually, it is far easier and less prone to errors to describe a subsystem using all its design details "as so" than to formulate an abstract property that sums up this subsystem. When trying such a formulation, one quickly discovers the asymptotic difference between an *almost true* property and an *always true* property for characterizing the considered subsystem!

For the proof team, it would not be realistic to expect such well-defined abstract properties from the experts or from the project documents. ClearSy uses the following method, starting from identified top properties and a first understanding of subsystem roles:

- Play scenarios trying to violate the wanted property (for instance, at top level try to play a scenario leading to a train collision), in a light and fast way, until the reasons why violating the property is impossible appear.
  - Ask feedback about those scenarios from system designers: as the desired property (for instance no collision at any time) has been their concern, they will quickly explain why the property is ensured in that case (unless there is a real pitfall!).
- Once the reasons why the property is ensured have appeared, explain those reasons, at first informally then more and more rigorously.
  - Again, ask feedback about those simplest reasons why the property is ensured from original system designers.

We repeat these steps until sufficient abstract properties for subsystems and context appear, so that the target properties seem to be ensured. Like this, close contacts with domain experts and designers ensure that the proof team manages to find the required assumptions efficiently, without illegally re-designing the system (in a way that would be incorrect compared to the real system and that would not be functional, as the proof team is not the design team).

Starting from top level target properties, we use this process to obtain the three components previously cited: target properties, context /subsystems properties, reasoning. This is done in several successive steps: the sub-properties of a step become targets properties for next steps, and so on in a hierarchical way until we reach the appropriate level of detail. Note that seen from a proof point of view, the context and subsystems properties are the assumptions for a given proof step: for this reason we call them sub-properties or assumptions indifferently. Of course, at the end of the whole process only the assumptions from all the terminal branches of the hierarchical tree will be presented as the output assumptions, to be finally validated and rechecked in case of system evolution (Fig. 1).

Key points in the methodology. Even if using "natural language" in this phase is considered necessary, it means that before the formal stage (where we use Event-B [1, 3] and Atelier-B [2]) all the ambiguities of the natural language can remain. To detect such ambiguities when first writing the properties, we use the following criterion: *in any possible scenario, it should be possible to state unambiguously whether the property is*

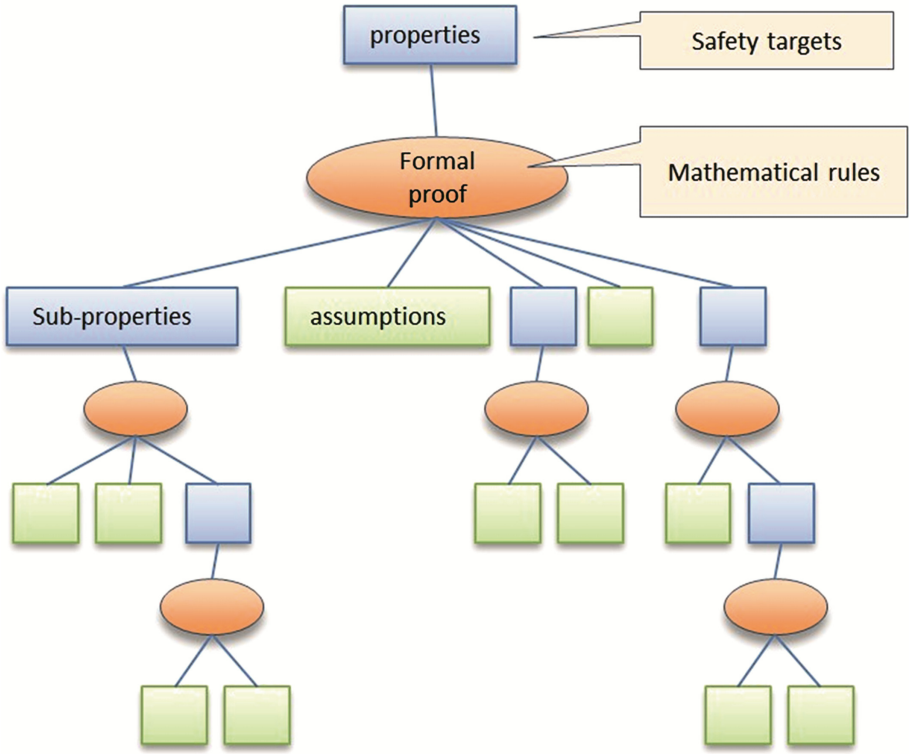

Fig. 1. Properties, sub-properties and assumptions

true or false in that case. Testing properties using this criterion leads to many questions before the formulation is satisfying. Another good test is that if an assumption or property is required from a subsystem, then we should be able to find a realistic accident scenario when removing this assumption. Such accident scenarios must be kept with the assumptions, they are their best justification.

To work in such close contact with the designers /experts, the proof team has to conform very strictly to its role:

– The proof team shall be neutral regarding the design choices;
– They should accept to go inside details with the experts, even if it means analyzing a lot of "how" to find a small amount of "why" (we say that they should accept to "plunge in the domain");
– They should do so to be a help to find and solve potential pitfalls, they should avoid at all costs any risk of discrediting the work of the designers.

Despite the fact that subsystems properties should be expressed without internal design details, to close the gap between 'almost true' and 'always true' the proof team often has to do *incursions inside the design details*. This is 'plunging into the domain'

as expressed above; it is ok as long as it is done with the experts, and with the will to efficiently find the subsystems properties and to formulate them independently from these details.

5.3 The Formal Phase

In the subsequent formal phase, all the natural language proofs are rewritten in formal language. ClearSy uses Event-B and Atelier-B for this purpose: schematically, the target properties are written into top level B models, the subsystem properties are written as B refinement models in such a way so that the proof performed by Atelier-B (according to the definition of refinement in B) shall be equivalent to the reasoning done in natural language.

The formal proof in this phase is a verification, as the proof itself (the "reasons why") is found during the natural language phase. In our experience, the reasoning involved is rarely very complicated or requiring high level mathematics: most of the difficulty in the whole process resides in formulating subsystem and context properties. Once this is done for a given topic and level, the keys of the reasoning already exist because the contributing properties were revealed by examining *how the subparts actually ensure target properties*. This work is done in collaboration with the experts and the designers, so if pitfalls exist they are normally resolved (or at least discussed and mitigated) at once. Thus, this process does not produce a list of bugs to be solved at the end: if everything goes well it should finish with a proof under approved assumptions, all pitfalls resolved or mitigated.

Once found how a given target property is ensured, with context and concerned subsystems roles optimally formulated, the reasoning often seems very simple. So there is a temptation to conclude that the concerned proof step is obvious and that it is not worth rewriting in B and Atelier-B. It is almost a quality sign: the better the expression of the "reasons why", the simpler the B models and the more obvious the proof. This is particularly true for system level proofs because we remain before the complexity wall of detailed functions names, data formats or electrical interfaces.

Our experience is that rewriting in B and proving with Atelier-B *is always worth it*. With the best possible preparation, all the natural language demonstrations that we transformed into actual B models and proved were changed during this process. Very often, turning natural languages assumptions into B formulas reveals cases where the meaning is uncertain or blurry: these are typically cases where the criterion (the one defined as a key point above) does not hold, i.e. scenarios where one cannot state whether the concerned property is true or false. Less often, extra pitfalls are discovered during the proof phase; then it is usually complex cases that would probably be impossible to find without proving.

An example of this kind of complex pitfall that we discovered only in the final proof phase concerned the safe braking function of CBTC trains, in very specific cases where an initial backward movement of the train during residual traction phase could impair the correctness of the braking calculus. Real track slope values were probably such that this could not actually occur, but well, the point was not spotted and not verified before.

6 Results and Their Usage

6.1 Where to Stop

The system level proof process ends when the chosen level of detail is reached, with all the formal proofs done. This chosen level of detail is variable: one can decide to go down to actual interfaces and design of subparts (for instance down to electrical relay schematics for parts done with this technology), or to stop at a higher level. In our CBTC projects, ClearSy went down to a quite detailed level in subparts (including for instance, how slipping wheels are detected by the onboard computer), but above the level of the data formats and the actual internal design. The actual computer code was not examined; neither the data formats like the system track database. We believe that this chosen limit is suitable to solve system level problems in a sufficiently detailed way, without including the complexity of software data representation or electrical signals. Note that system interface specifications usually must go deeper, as they should dictate the message formats between subsystems.

6.2 Output Documents

After the formal proof stage, all the final assumptions (about context, about subsystems design below the chosen final level…) are expressed. In the B models this formulation is not easily readable, so we translate them back into natural language in documents called "books of assumptions".

In these documents, we give the following information:

- Target properties (explained in natural language, illustrated with examples);
- Assumptions (explained in natural language)

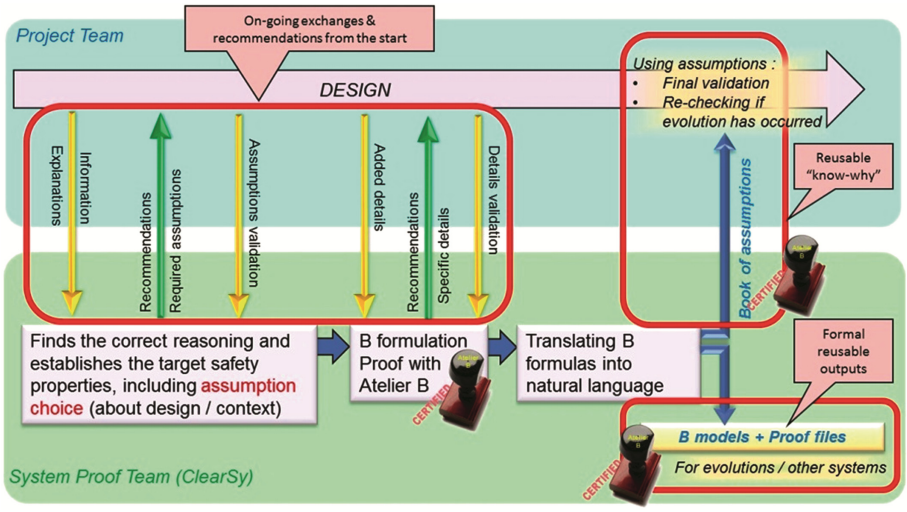

Fig. 2. The global process with its outputs outlined.

- For each assumption, we show scenarios of what could occur if it did not hold;
- We also give information about who validated the assumption, what are the concerned teams and whether extra validation is needed (if the system is not yet ready, or in case of evolutions).

The sentences used in these documents are indeed retranslated B formulas, so they are very precise (although sometimes with a not very literary style!). Experts and designers that participated in the process are already used to the extracted notions and the agreed assumptions, however we found useful to organize several day long presentations to explain the results to a larger audience within our customers. The final results are useful only as long as they are understood and used! (Fig. 2).

6.3 Safety Cases and Standards

Besides obtaining the assurance that a formal proof could be done at system level, the benefits of this process for a given system should naturally concern the safety case, as proven target properties are basically safety properties. Standards (like EN 50126, 50129 and 50128, see [5–7] in railway domain or EN 61508 more globally) favor formal methods but more at software development level. Taking into account the extra guarantee of a system level formal proof is left to the ISA to decide upon, although it is probably in the spirit of these standards to seek 'safety proofs' in any sense.

Nevertheless, system level proofs certainly do not exempt from performing all necessary safety cycle steps (including quality, organization, verification & validation, etc.). One paramount topic is the study of possible failures and their probability. In the system proof, safety target properties are clearly meant to hold whatever the possible failures, considered at the appropriate level of probability regarding the possible accidents. So the sub-properties and assumptions that the proof relies on should also hold whatever the possible failures, considered at the appropriate level of probability. This is how the failures & probabilities studies are related to the proof.

So failure rates and hazard studies remain of paramount importance; the proof helps in the definition of properties to be ensured despite these failures but does not change this part of the safety case otherwise.

7 A Sample Case Study: The Route Cancellation Example

How exactly are the properties written, how is the proof performed and how is this methodology based on natural language proof /formal proof applied? To give a better idea of the process, let's show a very simple example. Consider the following drawing:

In this Fig. 3, there is a switch beyond a signal. Obviously the switch should be locked when a train arrives, otherwise a derailment could occur. The problem is the following: what mechanism should be installed so that when the line operator wants to close the signal and change the position of the switch, no derailment shall be possible? The usual railway solution to this is to install a delay-based route cancellation system:

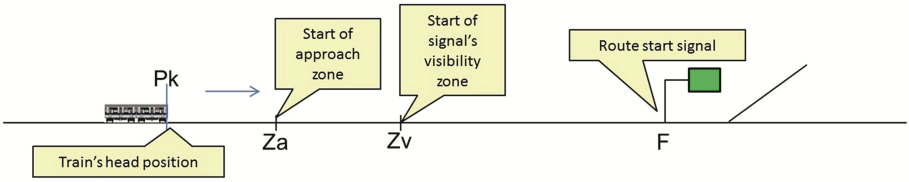

Fig. 3. The route cancel example.

- If the train is far enough (before some "approach" zone limit), the line operator can cancel the route and move the switch at once because the train has enough space and time to stop before the signal;
- If the train is already near, when the route is cancelled the signal should close but the system shall keep the switch locked for a certain time. Then the train will either stop before the signal or go beyond during this time. If the train is detected beyond the signal, the switch is maintained in locked state until the train has cleared the route.

Above is the purely informal reasoning. It is a good starting point; however it does not give the constraints between the delay and the train's stopping capabilities. Moreover, such informal clues do not provide any certainty about whether the switch is protected in all cases and under what assumptions (about train speed, braking, …). Things become more precise when we define variables /constants /notions to denote this system. Let's not describe the possible variables here in too much detail; with obvious definitions we can sense the necessary assumptions:

- If the train is beyond F (Pk > F), then no unlocking shall be possible;
- There must be a visibility zone (Zv) such that when the train is near (Pk > Zv) and the signal is red, the train will always stop in a given maximum delay (Ts) and a given maximum distance (Ds); otherwise the train could completely ignore the signal.
- The approach zone (Za) such that the train detection inside will trigger the locking delay (T) when the route is cancelled must be larger than the worst stopping distance Ds: otherwise the train could overrun the signal after a route cancel that unlocked the switch without delay. For even more obvious reasons the visibility zone (Zv) shall be larger than Ds: the signal F would be useless otherwise.
  - Note: in the figure Za is before Zv, but this is just an example. There is no such constraint.
- The unlocking delay T shall be greater than the worst stopping time Ts: otherwise the timer may expire just before the train overruns F, thus unlocking the switch in front of the train beyond F.

At this stage, the named variables allow a far more precise description of the problem; the constraints and necessary assumptions appear thanks to the scenarios seen above. These scenarios are what we could call "otherwise" scenarios: they reveal the necessity of each assumption by showing what could occur otherwise. The target property is now easily defined: if the train is beyond the signal (Pk > F), then the switch should remain locked (using some variable named 'unlocked': unlocked = False).

We now have a strong intuition that the system is safe with only the few assumptions above (F-Zv > Ds, F-Za > Ds, T > Ts), but this is not yet a proof. Here comes the natural language proof step: finding out how we can conclude that the switch will always be locked if the train reaches it. This is a case by case reasoning:

- If the route cancel occurs when the train is not yet in the approach zone (Pk < Za), then it will stop before F (Za + Ds < F). So no question is to be asked about the switch. The same applies if the route cancel occurs when the train is not yet in the visibility zone (Pk < Zv), because the visibility zone is large enough (Zv + Ds < F).
- In the remaining case, the route cancel occurs when the train is already in the approach zone and in the visibility zone. Because the signal is directly visible, the stopping delay of the train can be counted from the route cancel event; because it is in the approach zone, the switch will remain locked during T after this route cancel event. So the switch is still locked when the final stopping of the train occurs before T; if the train is beyond F at that moment it will be detected and the switch will remain locked until the train has cleared the route.

This reasoning can be explained and checked with domain experts: this is the "natural language proof". It may seem simple, but it is now possible to carefully examine the meaning of the variables and the related assumptions: the most crucial problems are usually solved at this moment.

If we stopped here, there would be no quantifiable difference compared to an informal text explaining the system, however rigorous the previous "proof" may seem. The next step beyond is to use a computerized tool to verify this proof: we use Atelier-B with its event-B language capabilities. Constructing an event-B model such that the proof will denote the above reasoning is quite straightforward (provided enough knowledge in the event-B method of course). A single B model is suitable, with sate variables denoting Pk, time, date of the route cancellation if it occurred and so on. The target property (Pk > F => unlocked = False) can be written in the invariant, if the set of B events in the model is so that this invariant can be proven then this property is proved. Those events will be:

- Events denoting the movements of the train (in different cases: without visible red signal, with visible red signal, etc.);
- Events denoting the ground system: route cancellation, switch unlocking after the delay, etc.

The obtained B-model in this case is about 80 lines long; its proof with Atelier-B is almost fully automatic.

Such B models for system level proofs with this method are usually far simpler than B models at software level: they are often less than 1000 lines per model, and each topic is usually a single refinement chain with less than 3 or 4 refinement steps. Of course they are more complex than this toy example (80 lines, no refinements), but actually the "reasons why" at system level tend to be quite simple and so are the corresponding B models. The difficulties in this task (and the benefits) reside in *finding the correct assumptions*, and *successfully turning all explanations into a rigorous reasoning*. From a mathematical point of view we use only simple rules and results; system designs can

rely on elaborated mathematical results or complex physical laws but then those central aspects are well known and covered by theories and proofs that should be taken as assumptions in the B proof.

8 Conclusion

According to ClearSy's experience today, a system level proof is feasible with manageable cost for any large system. Of course, the correct organization and the results and their usage are highly dependent on the nature of the project, and particularly on whether the design is known or under development. When driving such proof projects, the important words are adaptation, flexibility and communication rather than theoretical mathematics!

In industrial projects, the efforts to reach the required performance and to obtain the mandatory documents obviously come first. Extra efforts to reach higher levels of confidence (like actually proving properties, using formulated assumptions and logics) are always a matter of conviction. Let's hope that global experiences in more and more projects will accumulate evidence in favor of this conviction.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Atelier B website. http://www.atelierb.eu/
3. Abrial, J.R.: The B-Book. Cambridge University Press, Cambridge (1996)
4. Boulanger, J.L.: Formal Methods Applied to Industrial Complex Systems. Wiley-ISTE, Hoboken (2014)
5. EN50126: Railway Applications - The Specification and Demonstration of Reliability, Availability, maintainability and Safety (RAMS)
6. EN50129: Railway Applications - Communications, signaling and processing systems – Safety related electronic systems for signaling
7. EN50128: Railway Applications - Communications, signaling and processing systems
8. Sabatier, D., Burdy, L., Requet, A., Guéry, J.: Formal proofs for the NYCT line 7 (Flushing) modernization project. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 369–372. Springer, Heidelberg (2012)