From Requirements to Development: Methodology and Example

Wen Su[1], Jean-Raymond Abrial[2], Runlei Huang[3], and Huibiao Zhu[1]

[1] Software Engineering Institute, East China Normal University
{wensu,hbzhu}@sei.ecnu.edu.cn
[2] Marseille, France
jrabrial@neuf.fr
[3] Alcatel-Lucent Shanghai Bell
runleihuang@alcatel-sbell.com.cn

Abstract. The main destination of this paper is the industrial milieu. We are concerned with the difficulties encountered by industrial developers who are willing to apply "new" approaches to software engineering (since they always face the same problem for years: how to develop safe software) but are in fact disappointed by what is proposed to them. We try to characterize what the relevant constraints of industrial software projects are and then propose a *simple methodology* able to face the real problem. It is based on the usage of Event-B [1] and is illustrated by means of an *industrial project*.

1 Introduction

We believe that, for a long time, software engineering has been considered either a theoretical discipline in certain circles of Academia, while a rather purely technical discipline in others. In the former case, the focus is put mainly on the mathematical semantics of the programming language that is used, while in the second case the focus is put on the informal modeling of the problem at hand using semantically meaningless boxes and arrows. In our opinion, none of these approaches is very useful for the working software developer facing industrial problems. This is because a vast majority of industrial software development project is characterized as follows:

1. The problem is usually not mathematically involved.
2. The industrial "Requirement Document" of the project is usually very poor and difficult to exploit.
3. The main functions of the intended system are often quite simple.
4. However, many special cases make things complicated and difficult to master.
5. The communication between the software and its environment involves complicated procedures.

As a result, the developer is very embarrassed because he/she does not know how to "attack" the project development. This is particularly frustrating because, from a certain point of view, the function of the final system seems to be very simple (case 3 above). For solving these difficulties, industrial managements have developed some "processes" defining the various phases that engineers have to follow in order to achieve the intended result, namely to have a final software satisfying its specification. Roughly speaking, the various phases of such industrial development processes are usually as follows:

1. Choose a programming language to code the software.
2. Write the software specifications, using various semi-formal approaches.
3. Design the system by cutting it into pieces with "well defined" communication mechanisms.
4. Code the various pieces using the chosen programming language.

5. Verify the code by means of various testing approaches.
6. Deliver the documentation of the system.

Note that phase 1 (programming language choice) is usually not an explicitly written part of the development process although it is implicitly very often decided quite early as indicated here. The discipline imposed by such a development process is certainly not a bad habit. However, in our opinion, it does not solve the real difficulty faced by the developer: how to structure the development approach so that one can have a strong feeling that the final product is indeed the one that was intended.

The purpose of this paper is to propose a systematic approach able to help the industrial developer to solve the mentioned difficulties. We think that such an approach or a similar one is not used by industrial practitioners, this is the reason why we think it is worth proposing it here. We shall first present our simple methodology in section 2 and then illustrate it by means of a real industrial project in section 3.

2 Methodology

The key ingredient of this methodology is derived from the observation that the system we intend to build is *too complicated*: it has therefore to be initially simplified (even very much simplified) and then *gradually* made more complicated in order to take account in a smooth fashion of all its peculiarities.

However, one of the main difficulties here for people to adopt such a practice is that they are not usually used to consider first a system that is simpler than the one they have to develop: they try to immediately take into account all the complexities of the system at once. Also the usage of a top-down approach is in fact very rare in industry. The practice of heavy testing has made people writing code and only then trying to validate it.

2.1 The Requirement Document

But, of course, in order to figure out what the real complications of the system we intend to build are, we cannot in general rely on the industrial requirement document that is at our disposal. As said in the Introduction, this document is quite often very difficult to exploit. So, our first action is to *rewrite this document*: this is to be done of course together with the people who wrote the initial one.

It is important to state that we do not consider that the initial requirement document is useless: most of the time, it contains all the details of the future system. We just say that it is difficult to read and exploit for doing our initial engineering work. In section 3.2, we shall explain more on the requirement document while presenting our example.

2.2 The Refinement Strategy

As said in the previous section, the key idea of this approach is to proceed by successive approximations. For doing this, it is important to prioritize the way we are going to take account of the requirements that have been carefully identified in the new requirement document. When we say "to take account of the requirements", we are not very clear here. To take account of them in order to do what?

1. To write the code. But how can we write software code by successive approximation? **NO**, writing the code is far too early a phase at this stage.
2. To reshape once again the requirement document. **NO**, one rewriting is sufficient.
3. To write a specification document by translating the requirements into boxes and arrows. **NO**, we are not sure that it will add something to the requirements that we have been very careful to write by simple natural language statements: experience shows that such translations, far from making things clearer, quite often make things very obscure and therefore difficult to exploit later to write the final code.

4. To write a specification document by using a mathematical language for translating the requirements. **NO**, the term "translation" is not adequate here. The requirements are usually not amenable to a direct translation.
5. To write successive mathematical models defining gradually some *mathematical simulations* of the dynamic system we intend to build. **YES**, that will be our approach. This notion of model will be developed in the next section.

2.3 Some Rules

Now that we have defined (at least vaguely for the moment) what is to be done after taking account of the requirements, we have now to give some rules by which this choice among all the requirements can be performed in various steps. This is not easy to do so, for the very good reason that there is not in general a single "obvious" choice. However, we can give some **rules of thumb**:

R0 Take a short number of requirements at a time.
R1 Take account of a requirement partially only if it seems too complicated to be swallowed in one step.
R2 Introduce the other parts of a complicated requirement in further steps.
R3 As a special case of the previous rules, when a condition depends on many different cases, first abstract it with a non determinate boolean condition and then make it concrete and deterministic in further steps.
R4 Introduce gradually the functions of the system.
R5 Start with the most abstract requirements: the main functions of the system.
R6 Introduce the most concrete requirements at the very end of the choice process.
R7 Try to balance the requirements of the software and those of its environment.
R8 Be careful not to forget any requirements.
R9 If a requirement cannot (for some reason) be taken into account, remove it from the document.

It is clear that this initial ordering of the requirements is not the last word concerning this question. We might discover later that our initial choice is not technically adequate. We might also discover that some requirements are simply impossible to achieve, or badly expressed, or be better modified, etc. We must feel free to modify our choices and to modify our requirement document as well. In section 3.4, we shall explain more about the refinement strategy of our example.

2.4 Modeling with Event-B [1] and Proving with the Rodin Platform [2]

Equipped with the "road map" (Requirement Document and Refinement Strategy) defined in previous sections [4], we can now enter our next phase: modeling.

Modeling versus Programming
The first thing to understand about modeling is that it is *quite different* from programming. In the latter, we describe the way a computer must behave in order to fulfill a certain task, while in the former we describe the way a system can be *observed*: the emphasis is put on the global simulation of an *entire system* taking account of the properties that must be obeyed. This is clearly something that is not part of the programming of a piece of software. In other words, the modeling we envisage is not that of the future software alone but rather the modeling of this software together with its surrounding environment. This aspect will be made clearer in the illustrating example presented in section 3.

[4] These two first phases of our approach can take a significant time, i.e. several months, for important projects.

Successive Approximations
Another important difference between modeling and programming is that modeling can be developed by successive approximations while this is clearly impossible, and even dangerous, with programming: every programmer knows very well how perilous it is to modify or extend a piece of software as one has no guarantee not to introduce subtle bugs in doing so.

Mathematical Modeling
The simulations done in our approach are not realized by using a simulation language: we rather build mathematical models of discrete transition systems. All this is described in details in [1] where this approach is called "Event-B". Next is a brief informal description of Event-B, which was largely inspired by Action Systems [4] [5].

Event-B
Roughly speaking, a mathematical model (a simulation) done with Event-B is simply defined by means of a *state* and some *transitions* on this state (the events). The state is defined by means of variables together with some permanent properties that these variables must be fulfilled: such properties are called the *invariants* of the state. Each transition is defined by means of two items: the *guard* defining the necessary conditions for the transition to occur and the *actions* defining the way the state is modified by the transition. As can be seen, any "state machine" (which is nothing else but a transition system) can be defined in this way. Some *proofs* must be performed in order to ensure that each transition indeed maintains the properties of the state variables (the invariants).

Superposition Refinement
A model described in the way we have just mentioned is able to be *extended*: this is done by adding more variables (and thus more invariants) and more events to it. This technique is called *superposition refinement* [3]. Besides extending the state and adding events, some extensions can also be performed in two more ways on already existing events: (1) by strengthening their guards, and (2) by extending their actions. It means that an existing transition can be made more precise in a superposition refinement by giving additional constraints to it.

Proofs
Invariant preservation proofs may have to be performed while doing a refinement. Moreover, in a refinement, proofs of guards strengthening have also to be added to those of invariant preservations. The important thing to understand here is that proofs already done in a previous model (the one that is extended) *remain valid after the extension*, and so on while doing further refinements. In other words, we can *accumulate* progressive proof work in a safe way.

Deadlock Freeness
An important property of a state transition system is *deadlock freeness*. A system (or some part of it) is said to be deadlocked if no transition can occur any more. Most of the time, we are interested in modeling systems that never deadlock. It is simply done by proving that a transition can always occur. As the conditions for an event to occur are defined by its guard, deadlock freeness is thus ensured by proving that the disjunction of the guards of the concerned events is always true.

The Rodin Platform
In the previous sections, we mention the necessity of proving things about models: invariant preservation, guard strengthening, and deadlock freeness. In a typical industrial project there might be several thousands proofs[5]. It is obviously out of the question

[5] In the example developed in this paper, we have 439 proofs.

to manually generate what is to be proved and to manually prove all of them. A tool has been built for doing this: the Rodin Platform [2]. It has been developed over the last seven years by means of European Projects fundings. The Rodin Platform is constructed on top of Eclipse. It contains many plug-ins among which some important ones are the Proof Obligation Generator and the Prover. The former is able to generate the statements to be proved by analyzing the models. The latter is able to prove the various statements generated by the previous plug-in either automatically or interactively (i.e. helped by the human user giving some hints to the automatic prover)[6].

Finding Errors while Proving
An important aspect of this proving effort is that we might encounter problems in attempting to perform some of the proofs: we might discover that they simply cannot be proven automatically nor interactively. This is the indication that something is wrong in our model. This might be corrected by modifying, removing or adding some invariants, or by doing the same on events guards or actions.

Proving versus Testing
What can be seen here is that proving plays for models the same role as that played by testing for programs. However, the big difference between the two is that proving is not performed at the end of the modeling phase but in the middle of it, more precisely at each refinement step. This means that proving is indeed part of modeling. In doing this, we might also figure out that certain requirements are in some cases impossible to achieve: this is where it might be advisable to return to the requirement document and modify it. We might also figure out that our refinement strategy has to be re-thought: some requirements might be taken in a different order so as to improve the proving process.

Checking Models against the Requirement Document
While developing the various successive models of a system, we have to follow what we prescribe in the refinement strategy. This is done by checking that, at each refinement step, we take account of the various requirements that were chosen. At the end of the modeling phase we must have taken all requirements into account.

Data Refinement
The next step is to envisage how the part of the model dealing with the future software can be translated into executable code. Before doing that however, it might be necessary to envisage other kinds of refinements needed to transform the data structures used in the model into implementable data structures. This kind of refinement is called *data refinement* [6]. It is not considered in this paper and not used in the example of section 3.

2.5 After Modeling

Once the modeling phase is finished, that is when in the last refinement we can figure out that we have taken successfully all requirements and done all proofs, then we can envisage to go into the next phase: coding and executing. This will be done by using automatic plug-ins of the Rodin Platform.

Concerning execution, some interesting plug-ins are also available on the Rodin Platform: these are AnimB and ProB [7]. They are able to directly animate (without prior translation) and also model-check some Event-B models. Such animations are quite useful: they allow users to see how the global model of the system can behave.

[6] In the example developed in this paper, all 439 proofs are done automatically (except two of them done interactively) by the prover of the Rodin Platform.

3 The Example

In this section, we propose to illustrate the methodology we have just briefly presented in the previous section. This will be done thanks to a real industrial example. In developing this example, we shall give more information than the general one we already gave on the way each phase is performed. We shall follow the various phases that were mentioned in the previous section:

1. Re-writing the requirement document in section 2.1.
2. Make precise the refinement strategy in section 2.2.
3. Develop the various models in section 2.4.

Our example is extracted from the software controlling the behavior of a train. This software is called the "Vehicle OnBoard Controller (for short VOBC): the part we shall develop is called the "Mode Selection Subsystem", it is a module of the VOBC.

3.1 Main Purpose of System

The purpose of the system under study is to detect the *driving mode* wished by the train driver (he has some buttons at his disposal to do that) and decide accordingly whether this mode is feasible so that the current mode of the train could be (or not be) that wished by the driver. The different modes are the following:

1. OFF: train stops,
2. Restricted Manual Forward (RMF): forward manual drive, no train protection,
3. Restricted Manual Reverse (RMR): backward manual drive, no train protection,
4. Train Protection Manual (ATPM): forward manual drive, train protection,
5. Automatic Mode (ATO): forward automatic drive, train protection.

The "train protection" is a special automatic procedure taking care of dangerous situations (like trespassing a red light). In certain circumstances, the VOBC shall trigger the emergency brake (EB) of the train in case the request made by the driver might put the train in a dangerous situation. When such a special case vanishes then the VOBC can resume with a normal behavior.

As can be seen, this system seems to be quite straightforward. It is indeed. However we shall discover in the sequel that there is a vast number of special cases and also some peculiar equipments so that the system becomes complicated to develop. In fact, we face exactly the difficulty that was mentioned at the beginning of this article: a complicated and intricate situation without any sophisticated mathematics.

3.2 The Requirement Document

The requirement document takes the form of two embedded texts: the *explanatory* text and the *reference* text. They are both written in English. The former contains general explanations about the project we want to develop: it is supposed to help a new reader to understand the problem. The latter is made of short (dry) labelled and numbered statements listing the precise requirements that must be fulfilled by the concerned system: these statements should be self-contained[7].

Some of the "requirements" are mainly assumptions concerning the equipment rather than, strictly speaking, genuine requirements. They are nevertheless very important as they define the *environment* of our future software.

[7] The example treated here is a simplified version of a real example. We simplify it in order to cope with the size of this paper.

1) Requirement labelling

We shall adopt the following labels for our assumptions and requirements: TR_ENV (train environment), DR_ENV (driver environment), VOBC_ENV (VOBC environment), and VOBC_FUN (VOBC functions).

2) The Main Actors

Here are the various "actors" of our system: a train, a train driver, and the VOBC (a software controller). In the sequel, we shall first define some assumptions about these actors and then focus on the main function of the VOBC.

3) Train Assumptions

The following assumptions are concerned with the devices, equipment, and information that are relevant to our project in the train: the cabin, the mode button, the emergency brake, and speed information.

A train has two cabins (cabin A and B), each one is either active or inactive.	TR_ENV-1

Each cabin contains a Mode Selection Switch (MSS), with available modes: - Off (OFF) - Restricted Manual Forward (RMF) - Automatic Mode (ATO) - Restricted Manual Reverse (RMR)	TR_ENV-2

Each cabin contains an Automatic Train Protection Manual button (ATPM).	TR_ENV-3

It seems strange, a priori, to have the mode ATPM not defined as another alternative in the MSS button: this makes things complicated. We shall see how in the development, we might first simplify this situation by considering that the switch has an ATPM position in order to focus more easily on the main problem. Here are more train assumptions:

The train has an Emergency Brake	TR_ENV-4	The train may be stationary	TR_ENV-5

4) Driver Assumptions

The following assumptions are concerned with the actions the driver can do and that are relevant to our problem: requiring a mode modification.

The MSS is used by the driver of an active cabin to request a certain mode	DR_ENV-1

The ATPM button is for the driver to request the Automatic Train Protection Manual Mode (ATPM)	DR_ENV-2

5) VOBC Assumptions

Next are a series of assumptions concerning the information the VOBC can receive from its environment.The next requirement shows that the VOBC can receive a large number of information: clearly we shall have to formalize this gradually. It is also mentioned that the VOBC works on a cycle basis. In other words, the VOBC periodically checks

this information and then takes some relevant decisions.

The VOBC has a periodic interrogation (cycle) to the train for the conditions: - The MSS position (OFF, RMR, RMF, ATO) - The active cab (cab A or cab B, no cab) - The speed (stationary,non-stationary) - The position - The orientation - The possible calibrated wheel - The Brake/Motor output (normal, failure) - The state of the driver screen (TOD) - The state of Brake Release - The ATPM button depressed - A valid LMA (Limit of Movement Authorization) - The startup tests completed - Trainline Healthy	VOBC_ ENV-1

The next requirement shows a complicated encoding of the information received by the VOBC. As will be seen below, we shall take into account this complex coding at the very end of our development only.

The VOBC has a periodic interrogation to the validation of Mode Selector, it is communicated the MSS position by means of the boolean below:

MSS	Mode 1	Mode 2	ACA	ACB	FWDCS	REVCS
OFF	0	0	X	X	X	X
FWD	1	0	1	0	1	0
FWD	1	0	0	1	0	1
REV	1	0	1	0	0	1
REV	1	0	0	1	1	0
ATO	0	1	X	X	X	X

** X means input does not play a part in determination of the mode.*
One active cab is required to be detected except OFF mode (zero or one)
All other boolean combinations are considered Mode Selector invalid.

VOBC_ ENV-2

The mention "FWD" and "REV" in this table for the "MSS" position come from the industrial document. More precisely, "FWD" stands for "RMF" and "REV" stands for "RMR".

Here is one the main output of the VOBC: triggering of the emergency brake.

The Emergency Brake is activated by the VOBC.	VOBC_ENV-3

6) VOBC Functionalities

In this section, we carefully define the various functional requirements of the VOBC.

· Active and passive state of the VOBC

In this section, we encounter a large number of cases where the VOBC can enter into the "passive" state. Obviously, we have to take such cases gradually only.

The VOBC can be in a passive or active state. On start-up, it is passive.	VOBC_FUN-1

After start-up, if the VOBC receives the startup tests completion then the VOBC shall move to active state.	VOBC_FUN-2

The VOBC moves to passive state if one of the following conditions is met: - Trainline is not healthy. - The mode selector input is invalid (VOBC_ENV-2) - The cabin combination is invalid (VOBC_ENV-2) - The Brake/Motor output is failure when the VOBC is in ATO mode. - The selected direction is towards the train rear when in ATO or ATPM mode - The position is lost when the mode is ATPM or ATO. - The mode transition required is detected but the transition fails - The current mode is ATPM or ATO but this current mode is not available (see below **Mode Availability** VOBC_FUN-7,8)	VOBC_ FUN-5

If the VOBC is in the passive state, then when the conditions for this are over, the VOBC tests the transition of the selected MSS mode: - If the test succeeds, it changes to the required mode in active state. - Otherwise, it stays in passive state	VOBC_ FUN-4

The previous requirement covers the case where the passive state was entered while in the ATPM mode. It goes back "naturally" in the RMF mode.

When the VOBC moves from active state to passive state, the emergency brake must be triggered.	VOBC_FUN-3

· Mode Availabilities

Here are some further requirements of the VOBC. We shall see that sometimes the ATO or ATPM modes are said to be "not available". Such complicated cases, again, will be taken into account gradually.

The ATPM and ATO modes can be *available* or *not available* by the VOBC.	VOBC_FUN-6

These conditions are required by an active VOBC to make the ATPM mode available: - Train position and orientation established - Wheels calibrated - Selected direction not reverse - Valid LMA received. - TOD is not failure	These conditions are required by an active VOBC to make the ATO mode available: - Train position and orientation established - Wheels calibrated - Selected direction not reverse - Valid LMA received - No Brake/Motor Effort failure detected - No Brake Release failure detected
VOBC_FUN-7	VOBC_FUN-8

· Mode Transitions (Basic Functionalities)

Here, at last, we reach the basic functionalities of the VOBC: the mode transition decisions. When in an active state, the role of the VOBC is to validate the mode required by the driver.

The VOBC provides the following operating modes for the train: - Off (OFF) - Restricted Manual Forward (RMF) - Automatic Mode (ATO) - Restricted Manual Reverse (RMR) - Automatic Train protection Manual (ATPM)	VOBC_ FUN-9

The VOBC can accept a mode change requested by the driver only if it has detected that the train is stationary	VOBC_FUN-10

If the VOBC detects a mode change requested by the driver while the train is not stationary then: - it activates the Emergency Brake - it maintains the current train mode of operation	VOBC_ FUN-11

If the VOBC detects a stationary train, it releases the Emergency Brake due to mode changes only when the VOBC is not in passive state	VOBC_ FUN-12

If the active VOBC receives a driver's mode request, it will not test the mode transition (see VOBC_FUN-14-19) until the train is stationary. - If the test succeeds, transit to the required mode - If it fails, transit to the passive state	VOBC_ FUN-13

Some of the following requirements about RMF or ATPM mode transition are complicated due to the presence of the ATPM button. This will not be taken into account at the beginning of the development.

The VOBC can transition to RMF mode while not in ATPM mode if the conditions below are met: - One cab is active. - The MSS is in the RMF position - The train is stationary.	VOBC_ FUN-14

The VOBC can transition to RMF mode while in ATPM mode if the following conditions are met: - One cab is active. - The MSS is in the RMF position. - The train is stationary. - The ATPM button is activated.	VOBC_ FUN-15

The VOBC can transition to RMR if the conditions below are met: - One cab is active. - The MSS is in the RMR position. - The train is stationary.	VOBC_ FUN-16

The VOBC can change to ATPM from RMF if all below are met: - One cab is active. - The MSS is in the RMF position. - The train is stationary. - The ATPM button is activated. - ATPM mode is available.	VOBC_ FUN-17

The VOBC can transition to ATO if the conditions below are met: - One cab is active. - The MSS is in the ATO position. - The train is stationary. - ATO mode is available.	VOBC_ FUN-18

The VOBC can transition to OFF if the conditions below are met: - One cab is active. - The MSS is in the OFF position. - The train is stationary.	VOBC_ FUN-19

3.3 Comments About the Previous Requirements

A Large Number of Requirements

We have 29 requirements to take account of. Notice however that the real example has far more requirements: more than one hundred. But even with this restricted number of requirements, we can figure out that things have become quite complicated to handle. It is not clear how we can treat this situation in a decent fashion. The difficulty comes

from the large number of variables we have to take into account. Again, this situation is *very typical* of industrial projects. We face here the exact situation that we want to solve in this article.

The Difficulty

At first glance, it seems very difficult to simplify things as we recommended in section 2 describing our methodology in general terms. It seems that the only approach we can use here is one where everything is defined at the same level. This is the case because all requirements are heavily related to each others: a typical case is VOBC_FUN_5 mentioning mode availability and VOBC_FUN_7,8 describing these availabilities.

The Solution: Abstraction, Refinement, and Proofs

We shall see in the next section how the introduction of abstraction and refinement will solve this difficult question: when some conditions are quite heavy (e.g. those making the VOBC state "passive" in VOBC_FUN-5), we shall abstract them by a simple boolean variable which will be later expanded. We shall see also that such an approach will have a strong influence on the *design* of our system.

Moreover, as explained in section 2.4, we perform some proofs at each refinement step allowing us to check that our system is consistent. Such proofs help us correcting errors that we might have introduced while modeling. Again, proving is part of modeling.

3.4 Refinement Strategy

In this section, we shall obey all the **rules** mentioned in section 2.3 as guidelines. We present here how to choose among all the requirements and perform them in various steps.

Initial Model: Normal Behavior

The idea is to start by *eliminating all complicated cases*, as mentioned in **R5**: only consider the main functions of the system, namely checking the consistency between the mode wished by the driver and the mode that is acceptable by the system, ignoring all the noise.

For instance, we shall suppose that the MSS button has an ATPM alternative: we can thus (temporarily) remove the ATPM button from the cabin (it was defined in requirement TR_ENV-3). This button will be re-introduced in the fifth refinement below. We can forget about the particularly complicated encoding between the train and the VOBC (defined in requirement VOBC_ENV-2).

We also eliminate all special cases where the emergency brake is possibly triggered (requirements VOBC_FUN-3 and VOBC_FUN-5).

Finally, we forget about the initialization of the VOBC (see VOBC_FUN-2).

The net result of all these simplifications is that we only consider the *normal case* where the wish of the train driver is positively received by the VOBC: this is the initial model.

In subsequent refinements we shall gradually re-introduce all the complicated cases we eliminated in this initial model. In doing this, we follow our rule **R4** concerned with the progressive introduction of the system functions.

First Refinement: Non-stationary Case

In this level, we take account of the most important special case that influences the system decision. By reading carefully the requirement document, it seems that this most important special case is the one where the wish of the driver is done while the train is not stationary (VOBC_FUN_10). We introduce this special case in this refinement. Here again, we follow our rule **R4**.

Second Refinement: Elementary Non-availabilities

Similarly to the previous refinement taking account of rule **R4**, we present here special cases inside the system decision. We also obey rule **R0**: taking a small number of requirements at a time. These special cases are the non-availabilities of the ATPM or ATO modes that are introduced in requirements VOBC_FUN_7,8 by some complex conditions.

For the moment, we shall follow our rule **R3** telling us that we can abstract a complex condition with a boolean variable: so, we only introduce the very fact that these modes might be unavailable by using abstract boolean variables obtained non-deterministically, without relying on the corresponding details. By doing so, we also follow our rule **R1** stipulating that we could take account of a requirement in a partial way only. These boolean variables will be given more deterministic concrete understanding in the sixth refinement below.

Note that the order of this refinement and the previous one could be interchanged as they do not depend on each other.

Third Refinement: Active and Passive States, Initialization

In this level, we extend our development of the VOBC by adding a system state (passive and active). This corresponds to requirements VOBC_FUN-1,2,3,4. Following **R1**, **R2**, **R3**, and **R4**, we take account in a partial way of the state of the VOBC by means of a boolean variable. Partially only because, again, all the fine details of the "passivity" are too complicated (they will be introduced in the sixth refinement). However, the availabilities of the ATPM and ATO mode are taken into account (see requirements VOBC_FUN-3) by means of the boolean variabIes introduced in the previous refinement.

We also take account of the initialization of the VOBC. Moreover, we consider the fact that only one cabin is active for accepting a transition.

Fourth Refinement: Emergency Brake

Here we focus on an important output of the VOBC which is influenced by all the special cases: we introduce the emergency brake (introduced in VOBC_FUN-3 and VOBC_FUN-11). It is triggered by all the special cases we have now considered (at least partially): non-stationarity (first refinement), non-availabilities (second refinement), and possible passivity (third refinement).

Until now, we took account of all the functions of the system in an abstract way. The following refinements consider now the very concrete requirements related to a large number of variables.

Fifth Refinement: Handling of the ATPM Button

In the initial model, we were cheating by having the ATPM button inside the MSS switch. This was done for simplification. We are now ready to introduce the special

properties of ATPM button for handling this particular mode. For this reason, we introduce special TRAIN variables and events. As mentioned in **R7**, we do introduce the environment variable in a way that is balanced with the VOBC development. This is what we do here: introducing the train variables only when we need it.

Sixth Refinement: Completion of Non-availabilities and Passive State Cases

We are now able to complete all cases of non-availabilities of the ATPM or ATO modes (VOBC_FUN_7,8). We also consider all cases of passivity (VOBC_FUN_5).

Seventh Refinement: Encoding of the Cabin and MSS button

We consider the encoding of the communication between the train and the VOBC, now taking partial account of requirement VOBC_ENV-2.

Eighth Refinement: Last Encoding

This refinement takes account of the last requirement of VOBC_ENV-2.

3.5 Refinement Strategy Synthesis

The following table maps the initial model and the eight refined models to the related requirements which are taken into account.

Refinement	VOBC Function	Train	Driver	VOBC
Initial	VOBC_FUN-9,14-19(p)	TR_ENV-2(p)	DR_ENV-1,2(p)	VOBC_ENV-1(p)
First	VOBC_FUN-10,11-19(p)	TR_ENV-5	–	VOBC_ENV-1(p)
Second	VOBC_FUN- $5(p), 6, 17(p), 18(p)$	–	–	–
Third	VOBC_FUN-1,2,4,5(p),14-19(p)	TR_ENV-1	–	VOBC_ENV-1(p)
Fourth	VOBC_FUN-3,11,12	TR_ENV-4	–	VOBC_ENV-3
Fifth	VOBC_FUN-15,17	TR_ENV-2,3	DR_ENV-2	VOBC_ENV-1(p)
Sixth	VOBC_FUN-5,7,8	–	–	VOBC_ENV-1,2(p)
Seventh	–	–	–	VOBC_ENV-2(p)
Eighth	–	–	–	VOBC_ENV-2

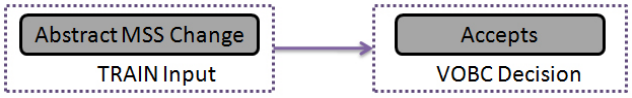
** (p) means the related requirement is PARTIALLY taken into account.*

As it is shown in this table, at each level we only take a short number of requirements, this follows **R0**. We also find many partial accounts. This follows **R1** and **R2**: we take account of complicated requirements partially only and leave the other parts for further steps. The whole refinement strategy follows **R5** and **R6**: start from the very abstract requirements and end with the concrete ones. After this refinement strategy is obtained, we take **R8** into account: to make sure no requirement has been forgotten, and then, following **R9**, remove all requirements that cannot be considered.

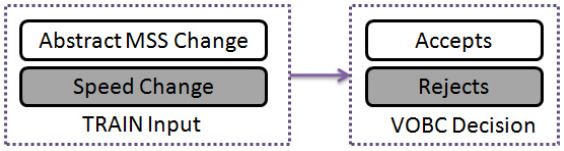
3.6 Formal Development

The formal development presented in this paper is only very short due to some lack of space. The interested reader can download the complete formal development of this example from the Event-B website [2]. In the following, we present some diagrams describing in a simple manner the events of each step. Such events are implicitly represented in some boxes standing for the various phases that will be executed. We follow exactly what was presented in section 3.4 describing our refinement strategy.

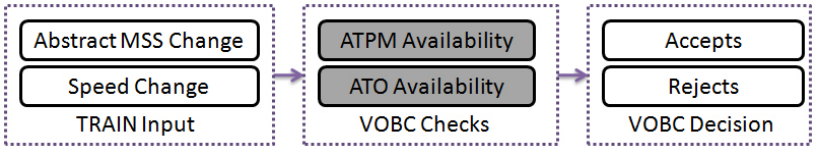
In the initial model, we have essentially two sets of events corresponding to the TRAIN Input phase where an abstract MSS might be changed and the VOBC Decision phase where the modification of the abstract MSS made by the driver is accepted:



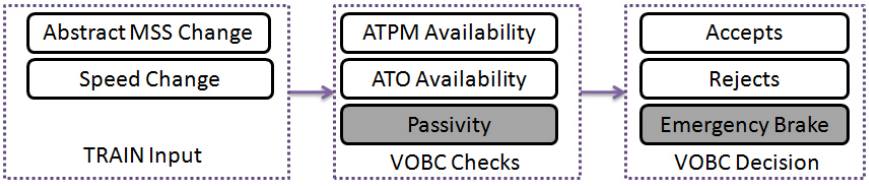
In the first refinement, we take account of the speed of the TRAIN and the corresponding possible rejection made by the VOBC:



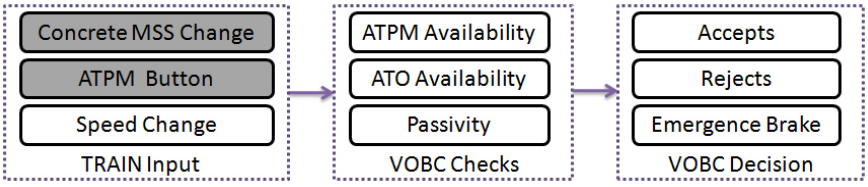
In the second refinement, we introduce some boolean variables dealing with the ATPM and ATO availabilities. These boolean variables are *non-deterministically* assigned in the VOBC Checks phase. The boolean variables are then used (read) in the VOBC Decision phase:



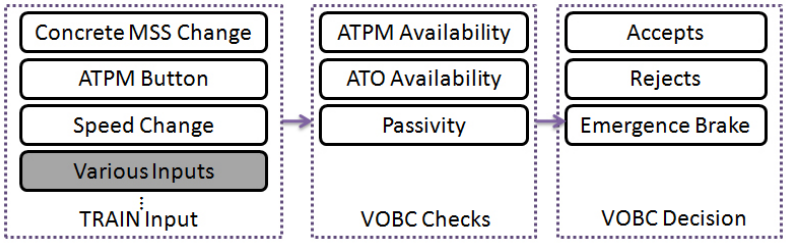
In the third and fourth refinements, we introduce a similar boolean variable for the "passivity" state and we deal with the Emergency Brake handling in the VOBC Decision phase:



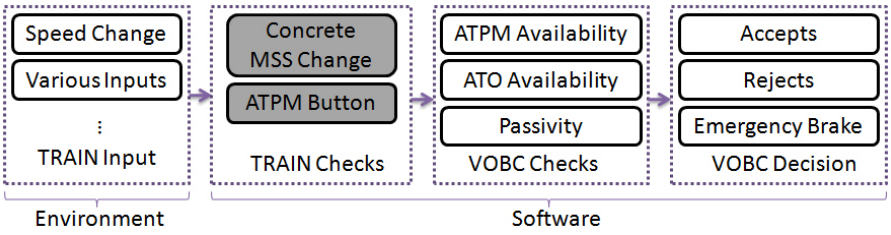
In the fifth refinement, we make the driver switch and button more concrete. More precisely, we now separate the ATPM button from the abstract MSS:



In the sixth refinement, we introduce a large number of various inputs in the TRAIN Inputs phase allowing us to make *deterministic* the assignments to the availabilities and passivity boolean variables in the VOBC Checks phase. An important aspect of what is done here is that the boolean variables introduced in the second and third refinements in the VOBC Checks phase are still used (read) in the VOBC Decision phase, which is thus *not modified*.



In the seventh and eighth refinements, we introduce more TRAIN Inputs and thus implement the MSS switch as well as the ATPM button. This introduces an intermediate TRAIN Checks phase. We have now a clear separation between the part of the model dealing with the future software and that dealing with the environment:



3.7 Proof Statistics

The entire formal development with the Rodin Platform generated 439 proof obligations all automatically discharged except two of them requiring a very light manual intervention (one click).

3.8 Timing and Determinism Issues

In the real industrial system, there are some important timing issues that have not been taken into account in this paper because of the lack of space. However, it its possible to give some information on how this can be formalized in our model. The problem is as follows: the driver MSS button change or ATPM button depression should last for a certain time in order to be taken into account by the VOBC. This is to avoid some outside troubles. The problem can be simply formalized by ensuring that any change in these buttons has to last continuously for at least some cycles (8 is a typical number) before being taken into account by the VOBC. This could have been incorporated at the end of the development.

Determinism (for the VOBC) is another important issue in industry. It means that exactly one event (of the VOBC) has its guard true. So, it a theorem we can prove. Deadlock freeness means there is at least one guard true, determinism is one step more precise: there is exactly exactly one guard that is true. It can be checked in any refinement, but it is more interesting to check it at the end.

4 Related Work

There have been recently several papers [8] [9] [10] on topics similar to the ones presented in this paper. They are all concerned with defining some guidelines for modeling complex systems. They treat problems that are more complex that the one envisaged here: how to structure the refinements of systems where a future software controller has to master an environment by means of some sensors and actuators. Their main message

is to start by defining the environment together with the properties to be ensured on it, and then (and only then) to study how a controller can handle the situation although it will base its decision on a fuzzy picture (due to the transmission time) of the real environment.

The case studied here is simpler than the one studied in these papers in that our controller is just there to decide whether the driver has the right to require a new mode. We do not control a complex situation. However, we treat a problem that is not so much studied in their examples, namely that of the presence of a large number of special cases and special equipments, transforming an apparently simple problem into one that becomes quite complicated (but not complex).

5 Conclusion

In this paper, we briefly recall a simple methodology to be used for industrial software developments. The main points we wanted to insist on are the following: (1) the importance of having a well-defined requirement document, (2) the need to enter in a formal model construction before the coding phase, and (3) the usage of superposition refinement in this modeling phase so that the system can be first drastically simplified and then gradually extended to fulfill its requirements.

This methodology was illustrated by a simple example representing a typical problem encountered in industry (although slightly simplified). In this example, we show a more important point, namely how the usage of superposition refinement leads naturally to the design of our system into successive phases enriching gradually the "contents" of its main variables until one can reach a final decision phase that is independent from the many more basic variables.

References

1. J.R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press 2010.
2. http: //www.event-b.org *Rodin Platform*
3. R.J.R. Back and K. Sere. *Superposition Refinement of Reactive Systems*. Formal Aspect of Computing 1995.
4. R.J.R. Back and R. Kurki-Suonio. *Distributed Cooperation with Action Systems*. ACM Transaction on Programming languages and Systems 1988.
5. M.J. Butler. *Stepwise Refinement of Communication Systems*. Science of Computer Programming 1996.
6. C.A.R. Hoare. *Proof of Correctness of Data Representation*. Acta Informatica 1972
7. M. Leuschel and M. Butler. *ProB: An Automated Analysis Toolset for the B Method*. International Journal on Software Tools for Technology Transfer 2008.
8. Thai Son Hoang and S. Hudon. *Defining Control Systems with Some Fragile Environment*. Working Report 723 ETH 2011.
9. M.J. Butler. *Towards a Cookbook for Modelling and Refinement of Control Problems*. Working paper, http://deploy-eprints.ecs.soton.ac.uk/108/, 2009.
10. S. Yeganefard, M.J. Butler, and A. Rezazadeh. *Evaluation of a guideline by formal modelling of cruise control system in Event-B* . In Proceedings of NFM 2010.