Utilizing Event-B for Domain Engineering: A Critical Analysis

Atif Mashkoor, Jean-Pierre Jacquot

▶ To cite this version:

Atif Mashkoor, Jean-Pierre Jacquot. Utilizing Event-B for Domain Engineering: A Critical Analysis. Requirements Engineering, 2011. inria-00590700

HAL Id: inria-00590700
https://inria.hal.science/inria-00590700

Submitted on 4 May 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

Utilizing Event-B for Domain Engineering: A Critical Analysis

Received: date / Accepted: date

Abstract This paper presents our experience of modeling land transportation domain in the formal framework of Event-B. Well-specified requirements are crucial for good software design; they depend on the understanding of the domain. Thus, domain engineering becomes an essential activity. The possibility to have a formal model of a domain, consistent with the use of formal methods for developing critical software working within it, is an important issue. Safety-critical domains, like transportation, exhibit interesting features, such as high levels of non-determinism, complex interactions, stringent safety properties, multifaceted timing attributes, etc. The formal representation of these features is a challenging task. We explore the possibility of utilizing Event-B as a domain engineering tool. We discuss the problems we faced during this exercise and how we tackled them. Special attention is devoted to the issue of the validation of the model, in particular with a technique based on the animation of specifications. Event-B is mature enough to be an effective tool to model domains except in some areas, temporal properties mainly, where more work is still needed.

Keywords Domain engineering · Formal methods · Event-B · Animation · Brama

1 Introduction

Domain engineering is a methodology to document the facts of a particular domain. A domain model, which is the outcome of the domain engineering phase, defines the key concepts of a particular domain, such as major entities, their inter-relationships, static and dynamic properties, functions, events, and behaviors. According to [15], the main activities of the domain engineering phase are: domain analysis, domain design, and domain simulation. While

the domain analysis identifies and captures the domain facts, the latter two concern the translation of these facts into system requirements.

The principle of understanding the domain before specifying the requirements is crucial to software engineering. The idea of having enough details about the environment in which the designed product is assumed to operate is already established in other engineering disciplines. In older engineering disciplines such as aeronautics, electronics, or chemistry, engineers know the domains of their respective fields. By contrast, in software engineering, systems are sometimes developed by people with an incomplete knowledge of their particular domain. Unsurprisingly, the requirements of such systems may be flawed although their correctness is a crucial issue.

System engineering is a methodology to transform users requirements into a system which best satisfies them. There are numerous reasons to perform domain engineering prior to system engineering. For instance, it identifies, models, constructs, catalogs, and disseminates the system scope, it helps stakeholders understand the system requirements better, it can be effectively used to verify that the system meets essential properties, and so on. Furthermore, domain engineering in a formal framework gives practitioners an effective grasp on concepts such as verifiability and validity of requirements.

We present here our preliminary experience with the engineering of a complex domain using Event-B. Event-B [3] is an evolution of the classical B method [2] for system-level modeling and analysis of large reactive and distributed systems. We believe that the use of Event-B is equally suitable for modeling environments and domains where such systems are assumed to work.

The domain under consideration for this work 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 when there is no explicit communication between vehicles, or to analyze situations where traffic jams occur. We had a simplified version of the road traffic domain in mind when specifying and the model reflects this. Since the model does not assume specific features of the vehicles or of their control, it is most likely usable for other systems such as train systems or baggage conveyors.

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 special interest in the validation of the model which was achieved 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: the next section presents the main motivation for this paper followed by a section on language, techniques and tools we have used. Then we present the domain description and the specification. Sections 5 and 6 describe the lessons which we learned

while specifying and validating our domain model respectively. In the end, we present the related work and finally we conclude our paper in section 8 with some proposed future work.

2 Motivation

Most customers express their requirements either in natural language or in terms of scenarios. Most of the requirements engineering methodologies are therefore non-formal or semi formal. One of the problems with less formal techniques is that they may be ambiguous, which makes the requirements engineering phase error-prone.

With the help of well-defined syntax and semantics, formal specifications can concisely express the software requirements. However, due to their complex structures and mathematical contents, they are difficult to read and understand for customers. Actually, formal specifications may sometimes not be able to intuitively reflect the concepts and behaviors of systems in the real world. The conventional issue of validation may therefore impair the requirements engineering phase.

An earlier involvement of customers and use of formal techniques in software development may be a solution to the aforementioned requirements engineering problems and a domain model is the right artifact to start with. A formal domain model precisely specifies the domain facts and with the help of techniques such as animation, we can demonstrate the model to customers for their timely feedback. Thus, we can build a "mental-bridge" between complex formal specifications and their perception in the real-world. Our rigorous validation technique, discussed later in the paper, is based on animation and involves customers in the software development process right from the start; consequently errors can be detected right on the spot.

3 Language, Technique and Tool

3.1 Event-B

Event-B is a formal language for modeling and reasoning about large reactive and distributed systems. Event-B is based on set theory and standard first-order predicate logic. Event-B is provided with tool support in the form of a platform for writing and proving specifications called Rodin[1].

An Event-B model is composed of two constructs: machine and context. Machines, which define the dynamic behavior of the model, contain the system variables, invariants, variants, and events. Variables are typed, their values may be integers, sets, relations, functions or any other set-theoretical construct. Invariants define the state space of the variables and their safety properties. Variants are related to the correction of refinements.

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

An event, which defines a transition from one state to another, can be defined as a binary relation built on the state set. This relation is composed of the guards and actions of the event. A guard is a predicate and all the guards together construct the domain of the corresponding relation. An action is an assignment statement to a state variable and is achieved by a generalized substitution. Combined together, all the actions form the range of the corresponding relation. The actions of a particular event are executed simultaneously and non-deterministically. Contexts, which define the static elements of the model, contain carrier sets, constants, axioms, and theorems. The last two are predicates expressed within the notation of first-order logic and set theory.

There are several relationships between machines and contexts: refinement, extension, and visibility. A machine can be a refinement of one, and only one, machine. It then contains a more detailed or concrete description of the model. A context can extend one, and only one, context. It contains the static pieces of information of a model associated to a refinement. A machine can see several contexts, that is, use their names and properties; a context can be seen by several machines.

Event-B embeds the concept of refinement which is then the basic element of specification development processes. A refinement consists in introducing either new variables or new events. When appropriate, an abstraction invariant, often called *gluing-invariant,* relates the new variables to the abstract ones. Individual events can also be refined by strengthening their guards and adding actions to the new variables. The same abstract event can be refined into several concrete ones. New events can be introduced, too. Formally, they are refinements of the `SKIP` event. Most often, new events express how an abstract event is decomposed by a sequence of more concrete events. Such a decomposition may lead to a divergent model: a model where the sequence of concrete events never reaches its end and then prevents the abstract event from firing. Variants may be explicitly introduced to guarantee the absence of divergence. They are natural number expressions on the state of the model. When declared as “convergent,” concrete events must strictly decrease the variant; when declared as “anticipated,” they must not increase the variant.

The semantics of refinement are given by proof obligations. Proving a refinement correct amounts to proving that concrete events maintain the invariant of the abstract model, maintain the abstraction invariant, and, when appropriate, decrease variants monotonically.

In practice, it is often useful to think in terms of reification of variables and of decomposition of an event into several smaller ones. This point of view helped us to organize the development steps and to get a better rationale for each refinement.

3.2 Animation

The main goal behind animation is to demonstrate the requirements narrated in the specification document. This demonstration facilitates the understanding and correction of complex specifications. It is an approach which lets the specifier analyze the specification against possible sets of behavioral scenarios. These behavioral scenarios, which in turn are sequences of events, constitute the behavior of the specification. For instance, different scenarios can happen when a vehicle crosses an intersection, depending on whether other vehicles are already on or approaching the intersection.

To use the animation for validation purposes, all typical behavioral scenarios of the specification should be analyzed. The behavioral scenarios, which define the functional behaviors of the system through a sequential execution of events, are animated by feeding some initial values to animators at startup. These startup values may not be required by animators which provide these values to specifications themselves. During the animation, we can observe whether the scenario runs as expected, without violating invariants, guards or post-conditions. The animation process is continued until all the scenarios are exhausted or some error perturbs the intended course of events.

In an ideal world, all typical scenarios should be animated. However, depending upon cost and timing constraints, conducting animation on selective scenarios, which are considered critical for the validation of the specification, may be an effective approach.

3.3 Brama

Brama [42] is an animator for Event-B specifications. It is an Eclipse based plug-in for the Event-B platform Rodin. Brama can be used in two complementary modes. Either Brama can be manually controlled from within the Rodin interface or it can be connected to a Flash[2] graphical interface through a communication server; it then acts as the engine which controls the graphical effects.

figure 1 shows the standard Rodin interface of Brama. It provides us with a simple, but effective, visualization of the behavior through two windows which synthesize the current state of the animation. On the left hand side, we can see all the events and which of them are enabled, that is, their guard is true. On the right hand side, we can read the values of the state variables. An enabled event is fired by simply clicking on it. As the tabs indicate in the lower part of the window, it is possible to visualize the current state of the animation on different refinements. The buttons allow for adapting the visualization and editing values which act as parameters for the events.

A typical animation session begins by setting the values of the constants in the different contexts seen (either directly or transitively) by the animated machine. Then, the user must fire the `INITIALISATION` event, which is, at

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

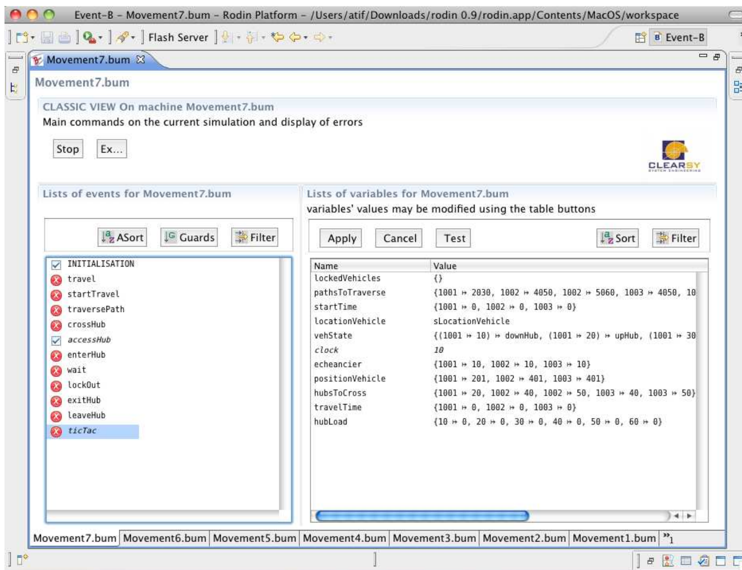

Fig. 1 The Brama animator for Rodin

that time, the only enabled event. After this, the user plays the animation by firing the events until there are no more enabled events, or the system enters a steady loop, or an error occurs (broken invariant or a substitution that Brama does not know how to compute).

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

4 Domain Description

4.1 Domain overview

Our work takes place within the framework of the projects TACOS[3] and CRISTAL[4]. These projects aim at studying new transportation systems using autonomous and self-service vehicles known as CyCabs [5]. CyCabs are small computer-controlled electric cars. They can move in three modes: driven by a human, driven by their inboard computer, or within a *platoon*. In this last mode, several CyCabs assemble as a train without material connections between the cars. Except for the leader which can be manually driven, platoon

[3] http://tacos.loria.fr

[4] http://www.projet-cristal.org

members are controlled by systems which aim at keeping cars as close as possible to each other and at following as closely as possible the trajectory of the leader. CyCabs can be used as the basis of a car-sharing system in urban areas. There are several scenarios on the operation of such systems. All share two important features. CyCabs will move in the public space, possibly on dedicated lanes, and will have strong interactions with other road users. Driverless moving modes and platooning are necessary for providing customers with new services, such as transient buses, relocation of CyCabs between stations in order to adjust vehicles and parking availability during the course of the day. These features imply that systems and vehicles need to be certified.

The certification of a vehicle or a system is a process where it is verified that the vehicle meets minimal requirements which allow it to operate within a certain domain. These requirements 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 these 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 the 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, called a *hub,* to another with the use of vehicles. We suppose the existence of a network composed of *stations* (hubs where vehicles can stop to be loaded and unloaded), *junctions* (hubs where roads join), and *paths* which connect stations and junctions 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. The first is the idea that collision between vehicles must be avoided. The second is related to the fact that travel time is at the root of nearly all decisions made about transportation, either individually or socially.

4.2 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 the 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 the 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 path traversing and hub crossing events corresponds to a change of observation level. Figure 2 summarizes the four levels:

1. The first level of observation contains the definition of a `travel` event and is specified by machines `Movement0`, `Movement1` and `Movement2`.
2. The second level of observation decomposes `travel` events into `crossHub` and `traversePath` events. This is specified by machine `Movement3`.
3. The third level of observation decomposes `crossHub` events into `enterHub`, `leaveHub`, and `wait` events. This is specified by machines `Movement4` and `Movement5`.
4. The fourth level of observation decomposes `traversePath` events into `waitToEnterOnPath`, `leaveHub`, `moveOnPath` and `waitToMoveOnPath` events. This is specified in `Movement6` and `Movement7`.

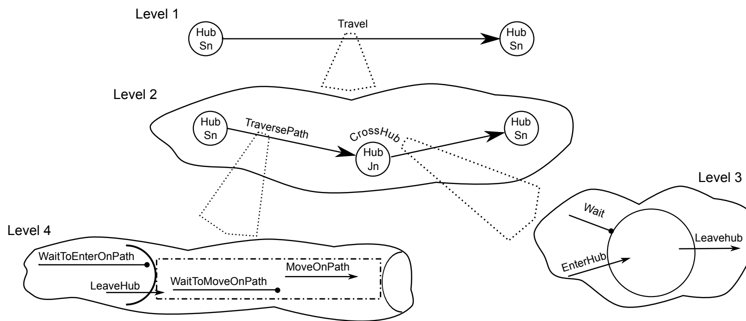

Fig. 2 Levels of observations

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 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, paths, routes) and express the property that travel only occurs between connected stations.

The second level of observation is about the property that 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 traveling event.

The third level of observation is motivated by the introduction of the property of non collision at hubs. Such collisions are abstractly defined as the pres-

ence of too many vehicles in 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 `hubLoad` (the number of vehicles present on the hub) and `hubCapacity` (the maximal number of vehicles that can be safely present on the hub). The second refinement at this level corresponds to the introduction of the question of travel time, which does not require a further observation leap.

The 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 which we model:

Collision Avoidance: In the real world, collisions are situations that must be avoided. We chose to model them as a breach of an invariant. A well-behaved domain is then one where no event breaks the invariant. Since events' semantics is based on weakest preconditions, the events' guards are then a good description of the conditions that a domain must meet to be well-behaved.

In real life, collisions can be classified into three types: front, rear and side. Front collisions are implicitly prevented by the topology 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 in the same place at the same time, we chose to model it more abstractly on the hubs. Our definition relies on the idea that a hub can only carry a fixed number of vehicles at a time. So, the invariant to maintain is easily written as:

$$\forall\, h \,.\, h \in \mathsf{Hubs} \Rightarrow \mathsf{hubLoad}(h) \leq \mathsf{hubCapacity}(h)$$

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

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

$$\begin{aligned}&\forall\, v1,\, v2 \,.\, v1 \in \mathsf{Vehicles} \wedge v2 \in \mathsf{Vehicles} \wedge v1 \neq v2 \wedge\\ &v1 \in \mathrm{dom}(\mathsf{vehiclePosition}) \wedge v2 \in \mathrm{dom}(\mathsf{vehiclePosition}) \wedge\\ &\mathsf{vehiclePath}(v1) = \mathsf{vehiclePath}(v2) \Rightarrow \mathsf{vehiclePosition}(v1) \neq \mathsf{vehiclePosition}(v2)\end{aligned}$$

where `vehiclePath` signifies the current path of the vehicle and `vehiclePosition` is a refinement of the location of a vehicle on the path. This invariant assumes two facts: `vehiclePosition` is a partial function whose domain is the set of vehicles actually engaged on a path, and different paths never share locations. This last situation would be modeled as a junction.

In a further refinement, positions on paths are modeled as an interval between integers, starting at 0 and ending at `pathLen`. This allowed us to

introduce the natural concept of safety distance (`criticalDistance`) that is used in the guards of the moving events. An instance of such a guard is:

$$\forall\ v\ .\ v \in \mathsf{vehiclesOnPath} \wedge \mathsf{vehiclePosition}(v) > \mathsf{vehiclePosition}(\mathsf{vehicle}) \Rightarrow \mathsf{vehiclePosition}(v) - \mathsf{vehiclePosition}(\mathsf{vehicle}) > \mathsf{criticalDistance}$$

We use integers for positions instead of real numbers because we want our model to be fully provable within current event-B tools. Presently, the provers available within Rodin are restricted to integers. The issue of discretization is a complex one which we have addressed in another work [46].

Time: Time is a very important paramater in the domain of transportation and our model needs to incorporate it. This parameter 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 travel time, where a clock is only observed at the beginning and at the end of a travel. Another flavor is continuous time which is used in modeling the kinematics where it controls the movement of the vehicles.

Since Event-B lacks an explicit concept of time, we used the timing patterns for Event-B proposed by Cansell et al [13]. In this technique we use natural numbers to model time and a special `ticTac` event to make a global clock (`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 observation 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 (`activationTime`) which contains the time at which a moving vehicle must perform an event. The following invariants are introduced:

$$\mathsf{activationTime} \in \mathsf{Vehicles} \nrightarrow \mathsf{NAT}$$
$$\mathsf{activationTime} \neq \varnothing \Rightarrow \mathsf{time} \leq \min(\mathsf{ran}(\mathsf{activationTime}))$$

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

$$\mathsf{vehicle} \in \mathsf{dom}(\mathsf{activationTime}) \wedge \mathsf{time} = \mathsf{activationTime}(\mathsf{vehicle})$$

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

$$\mathsf{activationTime} := \mathsf{activationTime} \lhd\!\!- \{\mathsf{vehicle} \mapsto \mathsf{time} + \mathsf{timeInc}\}$$

where *timeInc* is an increment dependant of the event considered. It can be a constant, an arbitrary value, or a computation on the event queue.

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

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

Elements of an earlier version of this specification are discussed in [33]. A more recent verified version of our specification is available at the following web address: `http://dedale.loria.fr/?q=re-spec`.

```
EVENT ticTac REFINES ticTac ≙
ANY
 tic
WHERE
activationTime ≠ ∅ ∧ tic = min(ran(activationTime)) ∧ tic > time
THEN
time := tic
END
```

Fig. 3 Event ticTac

5 Lessons Learned: Specification

5.1 Assumptions vs. requirements

One of the main reasons to use mathematical formalisms and tools is to explicitly define the elements of interest. At the time when domain modeling is of importance, the focus is on "requirements" and "assumptions." Traditionally, the former denotes what a particular system is expected to do, and the latter what the system can expect from its operating environment [47].

In B, which was designed as a language to specify and develop systems, functional requirements are expressed by invariants. In Event-B, where we are modeling an environment which controls the system, we cannot locate the properties of interest as easily. Part of the problem is that it is possible in Event-B models to mix system and environment properties. While always expressed as predicates on the state, properties can be found in three places: in the invariants of the machines, in the axioms of the contexts, or in the guards of the events. It may be then interesting to relate the type of assumptions with their location in the text of the specification.

A domain model is composed of different assertions about the particular domain. So these assertions are used as assumptions by systems operating within the domain. A system designer uses these written assumptions, but also unwritten, implicit, assumptions. Of course, the goal of a domain model is to make explicit as many assumptions as possible which are essential for the correct operation of a system. In our Event-B models, these could be classified into structural facts, behavioral laws, and enforceable properties.

Contexts in Event-B are used to describe the constants in a model. So, they contain all the structural facts. For instance, it is in contexts that a transportation network is described as a set of nodes (hubs) and vertices (paths), that hubs are partitioned into stations and junctions or that vehicles are constrained to bounded speeds, accelerations, and decelerations. Axioms in the contexts allow us to define the properties of the structure. For instance, routes are defined as sequences of contiguous paths, with each hub visited only once, the first path starting from a station, and the last path leading to a station.

Behavioral laws are described by events. More precisely, as assumptions, they are located in the guards of the events. For instance, the law which states that travel occurs only between stations, or the one which states a travel

is associated to a route are both found in the guards of the travel event, respectively in `Movement1` and `Movement2` refinements.

We refer to enforceable properties as those properties which are necessary to have a well-behaved model. Collision avoidance is high among enforceable properties in the transport domain, for instance. Such properties fall in between requirements and assumptions: a system working into the domain can assume the property, but must guarantee to keep it unbroken. Quite obviously, such properties are expressed by invariants.

Whether a particular domain assumption should be expressed as a behavioral law or as an enforceable property is a difficult question which has no clear-cut answer. If we consider the issue of collisions, we used an invariant, but we could have introduced a special `collide` event. Formally, there is a strong relationship between the two descriptions: the guard of the hypothetical `collide` is the negation of the invariant. The choice between the two expressions depends on the kind of system one has to develop. For instance, developers of a road traffic monitoring system will likely prefer to have `collide` events since their system will have to deal with such situations. Developers of a traffic light control system will likely prefer the invariant expression as it is one of the goals of their system.

Domain models are reference documents. So, they will often be read by people who need to check some intuitive assumption or to collect assumptions relevant to a certain part of the system. It is more difficult to extract assumptions from a model than to introduce them. This is connected to the traditional issue of readability of formal texts. Even assuming readers have an equal command on the formalism as writers, the former need to infer the semantics that the latter has only to write down.

Structural facts and enforceable properties are expressed by axioms and invariants respectively, so they are well localized, as a unique syntactic expression, in the text of the specification. The only confusing problem comes from the typing formulae which are part of axioms and theorems: most are purely technical but some convey information that can be seen as assumptions. For instance, the structural property that a connection belongs to only one transport network can be written either as

```
typing obsNetConnections ∈ Connections → Nets
property ∀ c . c ∈ Connections ⇒ card(obsNetConnections[{c}] = 1
```

or as

```
typing obsNetConnections ∈ Connections ↣ Nets // total injection
```

The assumption is less conspicuous in the second expression.

The extraction of a behavioral law is the real difficulty. The problem comes from the scattering of the expression of the law into the guards of several events. For instance, the assumption that a vehicle moves on a path only when it has some room to do so is scattered into 4 events.

One way to ease the extraction is to restrict the model development to the introduction of only one behavioral law per refinement and to document the

rationale for the refinement. Since Event-B supports small refinement steps, there is not much cost in refining slowly.

The very positive side of using Event-B for modeling assumption lies in the fact that consistency of assumptions can be assessed. When an assumption is expressed by an invariant, discharging the standard proof-obligations of Event-B ensures that the assumption is consistent with the model. Failure to discharge the proof-obligation is not a formal proof of inconsistency, but can conservatively be interpreted as such. Assumptions expressed as axioms are in the opposite situation: we can show inconsistency but cannot prove consistency. In that case, the proof-obligations are restricted to axiom's well-formedness and well-typing. This point is further elaborated in the section 5.7 as an observation on the tool.

5.2 Refinements vs. 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 correctness of the development. They give the development a flat structure which may impair its readability.

Observation levels are a way to provide a specification with a super-structure which eases its understanding. They reflect either the "natural" structure of the objects or the structure of the behavior. 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 terms of observation levels becomes apparent 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. Next we jumped directly to the third level to define the computation because durations could be associated to events at this level.

5.3 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 is described. Properties are often independent, at least as far as their definition is concerned. We experienced this with time and collision avoidance.

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. For instance, if we wanted to introduce a notion of energy consumption, we would like to start the new feature analysis as written in figure 4. From this, we could refine the

notion along the observation levels and merge the resulting model with the current specification.

```
INVARIANT
meter ∈ Vehicles → int // energy meter
energyConsumed∈ Vehicles → int
EVENT travel REFINES travel ≘
ANY
 vehicle , newLocation, meterReadingAtStart
WHERE
 vehicle ∈ Vehicles ∧ newLocation∈ GlobalLocations ∧
newLocation ≠ location( vehicle ) ∧ meterReadingAtStart ≤ meter(vehicle)
THEN
 location ( vehicle ) := newLocation
energyConsumed := meter(vehicle) − meterReadingAtStart
END
```

Fig. 4 Introduction of Energy consumption: what we want

Instead, Event-B's flat refinement structure would force us to write the `travel` event as illustrated by figure 5 and to introduce in all other events a dummy action of the form:

meter : | meter'(n) $\geq$ meter(n)

This action simply states that `meter` is susceptible to be modified by future refinements. Even if the addition of such an action does not pose any problem, it tends to clutter the text and to cause distraction.

```
INVARIANT
meter ∈ Vehicles → int // energy meter
energyConsumed∈ Vehicles → int
EVENT travel REFINES travel ≘
ANY
 vehicle , newLocation, r , origin , destination , meterReadingAtStart
WHERE
r ∈ routes ∧
// ...
// 14 lines of guards
// ...
meterReadingAtStart ≤ meter(vehicle)
THEN
 location ( vehicle ):= newLocation
travelTime( vehicle ) := time − startTime(vehicle)
activationTime := {vehicle} ⩤ activationTime
speed( vehicle ) := 0
 acceleration ( vehicle ) := 0
energyConsumed := meter(vehicle) − meterReadingAtStart
END
```

Fig. 5 Introduction of Energy consumption: what we have

In domain engineering the commonality/variability analysis and decomposition/recomposition of models have always been considered as integral fea-

tures. The example shows why such features would be welcomed in developing a domain model. Currently Rodin lacks tools to compose models. However, for its recent versions, several plugins have been proposed for composing Event-B models together: Feature Composition Plugin [19], Parallel Composition Plugin [40] and Shared Event Composition Plugin [43]. They are still prototypes and at early stages of development. We need to investigate them in more detail before we can use recomposition in our models.

5.4 Protocols/ordering constraints in events

Once events are decomposed into smaller events, it is crucial that these events be fired in a strict order so that a consistent behavior is modeled. 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 of errors.

This situation happens each time we introduce 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 these for the decomposition of travel. The control variable is the set of all hubs and paths the vehicle will have to pass through. The next hub to cross or the next path to traverse is easily defined as the member of the control set which is related to vehicle's position. This technique has the advantage that a variant is quite easy to define, but has the drawback of introducing 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. This can be seen as a form of coding a state machine. 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 formulae 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 could be stated and be checked against the behavior of the events. Diagrammatic notations, such as the structure diagrams of Jackson System Development (JSD) [23] or formalism like Communicating Sequential Processes (CSP) [22] could be used.

5.5 Time modeling

Unsurprisingly, the modeling of time raised many questions. We used the timing patterns for Event-B proposed by Cansell et al [13] in our models. 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:

- identify an event concerned by time;
- introduce the standard guard (same for all events);
- introduce a substitution of the timed 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: (1) is it legitimate to try to model this with the purely discrete means Event-B provides us? and (2) 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.

Technically, the refinement of traversePath to introduce the kinematics behavior did not pose many problems. The basic idea was to use the pattern presented above with a kind of “fixed tick” in the third step. The kinematic functions are modeled as axioms in the context specific to vehicles.

5.6 Safety and liveness properties

Safety: A safety property asserts that nothing bad happens [26]. Safety properties can be specified either as something that should never happen, or as some property that should always hold. Consider the safety property of collision avoidance. It is specified by the invariant of the model. All the invariant preservation proofs have been discharged. We are then assured that no event precipitates a collision.

It should be noted, however, 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 later.

Deadlock: A deadlock, in computation, is a state when some processes in a system are halted waiting for something to happen which can only be triggered by one of the halted processes. In transportation a similar phenomenon exists and is referred to as gridlock, which describes an inability to move on

a transport network (i.e. traffic jams). Both deadlock and gridlock 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., guards of all events are false, deadlock freeness would mean that some vehicles can always move i.e. at least one event is enabled all the time, such as stated with the following invariant:

$$G(E_1) \vee G(E_2) \vee ... \vee G(E_n)$$

where $G(E_i)$ is the guard of the event E_i.

In the transportation domain, we can always experience the situation of traffic jams which may prevent all vehicles from moving. Since gridlock is a fact of life, we choose to allow them in the specification. At a theoretical level, with the introduction of `wait`, we can say that a vehicle can wait in such situations, and at least this event can always be fired, but this is not an elegant solution. At the specification level, Rodin does not allow any deadlock freeness proof and it either needs to be done manually or with the help of a model checker, such as ProB [29].

As an impact of the decision to allow gridlock in the model, later in the specification, the introduction of time forced the gridlock situations to "pop up" during some proof obligations. A solution was to introduce new events to model these gridlock situations. We have identified three such situations at present:

1. when a vehicle needs 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";
2. when a vehicle needs to enter a path which is full of other (stationary) vehicles. This vehicle is then "locked in";
3. the third case is similar to the second case except the vehicle has already begun traversing the path. It is then "locked on path".

Modeling gridlock 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.

Have we identified all the gridlock situations? This question can be answered either way. We can answer "Yes" if we consider only the formal model. The *locked* events are direct consequences of the time model that is used in the domain specification. They are necessary to discharge the proofs related to the property that time is ever increasing. We can answer "No" if we consider the reality of which the specification is an abstract model. There could be other gridlock situations, associated with other notions of "progress" of the state of the model which are not yet described. The point is that the proof obligations of Event-B catch the gridlocks implied by the model.

Liveness: The liveness property asserts that something good will happen "eventually" [26]. We have noted above that liveness can be a necessary condition

to have systems which guarantee a given safety property. 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 the Event-B framework because liveness properties involve the temporal concept "eventually;" until now there is no standard way to define temporal constraints in Event-B specifications. Even so we know 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.

However, as proposed by [45], in order to prove the 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 the second level of observation:

card(hubsToCross)+card(connectionsToTraverse)

where `hubstoCross` (resp. `connectionsToTraverse`) is the set of hubs (resp. paths) that the traveling vehicles have still to cross (resp. traverse) to reach their destinations. One of the sets loses one of its elements each time a vehicle progresses on its travel. The proof that the newly introduced events `crossHub` and `traversePath` decrease the variant is a guarantee that they do not prevent the `travel` event to fire.

This notion of variant is useful to prove non divergence until the event `wait` is introduced at the third observation level. Since a vehicle can wait for indefinite periods of time for its turn to enter a hub, our variant cannot assure us that this event cannot take control forever. This is a fact of life: the 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 the 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, for example, no model shall deadlock, or models shall always be live. The point is that Event-B does not provide us with the mean to express cleanly those kind of properties. We consider this as an important shortcoming.

5.7 Language and tools

Our unconventional use of Event-B and, consequently, of Rodin raised a few issues with the modeling language and the tool support. While the observations discussed below sound negative, we must emphasize the overall quality of the

language 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:

1. Rodin failed too often to automatically discharge obvious proofs, even those so obvious that it took a simple click by the user to direct their completion. This becomes tedious and very distracting. Particularly annoying are the numerous sub-goals akin to type-checking that are generated by the deduction rules and discharged with a click. They tend to disrupt the concentration required by tricky proofs; we expect tools to help rather than distract on this aspect.
2. 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:

1. Refinement is the only structuring mechanism in Event-B. As discussed above (section 5.2), grouping machines in other ways would be appreciated. This would not necessarily require a modification of the language, but could be achieved by the tools.
2. The internal structure of Event-B machines and contexts is too flat. Again, a possibility to structure axioms or events into categories would improve greatly the readability. For instance, we classified our axioms into three categories (technical, typing, and property) and found this practice very helpful to maintain clean and readable specification.
3. 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 parts of specifications that are already sufficiently complex.

6 Lessons Learned: Animation

6.1 Animation of specifications

An important part of the transport domain model amounts to specifying complex behaviors. Some are explicitly defined (e.g., the succession of `crossHub` and `traversePath` during a travel), some implicitly (e.g., the correct interaction of vehicles at intersections), and other unknowingly (e.g., only one vehicle at a time was allowed to travel in an early, erroneous in that case, specification). As a modeler, we are confronted with three questions: does our model

specify an actual behavior observed in the domain? Does our model specify 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. The first question is about modeling "good" representations of the actual world. The 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 discovered that animation is a very valuable technique to help in answering 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 (e.g., fixing a network's actual topology) 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. Of course, animation alone is not sufficient to decide if a model is "good" but, by allowing concrete observations of the model behavior, it facilitates the comparison between model and reality.

It should be noted that our choice of tool, Brama, is contingent. At that time, it was the only one able to animate Event-B specifications. More recent tools such as AnimB[5] and ProB [29], are now available and fully compatible with Event-B. While our proposed heuristics (discussed in the next subsection) should surely be adapted to these specific tools, we suspect that the general philosophy of animation we have adopted is still valid.

6.2 Animation for specification validation

Soon we discovered that not all specifications could be animated for validation purposes. Not only do 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 then designed and described, as rigorously as possible, a set of heuristics which transform a non-animatable specification into one that the animator Brama could animate. One can wonder why we do begin by producing an animatable specification. The reason is that our transformation heuristics "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 for proofs.

Naturally, in the presence of some undischarged proof obligations, the relation between the behaviors seen during the animation and the ones described

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

in the initial specification becomes a crucial issue. To solve this, we propose a stepwise framework to include the animation in a rigorous validation process. Our proposed validation methodology, for each observation level, is summed up by figure 6 and as follows:

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

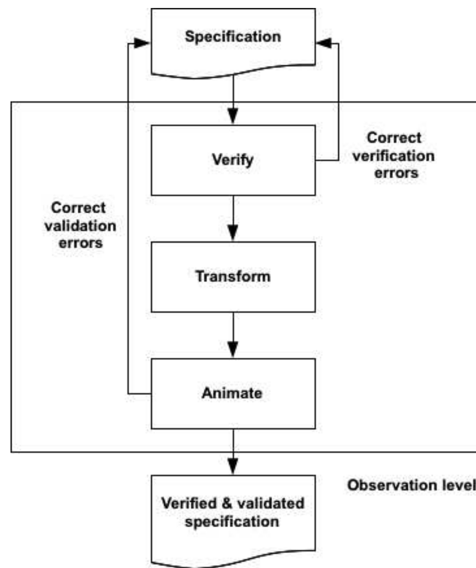

Fig. 6 The stepwise validation framework

In our proposed framework, a verified specification must be the starting point of the validation process. If there are verification errors in the specification they must be corrected before proceeding towards the transformation step. Our belief is that there is no point in engaging ourselves in a costly series of validations on a piece of code which is non-verifiable. Furthermore, failures to discharge proof-obligations are rather frequent and reveal problems in the specification. Although most problems are minor and easily corrected by small modification in some expressions, the Event-B text is generally not exactly the same at the end of the verification than at its beginning. The model then is different. Last, the proof of correctness for the application of some heuristics depends on the fact that they are applied to a verified specification.

As soon as all proof-obligations have been discharged, we proceed further towards the animation step. During this procedure, whenever we discover any element in the specification which is non-animatable, we inspect the problem and try to match the case with the list of our proposed transformational heuristics. This inspection and matching practice includes checking if the application condition defined by the heuristic holds, and also that the use of this heuristic can be justified. This justification can either be provided in the form of a formal proof within Event-B (discharge of a poof-obligation) or by a rigorous argument which generally uses the fact that the process starts from a verified model.

The fact that transformations have been applied means now that the specification is animatable. Animation demonstrates the behavior of the specification. If the demonstrated behavior is as per expectations, then we have the verified and the validated specification in our hands. However, if this is not the case and a closer look at the specification has revealed deviations from the intended behavior, then we need to go back to the initial specification and would have to correct the anomalous behavior. This triggers the loop, i.e., re-proving, re-application of the heuristics, and re-animation until the specification conforms to actual expectations.

Of course, for this process to be valid, we need to ensure that the initial and transformed specifications are equivalent in some way. Since the heuristics are not "semantic preserving" in the strong sense, we have to define an ad-hoc semantics and preservation properties. Intuitively, we need to guarantee that: "anything that is observed during the animation of the transformed text would have been observed on the animation of the initial text." Another way to state it is that the behavior of the transformed specification is a subset of the behavior of the initial specification. This leads to our giving a formal definition of behaviors as sequences of states and events, and of the relationship between behaviors of two specifications. A formal notion of "shared behavior" provides us with the basis for the preservation property. Those definitions are easily related to the actual observations on animations: enabled events (*enabledness property*), and state values (*reachability property* and *closure property*).

The current heuristics have been analyzed. Some heuristics can be proven to preserve behavior in all situations, either because they are strongly semantic preserving, or because, like invariant removal, they equate the shared behaviors with the initial behaviors. Others lead to the generation of proof-obligations which, when discharged, ensure the preservation property.

We do not discuss transformational heuristics here. For details, see this research report [31] which discusses the symptoms, transformations, cautions, justifications, and proofs of all these transformational heuristics. Two complementary case studies, employing our proposed approach of stepwise validation, can be found in [32] and [30].

6.3 Animation for features exploration

Primarily, we have used animation as a quality assurance activity, i.e., to validate and gain confidence in our specifications. It is closely related to prototyping. The benefit of this approach is that we can convert the specification into a prototype without translating it into a program. It then acts as a quick and low cost validation technique.

The use of animation *after* the proofs of both the model and application of heuristics is essential to get a trustworthy validation. However, we have discovered that animation is also a useful tool when used *before* the proofs. In such cases, animation is used to explore new features.

The introduction of a feature raises three issues: (1) the definition of the feature, (2) its formal specification, and (3) its consistency with the current model. Regarding the first issue, animation provides us with a good intuitive understanding of how the model "works." This helps to realize how the feature can be introduced and how it will fit into the model. Expressing a feature into guards, actions, axioms, or invariants is a difficult exercise, even for simple behaviors. Small variations in the formal text may lead to "incorrect" behaviors. Using animation to check that the formal text specifies the intended feature before embarking on the verification is cost-effective. Regarding the third issue, animators like Brama, which verify continuously that invariants hold, are very effective in catching incomplete or inconsistent specification of the feature. Like a good debugger which helps programmers to fix a program rapidly before going to extensive testing, animation helped us to "fix" the specification before going through the formal proofs.

7 Related work

7.1 Event-B vs. RAISE

This research is closely related with Dines Bjørner's work [8–10]. In his work, he uses RAISE Specification Language (RSL) [1] for the description of domains, and concentrates on the formalization of as many domain facts as possible. Our objective is slightly different. Although we aim towards the enrichment of the transportation domain model, our concerns are also to check the capability of Event-B as a domain engineering tool and to point out and address (where possible) the issues with which we are confronted during this exercise. In the following paragraphs, we present a brief comparison of Event-B with Rigorous Approach to Industrial Software Engineering (RAISE) [21]:

Just as original B evolved into Event-B, RAISE is considered as an extended version of Vienna Development Method (VDM) [24]. It is based on enhanced features of several formal techniques, such as model oriented features of VDM, algebraic features of OBJ [18], concurrency features of CSP, modularity features of Meta Language (ML) [39], real time, etc.

The algebra-theoretic nature of most of its constructs is the basis of its structuring mechanisms. Its states are specified via types and predicates, like other formal methods, but a change in state can be specified in several ways, such as imperative, axiomatic, algebraic notations, etc. Event-B, on the other hand, is based on events which are controlled via guards. State transitions are defined via generalized substitutions. These guards and substitutions are similar to the concept of pre and post conditions.

Event-B enjoys a much more liberal refinement mechanism compared to RAISE. In RAISE, a refinement must have a signature that includes the signature of the abstract model. It is a tight 1-1 relationship. Event-B, on the other hand, relaxes this strict 1-1 relationship such that its syntax allows abstraction to be refined in more than one way. An abstract event can be refined by several events within the same refining machine. See section 3.1 for a detailed discussion on the refinement mechanism of Event-B.

RAISE achieves the notion of correctness of refinement through a standard principle of refinement consistency, i.e., at any time if an abstract operation is available, any refinement of it must also be available (enabledness-preservation). Event-B adds the idea of non-divergence to it. So, in Event-B a refinement is correct if it is enabledness-preserving as well as non-divergent.

In RAISE, it is theoretically possible to express and prove the liveness of each machine separately, but authors like [16, 20] have reported different experiences. Erasmy et al [16] failed to prove the liveness of the system because the justification editor lacked rules for the parallel combinator, i.e., (exp1 ∥ exp2) and its interaction with other combinators like sequential combinator, i.e., (exp1 ; exp2), rules for hiding schemes (schemes that hide some of the declarations inside), etc. According to [20], apart from applicative style, the concurrent style in RSL cannot be used to specify pure progress properties, e.g., fairness, which is one of the liveness properties. The poor ability of Event-B to describe temporal properties is discussed in detail in section 5.6.

The tools supporting both RSL language and the RAISE method have been commercially available since 1991. These tools revolve around the activities of writing specifications, type checking, performing justifications, translations of specifications into imperative languages like C++, Ada, etc., and documentation. Plugins for translations into Standard ML (SML) [36] and Prototype Verification System (PVS) [37], and generation of RSL from UML class diagrams are also available. Although, RAISE is sometimes criticized for its incomplete set of rules for the justification editor, such as absence of rules for the parallel combinator, interaction of channel hiding and the parallel combinator, etc., yet overall its toolset is easy to use, uniform and relatively fast.

Extensive tool support for Event-B is one of its powerful aspects. Event-B is supported by the platform Rodin which helps specification writing and proving. The Atelier-B [14] provers provide additional automated proof facilities to the existing Rodin provers. Animators like ProB, AnimB and Brama make possible the execution of specifications for their validation. The UML-B plugin [44] allows users to translate UML models into Event-B specifications for verification and validation. The B2Latex plugin allows the printing of Event-B

specifications into latex for documentation purposes. We can also run B models into the Rodin platform with the help of the B2Rodin plugin. There are also plugins for model decomposition, recomposition and code generation.

7.2 Event-B and goal models

A formal domain model should prove to be useful at the time when the system requirements are being specified. It can be used either as an inspirational source for the specification or as a testbed to check the compatibility of the requirements with the domain. However, current best practices would recommend the use of a goal oriented requirement engineering methodology to elicit requirements. Knowledge Acquisition in autOmated Specification (KAOS) [27] is a good candidate for this activity. Apart from its ability to state goals, their decompositions and their relationships, KAOS allows requirements engineers to express temporal properties by real-time Linear Temporal Logic (LTL) formulae. This complements the weakness of Event-B is this area.

The relationship between KAOS and Event-B is an active research field. In [7,4], a syntactic extension and patterns to model the notion of *obligation* introduced by the temporal model into Event-B are proposed. In [35,34], a simpler approach is explored. Both approaches rely on systematic transformation rules to derive an Event-B model from the KAOS model. They are consistent with the idea of a gradual introduction of formalism during the specification process. They provide us with a tool to study the relationship between Event-B formal domain models and requirements which are not yet fully clarified.

7.3 Event-B in the transportation domain

Previously, Event-B has been employed for the development of transportation systems, see for instance [38,11,17]. But most of the time the role of this language was limited to system modeling of a particular problem. Our work is different in the sense that we are modeling the domain, where such systems are assumed to operate. The specifications of these aforementioned railway systems do contribute towards the completion of the land transport domain model, but as a part of the whole. Our model is more general and could be used for different kinds of transportation systems, such as road, railways, conveyors, etc.

7.4 Event-B in the community

There has been some self-reflection within the Event-B community regarding some of the issues raised in this paper. For instance, the elegant expression of explicit timing properties or LTL expressions in Event-B is known to be

a challenging task. The expression of these properties, a key element for utilization of formal methods in the automotive sector, is currently non-standard in Event-B. Therefore, the correctness of the specifications which incorporate such expressions cannot be proved in Rodin alone.

Joochim et al [25], like Cansell et al [13], propose a timing pattern which uses global time and also interacts with a number of active times. This pattern formalizes the Timing Diagram of UML rather than considering timing properties in general. In addition, its usage is recommended at abstract stages rather than in later refinements.

In another work [6], the authors propose an extension of Event-B to incorporate three LTL operators: *next*, *eventually*, and *bounded eventually.* In this work, standard Event-B structures, WHEN, THEN and END are modified to represent these three LTL operators. Such models deviate from the standard Event-B notations and their verification and validation become a major challenge.

8 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 in requirements engineering.

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 CSP||Event-B [41] or event refinement diagrams [12] can be made practical is still an open issue. 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 inspired by our needs with respect to animation. Application of the transformational heuristics 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 in the form of a plugin for Rodin. The boring parts of the transformation do not contain overly complex text manipulation.

We would also appreciate to see tools evolving in the direction of richer visualization 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.

As we argued, knowledge of the particular domain before prescription of requirements is a valuable asset. We have hinted at ideas, such as deriving invariants of a system from the properties expressed in the domain model. We now want to test this by studying the practical relationship of our domain model with a separate specification, written also in Event-B, of a platooning system [28]. In particular, we would like to study how we can immerse the specification of a particular system into the domain model.

Acknowledgements

This work has been partially supported by ANR under project ANR-06-SETI-017 TACOS (`http://tacos.loria.fr`) and by Pôle de Compétitivité Alsace/Franche-Comté under CRISTAL project (`http://www.projet-cristal.org`).

References

1. Specification Case Studies in RAISE. Springer (2002)
2. Abrial, J.R.: The B Book. Cambridge University Press (1996)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
4. Aziz, B., Arenas, A.E., Bicarregui, J., Ponsard, C., Massonet, P.: From Goal-oriented Requirements to Event-B Specifications. In: 1st Nasa Formal Method Symposium (NFM'09), California, USA (2009)
5. Baille, G., Garnier, P., Mathieu, H., Roger, P.G.: Le Cycab de L'INRIA Rhônes-alpes. Technical Report RT-0229, INRIA Rhônes-alpes, Grenoble, France (1999)
6. Bicarregui, J., Arenas, A., Aziz, B., Massonet, P., Ponsard, C.: Towards Modeling Obligations in Event-B. In: 1st International Conference on Abstract State Machines (ASM), B and Z (ABZ'08), London, UK (2008)
7. Bisztray, D., Heckel, R., Ehrig, H.: Verification of Architectural Refactorings by Rule Extraction. In: 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), Budapest, Hungary (2008)
8. Bjørner, D.: Software Engineering 3: Domains, Requirements, and Software Design (Texts in Theoretical Computer Science, an EATCS Series). Springer (2006)
9. Bjørner, D.: Development of Transportation Systems. In: 2nd ISOLA Workshop on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'07) Poitiers, France (2007)
10. Bjørner, D.: Domain Engineering: Technology Management, Research and Engineering. JAIST (2009)
11. Butler, M.: A System-based Approach to the Formal Development of Embedded Controllers for a Railway. Design Automation for Embedded Systems, 6, 355–366 (2002)
12. Butler, M.: Decomposition Structures for Event-B. In: 7th International Conference on Integrated Formal Methods (IFM '09), Düsseldorf, Germany (2009)
13. Cansell, D., Mery, D., Rehm, J.: Time Constraint Patterns for Event B Development. In: 7th International Conference of B Users (B'07), Besançon, France (2006)
14. Clearsy: User Manual of Atelier B, version 4.0 (2009)
15. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Publishing Co. (2000)
16. Erasmy, F., Sekerinski, E.: RAISE: A Rigorous Approach using Stepwise Refinement. In: Formal Development of Reactive Systems, 891, 277–293, Springer (1995)

17. Essamé, D.: Handling Safety Critical Requirements in System Engineering Using the B Formal Method. In: 23rd International Conference on Computer Safety, Reliability, and Security (SafeComp'04), Postdam, Germany (2004)
18. Futatsugi, K., Goguen, J.A., Jouannaud, J.P., Meseguer, J.: Principles of OBJ2. In: 12th Symposium on Principles of Programming Languages (POPL'85), Louisiana, USA (1985)
19. 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)
20. Gørtz, J.: Specifying Safety and Progress Properties with RSL. In: 2nd International Symposium of Formal Methods Europe (FME'94), Barcelona, Spain (1994)
21. The RAISE Language Group: The RAISE Specification Language. Prentice Hall, Inc. (1993)
22. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Inc. (1985)
23. Jackson, M.A.: System Development, Prentice Hall, Inc., (1983)
24. Jones, C.B.: Systematic Software Development Using VDM (2nd edition). Prentice Hall, Inc. (1990)
25. Joochim, T., Snook, C., Poppleton, M., Gravell, A.: Timing Diagrams Requirements Modeling Using Event-B Formal Methods. In: IASTED International Conference on Software Engineering (SE'10). Innsbruck, Austria (2010)
26. Lamport, L.: Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering, 3(2), 125–143 (1977)
27. Lamsweerde, A.V.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009)
28. Lanoix, A.: Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In: 2nd International Symposium on Theoretical Aspects of Software Engineering (TASE'08), Nanjing, China (2008)
29. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: 12th International Symposium on Formal Methods (FM'03), Pisa, Italy (2003)
30. Mashkoor, A., Jacquot, J.P.: Incorporating Animation in Stepwise Development of Formal Specification. Research Report INRIA-00392996, LORIA, Nancy, France (2009)
31. Mashkoor, A., Jacquot, J.P.: Transformational Heuristics for Animation - Towards Stepwise Validation of Specifications. Research Report HAL-00544261, LORIA, Nancy, France (2010)
32. 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)
33. Mashkoor, A., Jacquot, J.P., Souquières, J.: B Événementiel pour la Modèlisation du Domaine: Application au Transport. In: 9th Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09), Toulouse, France (2009)
34. Mashkoor, A., Matoussi, A.: Towards validation of requirements models. In: 2nd International Conference on Abstract State Machines (ASM), Alloy, B and Z (ABZ'10). Orford, Canada (2010)
35. Matoussi, A., Gervais, F., Laleau, R.: A First Attempt to Express KAOS Refinement Patterns with Event B. In: 1st International Conference on Abstract State Machines (ASM), B and Z (ABZ'08), London, UK (2008)
36. Milner, R., Tofte, M., Harper, R., Macqueen, D.: The Definition of Standard ML - Revised. The MIT Press (1997)
37. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A Prototype Verification System. In: 11th International Conference on Automated Deduction (CADE'92), New York, USA (1992)
38. Papatsaras, A., Stoddart, B.: Global and Communicating State Machine Models in Event Driven B: A Simple Railway Case Study. In: 2nd International Conference of B and Z Users (ZB'02), Grenoble, France (2002)
39. Paulson, L.C.: ML for the Working Programmer (2nd Edition). Cambridge University Press (1996)
40. Poppleton, M.: The Composition of Event-B Models. In: 1st International Conference on Abstract State Machines (ASM), B and Z (ABZ'08), London, UK (2008)

41. Schneider, S., Treharne, H., Wehrheim, H.: A CSP Approach to Control in Event-B. In: 8th International Conference on Integrated Formal Methods (IFM'10), Nancy, France (2010)
42. Servat, T.: BRAMA: A New Graphic Animation Tool for B Models. In: 7th International Conference of B Users (B'07), Besançon, France (2006)
43. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition Tool for Event-B. In: 1st Workshop on Tool Building in Formal Methods. Orford, Canada (2010)
44. Snook, C., Butler, M.: UML-B: Formal Modeling and Design Aided by UML. ACM Transactions on Software Engineering and Methodology, 15(1), 92–122 (2006)
45. Yadav, D., Butler, M.: Verification of Liveness Properties in Distributed Systems. In: 2nd International Conference on Contemporary Computing (IC3'09), Noida, India (2009)
46. Yang, F., Jacquot, J.P.: Scaling up with Event-B: A Case Study. In: 3rd NASA Formal Methods Symposium (NFM'11), California, USA (2011)
47. Zave, P., Jackson, M.: Four Dark Corners for Requirements Engineering. ACM Transactions on Software Engineering and Methodology, 6(1), 1–30 (1997)