Domain Engineering with Event-B: Some Lessons We Learned

Atif Mashkoor, Jean-Pierre Jacquot

Domain Engineering with Event-B: Some Lessons We Learned *

Atif Mashkoor, Jean-Pierre Jacquot

LORIA – Nancy Université
Campus Scientifique, BP 239,
F-54506, Vandœuvre lès Nancy, France
{firstname.lastname}@loria.fr

Abstract. Domain modeling is an important aspect of software engineering. This paper presents our experience of modeling land transportation domain in the formal framework of Event-B. The domain exhibits interesting features, such as high levels of non-determinism, complex interactions, stringent safety properties, etc.. Time is an essential multifaceted feature of the domain. We discuss the problems posed by the introduction of time and some of the solutions we developed. Assessing the validity of a model is also a complex task. We designed a technique based on animation which can help this activity.

1 Introduction

A domain model is a conceptual tool which documents key concepts: majors entities, their inter-relationships, static and dynamic properties, functions, events, and behaviors. There are numerous reasons to perform domain modeling prior to system engineering. For instance, the model constrains the system scope, it helps stakeholders understand better the system requirements, it can be effectively used to verify that the system meets essential properties, and so on.

As for system development, we expect that domain modeling in a formal framework will give practitioners an effective grasp on notions such as correctness and validity.

We present here our preliminary experience with the modeling of a complex domain using Event-B. Event-B [1] is an evolution of classical B method [2] for system-level modeling and analysis of large reactive and distributed systems. We believe the use of Event-B is equally suitable for modeling environments and domains where such systems are assumed to work.

The domain modeled is land transportation. This domain presents a lot of interesting features to push the use of Event-B to some of its limits. For instance, we want to model vehicles moving independently, to understand their interaction (without formal communication), or to analyze situations where traffic jams occur.

* This work has been partially supported by the ANR (National Research Agency) in the context of the TACOS project, whose reference number is ANR-06-SETI-017 (http://tacos.loria.fr), and by the Pôle de Compétitivité Alsace/Franche-Comté in the context of the CRISTAL project (http://www.projet-cristal.org).

We developed our model in the spirit embedded in Event-B. We liberally used refinements, both of machines and of contexts. We give a great deal of attention to proofs. Consequently, we now have a specification of the transport domain where all proof obligations have been discharged. We also had a special interest in the validation of the model. Although it can not be *proven* that a model represents adequately the reality, we have a reasonable conviction that our model is valid. This conviction is prompted by our innovative use of animation of specifications.

During this modeling, we gathered many observations about the use of Event-B on several levels: language, tools, methods, and so on. This paper aims at sharing the salient points of our experience.

The presentation of the paper is organized as follows: next section presents the language and tool we use: Event-B and Brama. Then we present the domain description and its specification. Section four and five describe the lessons which we learned while specifying and validating our domain model respectively. We conclude our paper in section six with some proposed future work.

2 Langage and Tools

2.1 Event-B

Event-B is a formal method for system-level modeling and analysis of large reactive and distributed systems. Main features of Event-B are the use of set theory and first order logic for modeling systems, the use of refinement to represent systems at different levels of abstraction and the use of mathematical proof to verify consistency between refinement levels. Event-B is provided with tool support in the form of an Eclipse-based IDE called RODIN[1] which is a platform for writing and proving Event-B specifications.

The use of Event-B is advocated as a language to reason about system modeling but we believe its use is equally suitable for modeling environments and domains. The reason for this conviction is pretty straightforward; domain engineering is the representation of facts of life and life is all about events happening now and then, therefore Event-B (which all revolves around events) is well suited not only for our purpose but for all such kind of engineering activities.

2.2 Brama

Brama [3] is an animator for Event-B specifications. It is an Eclipse based plug-in for Event-B platform RODIN which can be used in two complementary modes. Either Brama can be manually controlled from within RODIN or it can also be connected to a Flash[2] graphical animation through a communication server; it then acts as the engine which controls the graphical effects.

The figure 1 shows the “classic” interface of Brama running within the RODIN platform. On the left hand side, the events of the animated machine appear. They are in one of two states: enabled or disabled, depending upon the evaluation of the guards to

[1] http://sourceforge.net/projects/rodin-b-sharp/

[2] Flash is a registered trademark of Adobe Systems Inc.

TRUE and FALSE respectively. On the right hand side, the actual values of the machine variables are displayed. The buttons can be used to customize the display or to activate specialized value editors.

The basic user action is to click on an enabled event. At all steps, Brama checks that the values, either provided by the user or computed by the events, do not break the invariants of the machine or the axioms in the contexts.

An animation session begins by setting the values of the constants in different contexts seen (either directly or transitively) by the animated machine. Then, the user must fire the `INITIALISATION` event, which is, at that time, the only enabled event. After this, the user will play the animation by firing the events until there is no more enabled event, or the system enters to a steady loop, or an error occurs (broken invariant or non computable action typically).

A graphical interface can be connected to Brama in the form of a Flash application. Events can be fired from the graphical interface. A mechanism of observers is provided. Expressions and predicates can be individually monitored and their value communicated to the Flash program each time it changes. Last, a scheduler mechanisms allows for the automatic firing of events.

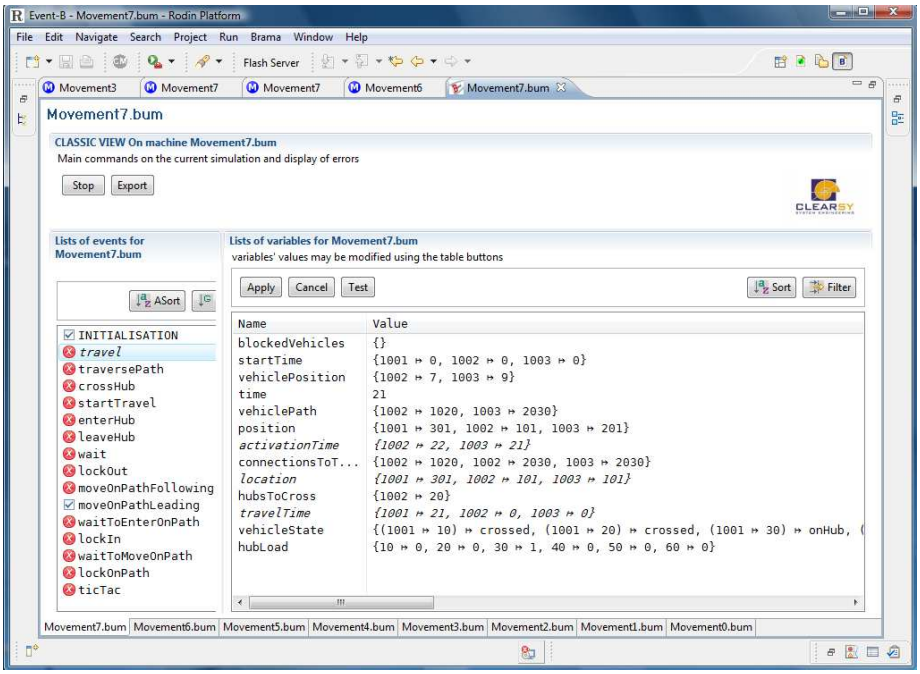

Fig. 1. The Brama animator for RODIN

3 Domain Description

3.1 Domain overview

Our work takes place within the framework of projects TACOS and CRISTAL. These projects aim at studying new transportation systems using autonomous and self-service vehicles known as CyCabs [4]. One of the related concerns is the homologation and certification of such systems for which no standard exists.

The homologation of a vehicle or a system is a process where it is verified that the vehicle meets some requirements. These are derived from the expression and formalization of desirable properties that the whole transport system must incorporate. The issue for software controlled vehicles is to have an expression of those properties amenable to the use of formal verification. The model of the land transport domain is aimed at providing us with the formal expression of these properties.

The model has been defined with Event-B specification language, following the refinement principles advocated by the B-method. We used the ability of Event-B to combine refinement and incremental enrichment of the specification. First, a general definition of transportation networks and the act of moving was given. Then, we introduced properties, one at a time.

Transportation is defined as the movement of people and goods from one location to another with the use of vehicles. We suppose the existence of a network composed of stations (places where vehicles can stop to be loaded and unloaded), intersections (places where roads join), and paths which connect stations and intersection together. Movements are constrained by the topology of the network: a vehicle must follow a sequence of adjacent paths to travel from its origin to its destination.

The general properties we want to express concerning transportation are safety and travel time. First is the idea that collision between vehicles must be avoided. Second is related to the fact that travel time is at the root of nearly all decisions made around transportation, either individually or socially.

3.2 Event-B specification

Our current domain model contains one abstract machine and seven refinements. In parallel with the machines, two contexts are being refined. The first is the context Net which models static properties of the network (its topology, quantities associated to its elements, etc.). The second is the context StartState which helps to set and prove the `INITIALISATION` event of the machines.

It is easier to read and understand the specification when the refinements are grouped into what we call "observation levels." A leap from one level to next occurs when we decompose an abstract event into several ones, corresponding to a finer grain analysis. For instance, the decomposition of the most abstract `travel` event into a sequence of paths traversing and hubs crossing events correspond to a change of observation level. The figure 2 summarizes the four levels:

- First level of observation contains the definition of `travel` event and is specified by machines Movement0, Movement1 and Movement2.

- Second level of observation decomposes `travel` event into `crossHub` and `traversePath` events and is specified by machine Movement3.
- Third level of observation decomposes `crossHub` event into `enterHub`, `leaveHub`, and `wait` events and is specified by machines Movement4 and Movement5.
- Fourth level of observation decomposes `traversePath` event into `waitToEnterOnPath`, `leaveHub`, `moveOnPath` and `waitToMoveOnPath` events. It is specified in Movement6 and Movement7.

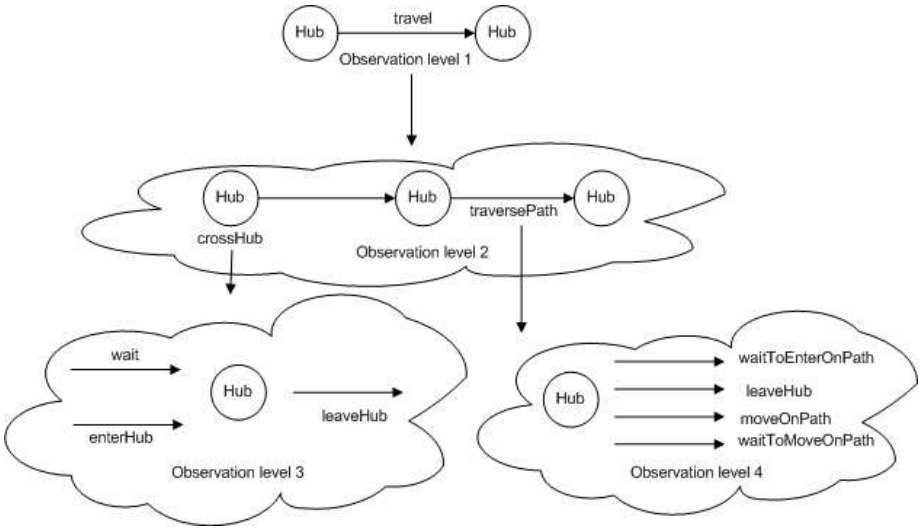

Fig. 2. Levels of observations

New observation levels were introduced when a property could not be expressed within the existing levels.

The first level of observation is about setting up the main domain vocabulary and defining the basic properties of the domain. In the context Net and in its refinements we define the basic vocabulary of the transportation network such as nets, hubs, stations, junctions, connections, paths, routes, etc. In the machine Movement0 we abstractly define the `travel` event as relocation of a vehicle from one place to another. The further refinements at this level introduce a finer topology of the network (junctions, stations, path, routes) and express the property that travel only occur between connected stations.

The second level of observation is about the property that a travel is constrained by the topology of the network. The abstract event is then decomposed into three events (`startTravel`, `crossHub` and `traversePath`) which must occur in a unique sequence to realize a travel.

The third level of observation is motivated by the introduction of the property of non collision at intersections. Such collisions are abstractly defined as the presence of too many vehicles on a hub at the same time. This lead us to decompose the `crossHub` event

as a sequence of `wait`, `enterHub` and `leavehub` events. The choice between `wait` and `enterHub` is controlled by the notions of *hubLoad* (the number of vehicles present on the hub) and *hubCapacity* (the maximal number of vehicles that can be safely present). The second refinement at this level correspond to the introduction of the notion of travel time which does not require a further observation leap.

Fourth level of observation is associated with the introduction of the property of non collision on paths (rear-end type of collision). The event `traversePath` is decomposed into a sequence of `waitToEnterOnPath`, `leaveHub`, `moveOnPath` and `waitToMoveOnPath` events. This models the abstract kinematics of the vehicles.

Following are two interesting properties of the domain.

Collision Avoidance: In the real world, collisions are situations that must be avoided. We choose to model them as breach of an invariant. This way, it is easier to identify the conditions for a well behaving domain through the guards of events.

In real life, collisions can be classified in three types: front, rear and side. Front collisions are implicitly prevented by the typology of the network: paths are oriented and model one way lanes. Side collisions occur at intersections, rear collisions on paths. This prompted us to use two disjoint invariants. The events introduced at the second level made this separation easy to implement.

While a real collision happens when two vehicles are at the same place at the same time, we choose to model it more abstractly on the hubs. Our definition relies on the idea that a hub can carry only a fixed number of vehicles at any one time. So, the invariant to maintain is easily written as:

$$\forall h \cdot h \in Hubs \Rightarrow hubLoad(h) \leq hubCapacity(h)$$

where *hubLoad* is the actual number of vehicles on a hub and *hubCapacity* the maximum number of vehicles allowed on the hub. *Hubload* is a function modified by the events, *hubCapacity* is constant property for each hub. Interestingly, this definition does not require the introduction of time. It abstracts from the kinematics of the vehicles on the hub.

The specification of the absence of rear collision is directly inspired form the natural definition. The corresponding invariant is:

$$\forall v1,v2.v1 \in Vehicles \wedge v2 \in Vehicles \wedge v1 \neq v2 \wedge v1 \in dom(vehiclePosition) \wedge v2 \in dom(vehiclePosition) \wedge vehiclePath(v1) = vehiclePath(v2) \Rightarrow vehiclePosition(v1) \neq vehiclePosition(v2)$$

where *vehiclePosition* is a refinement of the notion of the location of a vehicle on a path.

In a further refinement, positions on paths are modeled as an interval on integers, starting at 0 and ending at *pathLen*. This allowed us to introduce the natural notion of safety distance (*criticalDistance*) that is used in the guards of the moving events. An instance of such a guard is:

$$\forall v \cdot v \in vehiclesOnPath \wedge vehiclePosition(v) > vehiclePosition(vehicle) \Rightarrow vehiclePosition(v) - vehiclePosition(vehicle) > criticalDistance$$

Time: Time is a very important notion in the domain of transportation and our model needs to incorporate it. This notion is known to be tricky to define and to use. In fact, our domain suggests the existence of several flavors of time. One flavor is the travel time, where a clock is only observed at the beginning and at the end of a travel. Another

flavor is the time used in modeling the kinematics where it controls the behavior of the vehicles.

Since Event-B lacks an explicit notion of time, we used the pattern introduced by Cansell et al [5]. In this technique we use natural numbers to model time and a special `ticTac` event to make a global clock (note *time*) advance.

The modeling of time was motivated by the introduction of the `wait` event on the third level. We proceeded in two steps. The first was the introduction of the notion of a clock and the notion of travel time as a difference between two readings of the clock. Although technically realized as a refinement of Movement4, this introduction is logically situated at the first level. The second step was the actual computation of the advance of the clock.

To do this, we modeled the technique used in simulating queue systems. We introduced a timed event queue (noted *activationTime*) which contains the time at which a moving vehicle must perform an event. The following invariants are introduced:

inv1 activationTime $\in$ Vehicles $\pfun$ NAT
inv2 activationTime $\neq \varnothing \Rightarrow$ time $\leq$ min(ran(activationTime))

A new guard is then introduced in the events concerned by time:

vehicle $\in$ dom(activationTime) $\land$ time=activationTime(vehicle)

The action part of the event modifies the event queue accordingly.

The timing pattern is specified by the event `ticTac` as shown by figure 3.

ticTac $\widehat{=}$
REFINES
ticTac
ANY
tic
WHERE
grd1 activationTime $\neq \varnothing$
grd2 t = min(ran(activationTime))
grd3 tic = t $>$ time
THEN
act1 time := t
END

Fig. 3. Event ticTac

A vehicle is introduced in the event queue by the `startTravel` event. It is removed from it when it reaches its destination.

Elements of an earlier version of this specification are discussed in [6]. A recent (verified) version of our specification is available at the following web address:
`http://www.loria.fr/~mashkooa/eventb.htm`

4 Lessons Learned: Specification

4.1 Refinement and observation levels

Refinements and observation levels are distinct concepts. Refinements are the cornerstones of the B-method. They serve two purposes: methodologically, they allow specifiers to concretize the specification, and, technically, they induce proof obligations which guarantee the correction of the development. They give the development a flat structure which may impair its readability.

Observation levels are a way to provide specification with a super-structure which eases the understanding. They reflect either the "natural" structure of the objects or the structure of the computation. For instance, the second observation level in the model reflects the static topology of a network, while the third level is more about the protocol to cross a hub.

The major advantage of thinking in term of observation level stands out when we introduce a new property. This structure provides us with a strong guideline. We experienced it with the introduction of time. The vocabulary and abstract constraints (time is ever increasing for instance) were defined at the first level since this concerned only travels. Then we jumped directly to the third level to define the computation because durations could be associated to events at this level.

4.2 Parallel refinements

While the view of a development as a linear sequence of refinements makes sense in B, where a system is developed, it is far less pertinent in Event-B where an environment in described. Properties are often independent, at least as far as their definition is concerned. We experienced this with time and collision avoidance for instance. It would be even more important when we introduce properties such as energy consumption for instance.

The problem with the linear sequence is that when we introduce a new property, we need to do this into a complex piece of text. Furthermore, most of the text is irrelevant to the property at hand. In some sense, this breaks the sound principle of separation of concerns.

In domain engineering the commonality/variability analysis and decomposition/recomposition of models has always been considered as integral features. In order to be fully compatible with domain engineering, Event-B also needs to have these features. In recent versions of RODIN, two plugins have been proposed for composing Event-B models together: Feature Composition Plugin [7] and Parallel Composition Plugin [8]. They are still prototypes and at infancy stages. We still need to investigate them in more details before recomposition of our models.

4.3 Protocols / ordering constraints in events

Once events are decomposed into smaller events, it is crucial that these events be fired in a strict order in order to maintain the cohesive behavior. For instance, the decomposition of the travel event is thought of as:

$$travel \equiv (startTravel;\ (crossHub;\ traversePath)+)$$

Unfortunately, Event-B does not provide us with traits to express this protocol. Instead, we must make explicit definition of the protocol with the help of control variables and guards in the events. This is complex and a source for errors.

This situation happened each time we introduced a new observation level. So, going from second to third level, we decompose as follows:

$$crossHub \equiv (wait*;\ enterHub;\ leaveHub)$$

To go from third to fourth level, we decompose as follows:

$$traversePath \equiv (waitToEnterOnPath*;\ leaveHub;\ (waitToMoveOnPath|\ moveOnPath)*)$$

We use two basic techniques for controlling the protocols. The first is the introduction of control sets. We used it for the decomposition of travel. The control variables are the set of all hubs and paths the vehicle will have to pass. This technique has the advantage that a variant is quite easy to define but has the drawback to introduce complex computation of the sets. The second technique is the introduction of a notion of state markers, either through an explicit variable or a property such as belonging to the domain of a relation. The advantage of using state markers is their easy definition but their drawback is the difficulty to set variants and generally to connect state markers to invariants.

Although without formal substance, the previous regular-expressions like formulas were of great help to set up the explicit control. It would be a welcome extension of Event-B or of its supporting tools if that kind of expression or others such as the Finite State Machine (FSM) shown by figure 4 for instance, could be stated and be checked against the behavior of the events.

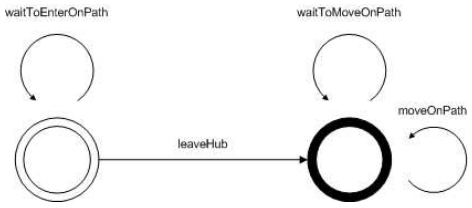

Fig. 4. FSM of traversePath

4.4 Time modeling

Unsurprisingly, the modeling of time raised many questions. We used the patterns proposed be Cansell et al [5] in our models. They are easy to use and blend well within Event-B. They assume a discrete time and in our model, travel time is of that kind. The "computation" of the clock with the timed event queue is cumbersome because it is explicit, but does not lead to specification difficulties. Indeed, a generic pattern emerged to write the refinement:

1. identify an event concerned by time;
2. introduce the standard guard (the same for all event);
3. introduce a substitution of the time event queue; the actual value to substitute is of course dependent on the event.

Kinematics introduce a flavor of continuous time. This raises two questions: Is it legitimate to try to model this with the purely discrete means Event-B provides us? How will it merge with the previous definition of time? The answer to the first question is "Yes" if the model is to be the basis for a software implementation. By essence, computers are discrete machines. A fundamental parameter of any control software for running machines is the frequency of their control loop. So, the actual time will be discrete. As the refinement of the specification to introduce sensible kinematics is not finished at the time of writing, we do not have a definitive answer to the second question. However, preliminary developments have shown that the event queue algorithm can be used. A kind of "fixed tick" can be introduced in the third step of the previous pattern to force events to be fired at a given frequency.

4.5 Safety and liveness properties

Safety: A safety property asserts that nothing bad happens [9]. Safety properties can be specified either as something that should never happen, or as some property that should always hold. We consider the safety property of collision avoidance. It is specified by the invariants of the model. All the invariant preservation proofs have been discharged. We are then assured that no event provokes a collision.

It should however be noted that the previous condition is necessary, but not sufficient to ensure safety in general. Although this does not yet happen in the current state of the specification, it will when kinematics will be fully specified. A moving vehicle should never be allowed to make a move which leads to a collision (i.e. no event should break the invariant), but it must also always be able to react (i.e. there should always be an enabled event). This last condition is similar to the liveness property discussed below.

Deadlock: A deadlock is a state when some processes in a system are halted waiting for something, which can only be done by one of the halted processes, to happen. The same definition can be used to define a traffic jam: vehicles mutually preventing others to move.

Deadlocks are something that implementers must avoid. It is then important to characterize them at the level of the specification.

While deadlocks can be thought of as a situation in Event-B where no event is enabled, i.e., all guards for all events are false, this implicit modeling is not very useful. One of the problems is that Event-B offers us no way to check for the absence of deadlock or to easily derive conditions (invariants) which prevent deadlock. Since traffic jam are a fact of life, we choose to allow them in the specification.

An interesting outcome of this decision was that the introduction of time forced the deadlock situations to "pop up" during some proof obligations. A solution was to introduce new events that model a deadlock, i.e. a traffic jam. We have identified three such situations at present:

- first is when a vehicle tries to enter a station which is already full of parked cars. No vehicle will leave the hub and the moving vehicle is then "locked out";
- second is when vehicle wants to enter a path which is full of other (stationary) vehicles. This vehicle is then "locked in";
- third case is similar to the second case except the vehicle has already begun the traversing of the path. It is then "locked on path."

Modeling deadlocks with special events has at least one advantage. The conditions of the blockage are clearly identified. Implementers who want a particular system to be jam free can derive their invariants from these conditions.

Liveness: Liveness property asserts that something good will happen "eventually" [9]. We have noted above that liveness can be a necessary condition to have safe systems. This notion can also be used for expressing non critical, but desirable properties. In our case, a desirable property is that a vehicle eventually reaches its destination and terminates its travel. This property cannot be formally expressed within Event-B framework because liveness properties involve the temporal concept "eventually" and until now there is no standard temporal constraint definitions for Event-B specifications. Even knowing that, due to traffic jams, the above liveness property is certainly not guaranteed, it would be very useful to be able to express it formally.

As proposed by [10], in order to prove liveness of our model we can prove that our system is non-divergent and enabledness preserving. By non-divergent we mean that newly introduced events do not take control forever and by enabledness preserving we mean that if an event is enabled at abstract level it is enabled at concrete level as well.

Non divergence is usually proven with the help of variants. We introduced the following variant at second level of observation:

card(hubsToCross)+card(connectionsToTraverse)

it states that the newly introduced events `hubsToCross` and `connectionsToTraverse` do not prevent the `travel` event to fire.

This notion of variants is useful to prove non divergence until the event `wait` is introduced at a third observation level. Since a vehicle can wait for indefinite period of time for its turn to enter a hub therefore our variant can not assure us that this event can not take control forever. This is a fact of life: land transportation domain is divergent.

We can prove enabledness preservation of the model by the standard consistency and refinement checking proofs which need to prove that the guards of one or more events in the refinement are enabled under the assumption that guards of one or more events in the abstraction are also enabled.

This discussion on safety and liveness properties indicates that they are complex and tangled issues. It also shows that as far as domain models are concerned, there should not be only one rule like, say, no model shall deadlock, or models shall always be live. The point is that Event-B does not provide us the mean to express cleanly that kind of properties. We consider this as an important shortcoming.

4.6 Language and tools

Our un-conventional use of Event-B and, consequently, of RODIN raised a few issues with the modeling language and the tool support. While the observations discussed be-

low sound negative, we must emphasize the overall quality of the langage and the tools: the major difficulties we encountered were caused by the complexity of the domain and by our own errors.

Considering the tool support, we have two observations:

- RODIN failed too often to discharge automatically obvious proofs. Worse, they were often discharged by a simple click. Although not really tiring, this "activity" becomes boring and, more importantly, very distracting. Many proofs require a lot of concentration; we expect tools to helps rather than distract on this aspect.
- RODIN does not warn when axioms are inconsistent. The detection of contradicting axioms is hard. Now, we rely only on heuristic rules. We suspect a contradiction when we notice that proofs become mysteriously easy to discharge. Then, we introduce an axiom or a theorem such as TRUE = FALSE. Success in the proof signs a contradiction, failure provides us only with reasonable assurance. We know that proving the non contradiction of axioms is non decidable. However, the indication by RODIN that it has detected an inconsistency would be welcomed.

Our work prompted three remarks on the language:

- Refinement is the only structuring mechanism in Event-B. As discussed above (section 4.1), we would appreciate to group machines in other ways. This would not necessarily require a modification of the language, but could be achieved by the tools.
- The internal structure of Event-B machines and context is too flat. Again, a possibility to structure axioms or events into categories would improve greatly the readability. For instance, we classified our axioms in three categories (technical, typing, and property) and found this practise very helpful to maintain clean and readable specification.
- The feature of Event-B which we missed a lot was the notion of sequences. Currently we specify them by using the standard definition of sequences. We consider this only as a patch: it works but it brings clutter to pieces of specifications that are already sufficiently complex.

5 Lessons Learned: Animation

An important part of the specification of the transport domain amounts to model complex behaviors. Some are explicitly defined (the succession of `crossHub` and `traversPath` during a travel for instance), some implicitly (the correct interaction of vehicles at intersections for instance), and other unknowingly (only one vehicle at a time was allowed to travel in an early, erroneous in that case, specification). As a specifier, we confront with three questions: Do our specification models an actual behavior observed in the domain? Do our specification model the behavior we actually want to describe? How do we specify a certain behavior?

These questions correspond to well known software engineering concerns related to three different development activities. First question is about modeling "good" representations of the actual world. Second question concerns the validation of the formal

expression against some already abstracted model. The third question is of a technical nature, related to the expressive power of the language.

We have found out that animation is a very valuable technique to help answer these three questions. While the observation of the animation (which does not need to have fancy graphics) gives a lot of information about the model and helps uncover errors, we also discovered that some activities around animation are also crucial. Activities such as setting up values for the animation (fixing a network actual topology for instance) and inventing scenarios to act or observe provided us with a lot of insight about the specification text, about the model, and even about the traits of the reality we wanted to model.

Unfortunately, we soon discovered that not all specifications could be animated. Not only tools have their limitations such as non supported features of the language for instance, but specification techniques, such as non constructive definitions, often prevent efficient computation of the values. To be useful, an animation needs to be reasonably fast.

We have then designed and described, as rigourously as possible, a set of rules which transform a non animatable specification into one that the animator Brama could animate. One can wonder why we do not produce an animatable specification first. The reason is that our transformation rules "downgrade" the initial specification on two important counts: the specification becomes far less readable and, more importantly, may become unprovable. The transformation process tends to alter and suppress elements that are essential to discharge proof obligations.

Naturally, the relation between the behaviors seen during the animation and the ones described in the initial specification becomes a major issue. To solve the issue, we propose a methodology to include the animation into a rigorous process, which is as follows:

1. start from a *fully* verified specification. This step is essential.
2. for each non animatable trait:
   (a) pick an appropriate rule
   (b) check that the applicability conditions hold
   (c) prove that the argument used in the justification part of the rule is valid
3. if an anomalous behavior is encountered, modify the initial specification, *prove* it to be correct, and restart from step one.

The proof in step 2(c) cannot be carried out in Event-B. Like mathematicians do, we use rigorous arguments, often relying on the fact that the initial specification has been proven correct, to assert that the initial and transformed specification share the same behaviors. In particular, those we want to observe.

The aforementioned animation issues, the rules, and the process are described in more details in [11] and [12]. It should be noted that our choice of tool, Brama, is accidental. At the time, it was the only one able to animate Event-B specifications. More recent tools such as AnimB[3] and ProB [13] are now available and fully compatible with Event-B. While our rules should surely be adapted to these specific tools, we suspect that the general philosophy of animation we have adopted is still valid.

[3] http://www.animb.org

6 Conclusion

We find Event-B an adequate language for domain engineering, however there are still some important questions to address. They are about the language, the tools, and the use of domain models.

About the language, the most limiting factor is the lack of expression of temporal or ordering constraints. We cannot straightforwardly state, and of course prove, properties such as liveness, deadlock freeness, fairness, and so on. Our domain exhibits many natural "protocols" and constraints; we do not think it is exceptional in this respect. Whether Event-B can be extended in this direction, or whether approaches based on mixing formalisms, such as advocated in [14] with CSP||B can be made practical, answers are beginning to appear. We just hope they can be used soon.

Tools are essential to formal methods. Without RODIN, the provers, and Brama, there is no way we could have reached the current state of the specification. However, they are still crude for an industrial usage. The tool we lacked the most was created by our needs with respect to animation. Applying the transformation rules which we invented requires some insight and intelligence (choice of the rule, check of the validity), but also tedious and boring work (text modification). We plan to implement the second part as a form of plugin for RODIN. The boring parts of the transformation do not contain overly complex text manipulation.

We would appreciate also to see tools evolving in the direction of richer visualisation of the specifications. Our notes about observation levels, flat linear structures, parallel refinement, or composition of refinement can be seen through this light. We do not call for incorporating these into the language: it would be unwise to break something that works quite well! Instead, we think that tools based on a better understanding of the needs of the specifiers would be a more promising approach. There is clearly a need for research in this direction.

For the last of our aforementioned questions, we do not have answers. We have hinted at ideas such as deriving invariants of a system form the properties expressed in Event-B domain model. That should be put to test. We intend to do this by studying the practical relation of our domain model with a specification, written also in Event-B[15], of a platooning system. In particular, we would like to study how we can "immerse" the specification of a particular system into the domain model.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2009)
2. Abrial, J.R.: The B Book. Cambridge University Press (1996)
3. Servat, T.: BRAMA: A New Graphic Animation Tool for B Models. In: B 2007: Formal Specification and Development in B, Springer-Verlag (2006) 274–276
4. Baille, G., Garnier, P., Mathieu, H., Roger, P.G.: Le cycab de l'INRIA rhônes-alpes. Technical Report RT-0229, INRIA – Rhônes-Alpes (1999)
5. Cansell, D., Mery, D., Rehm, J.: Time Constraint Patterns for Event B Development. In Julliand, J., Kouchnarenko, O., eds.: 7th International Conference of B Users. Volume 4355 of LNCS., Springer Verlag (2007) 140–154

6. Mashkoor, A., Jacquot, J.P., Souquières, J.: B événementiel pour la modélisation du domaine: application au transport. In: Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09), Toulouse, France (2009) 1–19
7. Gondal, A., Poppleton, M., Snook, C.: Feature composition - towards product lines of event-b models. In: 1st International Workshop on Model-Driven Product Line Engineering, Twente, The Netherlands (2009) 18–25
8. Poppleton, M.: The composition of event-b models. In: 1st International Conference on ASM, B and Z (ABZ2008), London, UK (2008)
9. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering **3**(2) (1977) 125–143
10. Yadav, D., Butler, M.: Verification of liveness properties in distributed systems. In: Second International Conference on Contemporary Computing (IC3'09), Noida, India (2009) 625–636
11. Mashkoor, A., Jacquot, J.P., Souquières, J.: Transformation Heuristics for Formal Requirements Validation by Animation. In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems - SafeCert'09, York, UK (2009)
12. Mashkoor, A., Jacquot, J.P.: Incorporating Animation in Stepwise Development of Formal Specification. Research Report INRIA-00392996, LORIA, Nancy, France (2009) http://hal.inria.fr/inria-00392996/en/.
13. Leuschel, M., Butler, M.: ProB: A model checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods. LNCS 2805, Springer-Verlag (2003) 855–874
14. Evans, N., Treharne, H.E.: Linking semantic models to support CSP||B consistency checking. In: AVOCS'05. (2005)
15. Lanoix, A.: Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008), France (2008-06) 8 pages