Mechanising the Validation of ERTMS Requirements and New Procedures.

Ángel Herranz, Guillem Marpons, Clara Benac Earle, Julio Mariño

Babel Group, Universidad Politécnica de Madrid
{aherranz,gmarpons,cbenac,jmarino}@fi.upm.es

January 31, 2011

Abstract
Our research aims at the mechanised treatment of ERTMS/ETCS specifications. The potential impact is twofold. On one hand, this will enable tools and techniques that can be used by manufacturers to automatically check the conformance of their equipment (on-board, track-side) with the operation requirements, prior to on-track testing. On the other hand, the use of these tools can also help in the design and validation of new operation procedures that somehow interact with the standards. We show a transformation of UML models of ERTMS/ETCS into Uppaal specifications that can then be used to analyse complex behavioural properties. Apart from the expected results, such as clarifying core elements, dispelling uncertainties, training of practitioners, and resolution of ambiguities, a valuable product can be automatically obtained: test cases.

1 Introduction

The European Railway Traffic Management System (ERTMS) has as one of its components the European Train Control System (ETCS), whose specification provides a standard for train control systems to guarantee the interoperability with track-side system across different countries and manufacturers. As with other industrial specifications, it is difficult to formalise and validate the ETCS specifications. They consist of several volumes of text written in English. The validation process often involves a large amount of costly experiments with real trains on real tracks. In 2007, the European Railway Agency (ERA, [5]) issued a call for tender for the development of a methodology, complemented by a set of support tools, for the formalisation and validation of the ETCS specifications.

It is often the case that the modelling notations used in the formalization phase are different to the modelling languages in the validation phase, given their focus on different aspects of a system, and therefore a model transformations is needed. In particular, UML is currently a successful modelling language used in industry although with poor validation capabilities. Formal methods, on the contrary, offer a collection of validating techniques that are often difficult to use in practise. The Eurailcheck project [6], originated from the successful tenderers of the ERA call, proposed the use of Rational[1] tools to write UML models, and the NuSMV model checker [8] to validate temporal logic properties of the these models.

Ineco-Tifsa [7] comprises a group of Spanish state-owned companies working on improving air, rail and road transport infrastructures. One of the tasks of their railway division is to participate in the specification of procedures that are specific to the Spanish railway system which must be consistent with the ETCS specification. This task is currently done by hand by well-trained engineers. However, within the company, there is a recognition of the need to improve the quality of their processes and to make sure that they follow the ETCS standard. Unfortunately, the results of the Eurailcheck project were not directly applicable to them. On one hand, they could not cover the costs of using proprietary software like Rational tools. On the other hand, it was not clear that the Eurailcheck tools could be easily installed and used nor that they were still maintained. For these reasons, Ineco-Tifsa approached us to get some help.

Our solution was to follow a similar approach to the Eurailcheck project, I.e. to model the specifications using a subset of UML [9] and the free UML toolbox BOUML [3], and to validate the model by translating

1 Rational is a trademark of IBM and Rational Software

this subset into Uppaal [2]. Uppaal is a model checker that allowed us to define models and to check properties that the specification should fulfil. Choosing the Uppaal model checker presents several advantages over NuSMV and other model checking tools:

1. Uppaal supports timed model checking, a key feature when modelling real-time and soft real-time systems, and which we use extensively in our models.
2. Simulations can be run and when a property is not fulfilled by the model a counterexample is generated. Such counterexample is automatically displayed as a sequence diagram, a formalism engineers feel comfortable with, and
3. It is possible to translate counterexamples and simulations back into UML sequence diagrams, as Uppaal sequence diagrams contain essentially the same information.

In this paper we show the modelling process in UML and sketch the translation of the considered subset into an Uppaal model by means of a concrete example, where some properties have been checked.

2 Case Study

As a proof of concept of our formalisation and validation methodology we have worked on a real example proposed by Ineco-Tifsa: the Level Transition Procedure defined in [4, Section 5.10]. ERTMS specifies different application levels, each one determining, among others, which communication channels between the trackside equipment and the on-board systems are available. The aforementioned procedure defines the possible sequences of events that make a train change its operation level, and the actions to be undertaken in each case.

2.1 Model

The formalisation process of the requirements should be done following a methodology. The methodology proposed by the Eurailcheck project [6] is very detailed. The main technical difference with our proposal is that we emphasise the importance of UML object diagrams while in the Eurailcheck proposal that kind of diagrams does not exist.

In the formalisation phase of our methodology, we rely on a subset of UML with extra consistency constraints. We use four different UML diagrams: object, class, state machine, and sequence diagrams [9]. The set of UML elements and the new constraints have been chosen to easily provide a semantics for all elements in the model and facilitate their translation into Uppaal. We have to take into account that we are modelling not only the embedded system but also some physical components.

2.1.1 Static Part of the Model

The purpose of object diagrams [9] is twofold. Firstly, they play a role in the discovery of classes, attributes, and associations that are later synthesised into class diagrams. They are also used to define specific scenarios that will be validated with Uppaal (see Section 2.3).

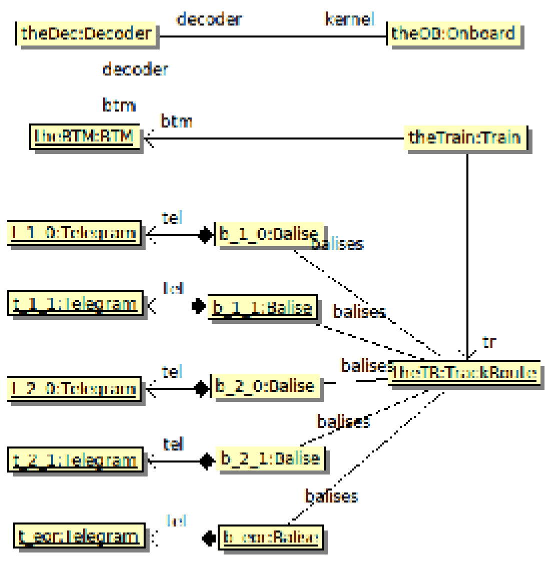

Figure 1: A UML object diagram defining a scenario to be validated.

Figure 1 shows a possible scenario containing a Trackroute with five Balises and one single Train. We have decided to decompose the train into different subsystems. Those relevant to our case study are the Balise Transmission Module (BTM), a Decoder (that converts low-level information received from the trackside into high-level signals), and the so-called Onboard system (responsible of operating the train given its current state and the inputs received). Every Balise contains a Telegram, that is transmitted to the train when it is over the Balise. The information of a telegram is decomposed into Packages, which are not shown in the figure. Packages contain Variables. An edge between two nodes of the diagram, called *link*, means that some kind of relation (structural, flow of information, etc.) exists between both objects. Links can have role names on their ends.

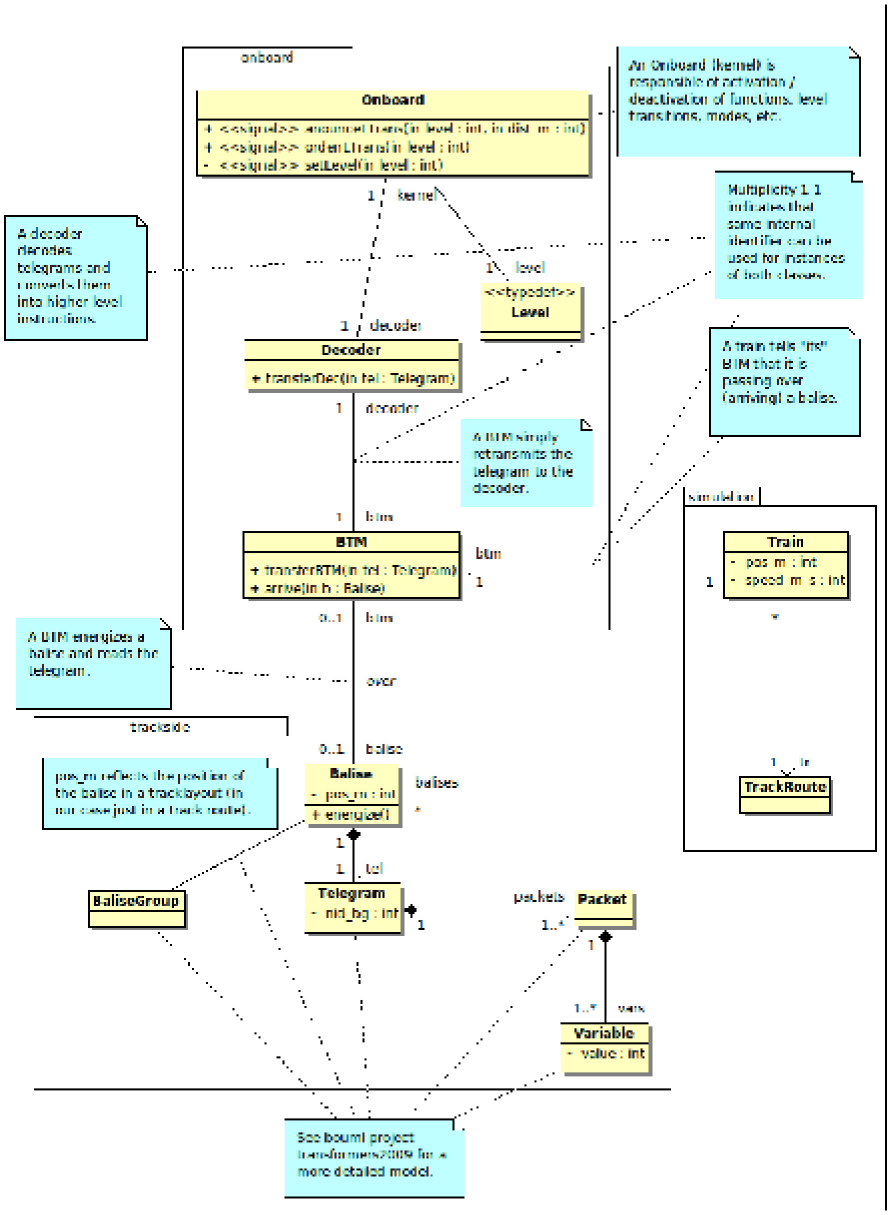
onboard
Onboard
+ <<signal>> announceLTrans(in level : int, in dist_m : int)
+ <<signal>> orderLTrans(in level : int)
- <<signal>> setLevel(in level : int)
An Onboard (kernel) is responsible of activation / deactivation of functions, level transitions, modes, etc.
1
kernel
Multiplicity 1 1 indicates that some internal identifier can be used for instances of both classes.
A decoder decodes telegrams and converts them into higher level instructions.
1
level
<<typedef>>
Level
1
decoder
Decoder
+ transferDec(in tel : Telegram)
A train tells "its" BTM that it is passing over (arriving) a balise.
1
decoder
A BTM simply retransmits the telegram to the decoder.
simulation
1
btm
BTM
+ transferBTM(in tel : Telegram)
+ arrive(in b : Balise)
btm
1
Train
- pos_m : int
- speed_m_s : int
1
*
0..1
btm
A BTM energizes a balise and reads the telegram.
over
trackside
0..1
balise
pos_m reflects the position of the balise in a tracklayout (in our case just in a track route).
Balise
- pos_m : int
+ energize()
balises
*
1
tr
TrackRoute
1
1
tel
BaliseGroup
Telegram
- nid_bg : int
1
packets
Packet
1..*
1
1..*
vars
Variable
- value : int
See bouml project transformers2009 for a more detailed model.

Figure 2: UML class diagram.

In class diagrams [9] we reflect all the types of objects that appear in concrete scenarios, called *classes*, and the types of links among them, called *associations*. Figure 2 shows the UML model that our scenario follows. Our subset of UML has extra consistency constraints, e.g., all instances (resp. links) that appear in an object diagram must be an instance of some class (resp. association).

2.1.2 Dynamic Part of the Model

To describe the dynamic behaviour of the system we rely on UML state machines [9]. Every instance of an scenario has, conceptually, at least one state machine describing how it reacts to changes in its environment. Some complex instances can have more than one state machine if this simplifies the model.

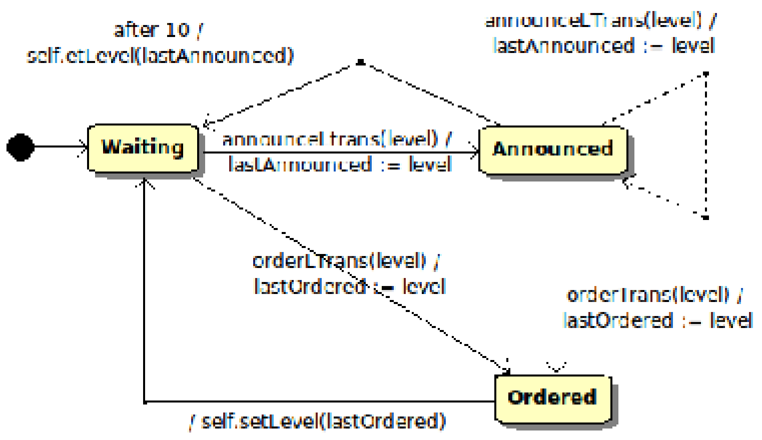

Figure 3: The UML state machine of an Onboard level transition.

Figure 3 shows how an Onboard system reacts to level transition announcements and orders coming from the trackside, finally performing a level transition (operation set_level()), if necessary (according to the specification in [4]). Level transition announcements and orders are signals received by an Onboard coming from the low-level devices of the train (Decoder and BTM). These components have received information (Telegrams) from the Balises in the Trackroute containing the corresponding announcements and orders. The state machines of these components define how to behave when the train arrives to a Balise, and how to translate the information received into signals for the Onboard.

Some constraints we have added to UML when defining state machines are the following:

- If the state machine of an instance *A* needs to send a signal *sign* to an instance *B*, the class of instance *B* must define an operation sign() with appropriate arguments. Some transition in the state machine of *A* will have an invocation to sign() in its activity, and some transition in *B* will have a call to sign() as its trigger. These operations are asynchronous and stereotyped with «signal».
- The aforementioned asynchronous operations are the only possible communication between instances in the model.
- All the operations and attributes needed to define the guard or the activity of a transition in a state machine must be defined in the corresponding class.
- Triggers with the reserved word “after” indicate that transitions occur by timeout.
- Only integer and Boolean attributes (or arrays of these types) are allowed.

We use sequence diagrams in a similar fashion than object diagrams: as a means to discover possible transitions in state machines and interactions among different machines. They also play a role in the validation phase, to give the user a counterexample of a unsuccessfully checked property.

2.2 Transformation

In this section we explain the transformation of a UML model [9] into an Uppaal model [2].

2.2.1 Classes and Instances

In general, each instance in the scenario will have a reference in Uppaal and some associated data corresponding to its attributes and association instances (links).

We start with the translation of Balise and its instances. Two Uppaal types are introduced to represent references as integers (`BaliseIdx`) and the set of attributes and roles in associations as records (`Balise`):

```
// Class Balise
typedef struct {
  int pos_m;
  Tel tel; } Balise;
```

```
// Maximum number of Balise instances:
const int MAX_BALISES = 5;
// A type to represent the references of
// Balise instances
typedef int[1,MAX_BALISES] BaliseIdx;
```

The references of instances are internal to Uppaal and have nothing to do with any concept in the requirements. A reference is assigned to every instance of Balise in our scenario through the global variable `balises`:

```
const Balise balises[BaliseIdx] := {b_2_0, b_2_1, b_1_0, b_1_1, b_eor};
```

Every b_*n*_*m* contains the attribute `pos` (its distance from the beginning of the track route) and the association (role `tel`) to its telegram. In the example, balise with reference 1, ie. `balises[1]`, is defined by the constant

```
const Balise b_1_0 := {200, tel_1_0};
```

Classes Telegram, Package or Variable are data types. When instances are not shared, we have consider more appropriate to give a direct translation of the classes into Uppaal types without introducing infrastructure for references:

```
// Class Var(iable)
typedef int Var;  // just one attribute.

// Class Packet
const int MAX_VARS_PER_PACKET = 8;
typedef struct {
  int nVars;
  Var vars[MAX_VARS_PER_PACKET]; } Packet;

// Class Tel(egram)
const int MAX_PACS_PER_TEL = 2;
typedef struct {
  int nid_pig;
  int nid_bg;
  int nPackets;
  Packet packets[MAX_PACS_PER_TEL]; } Tel;
```

Now we can understand the definition of telegram in the balise with reference 1:

```
const Tel tel_1_0 := {0, 101, 2, {announce_300, eot}};
```

where packets `announce_300` and `eot` are

```
// Packets
// 41: nid_packet, q_dir, l_packet,
//     q_scale, d_leveltr, m_leveltr,
//     l_ackleveltr, n_iter
const Packet announce_300 := {8, {41, 1, 63, 1, 300, 3, 1000, 0}};

// 255: nid_packet
const Packet eot := {1, {255, 0, 0, 0, 0, 0, 0, 0}};
```

To finish with the track layout we present the translation of the class Trackroute and its association with class Balise.

```
// Class Trackroute (TR)
typedef struct {
  int nBalises;
  BaliseIdx balises[MAX_BALISES]; } TR;
```

We can see that role balises has been added as an attribute that represent the association between both classes.

In this case, since in our scenario there are no more than one route, there is no need of creating the infrastructure for several instances and just one global variable is enough:

```
const TR theTR = { 5, {3, 4, 1, 2, 5}};
```

A route with 5 balises: `balises[3]`, `balises[4]`, ... The remainder of instances in the scenario are single instances of Train, BTM, Decoder, and Onboard with associations 1 to 1 between them. These associations are directly encoded by the existence of the Uppaal global variables that represents each instance:

```
// Class Train
typedef struct {
  int pos_m;
  int speed_m_s;
  TR tr; } Train;

// Class BTM (no attributes)

// Class Decoder (no attributes)

// Class Level (see ERTMS var M_LEVEL)
typedef int[0,4] Level;

// Class Onboard
typedef struct { Level level; } Onboard;
```

Here is the Uppaal variable that represents a train in position 0, speed 50 m/s and the planned route `theTR`.

```
// Class Level (see ERTMS var M_LEVEL)
typedef int[0,4] Level;
```

2.2.2 State Machines

In this paper we show direct translation of UML state machines into Uppaal automata. Main considerations are: (i) the context of each state machine is an instance of a class and all instances have its own machines, (ii) operations stereotyped as asynchronous («signal») represent events and are translated as channels, and some well known Uppaal idioms [1] are applied to the transmission of arguments, and (iii) synchronous operations are translated into Uppaal functions.

The class Balise has the operation +energize(). This operation is translated into an array of channels indexed by the balise reference:

```
chan energize[BaliseIdx];
```

Triggers in transitions of state machines of class Balise with the message expression self.energize()

are translated into an edge in an Uppaal automata with synchronisation `energize[`*r*`]?` , where *r* represents the reference of the balise (one automata per instance is created with a template indexed by references). Activity expressions with message invocation *b*.energize() are translated into and edge with synchronisation `energize[]!` where is the reference given to balise *b*.

Let us show in detail an example of message with arguments. In the Onboard instance, the state machine LevelTransManager will listen for a message +announceLTrans(level, dist_m). Global variables for the channel and message arguments are introduced in Uppaal:

```
chan announceLTrans;
Level announceLTrans_level;
int announceLTrans_dist_m;
```

This time, invocation of the operation is represented with the synchronisation `announceLTrans!` and with an assignment for the arguments:

```
announceLTrans_level := 2,
announceLTrans_dist_m := 1200
```

The machine listening for the level transition announcement will synchronise with `announceLTrans?` and must load arguments in local variables:

```
l := announceLTrans_level,
d := announceLTrans_dist_m
```

The state machine diagram for managing level transitions is translated into Uppaal and shown in Figure 4.

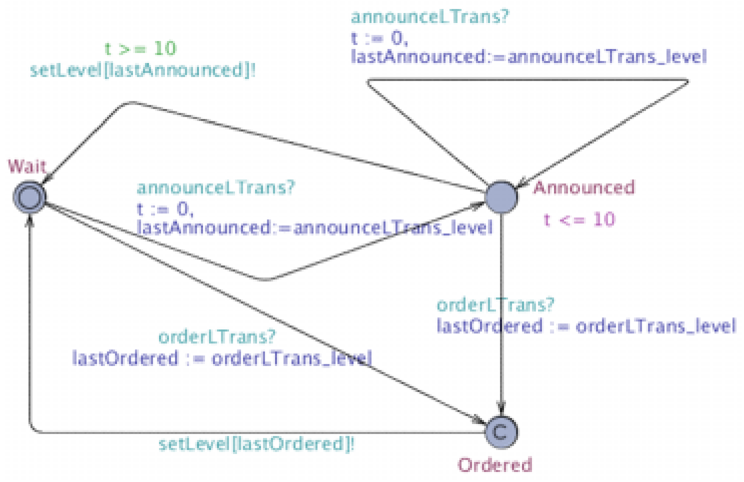

Figure 4: The Uppaal process of an Onboard level transition.

To animate and validate the model, the instance of the class Train contains a state machine that represent the physics of a train at constant speed.

2.3 Validation

In this section we explain some of the properties that have been validated for the Uppaal model. Some sanity checks can be systematically introduced with the established UML model semantics: types, multiplicities, etc. We claim that errors injected because of typos, non suitable data, or defective behaviour descriptions will be discovered by these *simple* checks.

Nevertheless, the most worrying errors are due to misunderstanding of requirements. These type of errors can be detected if we can formulate the properties we expect our model to fulfil.

2.3.1 Expected Properties of the Procedure

Given the procedure and the scenario, the following informal properties have been extracted. Here we show them with their formulation in the temporal logic query language of Uppaal:

- No other ERTMS level that 2 or 3 can be active in any moment in the on-board system (`A[]` means "*in any possible execution always*...".

  ```
  A[] (Onboard.level == 2 || Onboard.level == 3)
  ```

- Eventually, an ERTMS level transition will occur (A<> means "*in any possible execution eventually*...").

  ```
  A<> LevelStatus.LevelTransition
  ```

  `LevelStatus` is a Uppaal process with a location `LevelTransition` that represents that a level transition occurred.

- The system will never stop.

  ```
  A[] not deadlock
  ```

2.3.2 Detecting Defective Requirements

Finding out a defective requirement is quite difficult in a very well tested case study like this one. Nevertheless, for the design of new specifications the deployment of a tool like Uppaal is crucial for it to be successful.

3 Conclusions and Future Work

Formalization and validation of requirements, in particular, railway requirements is not an easy task, given the complexity and length of the documents containing them. In this paper we have shown through a case-study how to formalize ERTMS/ETCS requirements that include time information in UML, and how the UML models can be validated using the model checker Uppaal. The case-study presented here is part of a real industrial project where the approach was successfully applied.

Our approach presents several advantages. Formalizing requirements helps engineers to focus on the core elements and uncertainties of their systems requirements, and assist them in finding ambiguities. By using our approach, manufacturers can automatically check the conformance of their equipment against the operation requirements, prior to the more expensive on-track testing. The tools used in the approach can also help in the design and validation of new operation procedures that somehow interact with the standards. In addition, test cases can be automatically generated.

Our plans for future work include extending the UML subset that can be translated to Uppaal models, and to implement a prototype version of the model transformation. We think the scalability of our approach is promising but we will need more and bigger experiments to confirm this hypothesis.

References

[1] G. Behrmann, David A, and Prof. K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *LNCS, Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185, pages 200–237. Springer Verlag, 2004.

[2] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST) 2006*, IEEE Computer Society, pages 125–126, 2006.

[3] Bruno Pagès. BOUML On-Line Manual. http://bouml.free.fr/doc/index.html.

[4] VVAA. System Requirements Specification (subset 026) version 2.3.0. Technical report, European Railway Agency, feb 2006.

[5] European Railway Agency. http://www.era.europa.eu/.

[6] Eurailcheck. http://www.era.europa.eu/core/ertms/Pages/Feasibility_Study.aspx.

[7] Ineco-Tifsa. http://www.ineco.es.

[8] NuSMV. http://nusmv.irst.itc.it/.

[9] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman Publishing Co., Inc. 2003.