Formal Domain Engineering: From Specification to Validation

Atif Mashkoor

Nancy-Université
Université Nancy 2

Université Nancy 2

DOCTORAL SCHOOL IAEM

Department of Doctoral Formation in Informatics

Formal Domain Engineering: From Specification to Validation

Ph.D. Thesis

presented and defended publicly on July 12, 2011

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Informatics

by

Atif Mashkoor

Prepared at LORIA in team Dedale

Jury

President :	Prof. Patrick Heymans	-	Université de Namur (Belgium)
Examiner :	Prof. Maritta Heisel	-	Universität Duisburg-Essen (Germany)
Reviewers :	Prof. Marc Frappier	-	Université de Sherbrooke (Canada)
	Prof. Marie-Laure Potet	-	Université Joseph Fourier (France)
Advisors:	Prof. Jeanine Souquières	-	Université Nancy 2 (France)
	Dr. Jean-Pierre Jacquot	-	Université Henri Poincaré (France)

Loria Laboratoire lorrain de recherche en informatique et ses applications

Laboratoire Lorrain de Recherche en Informatique et ses Applications

Abstract

The main theme of this research is to study and develop techniques for modeling of software-controlled safety-critical systems. The area we focus in this thesis is the **specification** of a domain, where such systems are supposed to operate, and its **validation**. The contribution of this thesis is twofold: First, we model the land transport domain, a good candidate for this study because of its safety-critical nature, in the formal framework of Event-B and propose some *guidelines* for it. Second, we present an approach, based on the technique of animation and *low-cost* transformations, for stepwise validation of formal specifications.

Keywords: Domain engineering, Requirements engineering, Formal methods, Software testing, Event-B, Brama

Le thème principal de cette recherche est d'étudier et développer des techniques pour la modélisation des systèmes où la sécurité est critique. Cette thèse est focalisé sur l'étape de la spécification du domaine où de tels systèmes vont fonctionner, et de sa validation. La contribution de cette thèse est double. D'abord, nous modélisons le domaine des transports terrestres, un bon candidat pour cette étude en raison de sa nature critique vis-à-vis de la sécurité, dans le cadre formel de B événementiel et proposent quelques directives pour cette activité. Ensuite, nous présentons une approche, basée sur les techniques de l'animation et des transformations, pour la validation par étapes des spécifications formelles.

Mots-clés: Ingénierie de domaine, Ingénierie des besoins, Méthodes formelles, Testing de logiciel, B événementiel, Brama

In the sweet memories of my loving parents
who sacrificed their present
for the future of their children.

Acknowledgments

All praise belongs to ALLAH, the almighty, on whom ultimately we depend for sustenance and guidance.

Foremost, I would like to express my sincere gratitude to my co-advisor Dr. Jean-Pierre Jacquot for the continuous support during my Ph.D. study and research. I appreciate his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me to shape my goals, both in my research and life. I would always remember him as the best advisor and the mentor for the lifetime. I would also like to thank Prof. Jeanine Souquières, my main advisor, for supporting me financially, administratively and technically.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Marc Frappier, Prof. Marie-Laure Potet, Prof. Patrick Heymans and Prof. Maritta Heisel, for their encouragement, insightful comments, and positive criticism. I would also like to extend my gratitude to Prof. Dominique Méry and Dr. John Fitzgerald for their advices and time.

I owe my deepest gratitude to my friends, Sarah, Ehtesham, Dawood, Bilal, Usman and Mumtaz for being there for me physically, spiritually and morally whenever I needed them.

Lastly, and most importantly, I wish to thank my family: My eldest brother Kashif, my sister Rabia, my three nieces, Aliza, Mouniza and Arisha, and notably my beloved brother Rashid. They have supported me unconditionally and unprecedentedly. They gave me the choices I wanted, the time I needed, the strength I required, the support I wished; they gave me everything I demanded. Thank you guys for all of your support!

Atif Mashkoor
August 8, 2011
Vandœuvre lès Nancy

Contents

Chapter 1

Prologue

Contents

1.1 Introduction

The principle of understanding the domain before the specification of requirements is crucial to software engineering. The idea of having enough details about the environment in which designed products are assumed to operate is well established in other engineering disciplines. Engineers belonging to disciplines other than software, such as aeronautics, electronics, and chemistry, know the domains of their respective fields. Though, in software engineering, a system is sometimes developed by people with an incomplete knowledge of its particular domain. Unsurprisingly, requirements of such systems may be flawed and their correctness is a crucial issue.

[Bjørner 2007] also attests this assumption and believes that formal expression of requirements and domain models is essential for software correctness. This practice ensures safety which is one the major concerns which cannot be overlooked while designing complex and critical systems. The point is that the use of formal methods from domain modeling to implementation can help us ensure the safety and correctness of software in contrast to traditional methods which may lead to erroneous systems.

However, despite decades of advocacy, success stories, and good proof systems, formal methods are still not popular in software development. One of the reasons may be that tools and techniques may still have some deficiencies which impede their penetration into the industry. This thesis aims at hinting and (where possible) providing solutions to some of those hindering factors for at least one formal method, Event-B [Abrial 2010].

This chapter is organized as follows: Section 1.2 presents the main motivation for this research. Section 1.3 discusses the main results which we have obtained during this work. The peer-reviewed publications of this research are mentioned in section 1.4. Finally, section 1.5 discusses the organization of the thesis.

1.2 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 the 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 thesis, 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.

1.3 Contributions

We work on two levels: specification and validation. Following is the list of contributions at each level:

1.3.1 Specification level

The results obtained at specification level are as follows:

Guidelines for domain engineering with Event-B

We critically analyze the capability of Event-B as a domain engineering tool. We identify its shortcomings and (where possible) suggest solutions for improvement. We primarily answer three fundamental questions pertaining to domain modeling: what to specify? how to refine? and how to verify? The proposed general-purpose (domain-engineering) guidelines can be leveraged across the board.

Formal domain model of land transport

We provide a formal, consistent, and effective domain model of land transportation, written in Event-B. The stepwise refinements of the model express the intrinsic facts and laws of the domain along with two properties: collision avoidance and timing. We use theorem proving technique in order to ensure the verifiability of the model. The prescribed core functionality of the domain can be utilized for the emersion of several transportation systems.

1.3.2 Validation level

The results obtained at validation level are as follows:

Transformational heuristics for animation

Animation is one of many effective techniques for specification validation but not all specifications are directly animatable; some need to be transformed. In order to do so, we propose some heuristics which are designed with a strong constraint: they must preserve the behavior of the specification. Based on *precise semantics*, these heuristics guarantee that "anything that is observed during the animation of the transformed specification would have been observed on the animation of the original specification." We test our results on two case studies: the land transport domain model and a platooning system.

VTA

We propose VTA (Verify-Transform-Animate), a framework for stepwise validation of formal specifications. This breaks the requirements validation process into small steps and integrates it into the stepwise development of specifications. Our framework, based on *cheap* transformations, reduces the overall cost and time of the validation process.

The original and fully-verified Event-B specification of the transportation domain written by us is available at the following web address: `http://dedale.loria.fr/?q=en/transport-domain`. The specification of the platooning system which is used as a case study in this thesis is available at: `http://dedale.loria.fr/?q=en/platooning-system`.

1.4 Publications

The obtained results of this research have been published in the following form:

Journal

- A. Mashkoor and J. P. Jacquot, Utilizing Event-B for Domain Engineering: A Critical Analysis, *In: Requirements Engineering (REJ),* Springer, 2011

Conferences

- A. Mashkoor and J. P. Jacquot, Domain Engineering with Event-B: Some Lessons We Learned, *In: 18th International Requirements Engineering Conference (RE'10),* Sydney, Australia, 2010

- A. Mashkoor and A. Matoussi, Towards Validation of Requirements Models, *In: 2nd International Conference on Abstract State Machines (ASM), Alloy, B and Z (ABZ'10),* Orford, Canada, 2010

Workshops

- A. Mashkoor, F. Yang and J. P. Jacquot, Validation of Formal Specification: The Case for Animation, *In: 3rd Workshop on Security and Reliability (SecDay'11),* Trier, Germany, 2011

- A. Mashkoor and J. P. Jacquot, Branimation, *In: 1st RODIN User and Developer Workshop (RUDW'09),* Southampton, United Kingdom, 2009

- A. Mashkoor, J. P. Jacquot and J. Souquières, Transformation Heuristics for Formal Requirements Validation by Animation, *In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'09),* York, United Kingdom, 2009

Miscellaneous

- A. Mashkoor, Formal Domain Modeling: From Specification to Validation, *In: Doctoral Symposium of 16th International Symposium on Formal Methods (DS-FM'09),* Eindhoven, The Netherlands, 2009

- A. Mashkoor, J. P. Jacquot and J. Souquières, B Événementiel pour la Modélisation du Domaine: Application au Transport, *In: 9th Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09)*, Toulouse, France, 2009

1.5 Structure of the thesis

This thesis is organized into five parts. Part I presents the introduction of the thesis. We present the main motivation to initiate this research along with the contributions in this direction.

Part II of the thesis is dedicated to the state-of-the-art. This work revolves around three fundamentals of software engineering: domain engineering, requirements engineering, and formal methods. In this part, we discuss what is the main idea behind them and how they can be performed. We explicitly define the process of domain specification, verification and validation, which form the basis of this thesis.

Part III of the thesis illustrates the specification level of this work. We first introduce the formal method Event-B and then use it to model the transportation domain. Later, we present our experience gained as guidelines for domain engineering in the formal framework on Event-B.

Part IV of the thesis is devoted to the process of validation. We define the technique of validation by animation along with its limitations. We, then, propose some heuristics with a formal semantics which provide a bridge for the animator Brama [Servat 2006] in order to help it animating specifications. We also show their effects on two different case studies.

The thesis is finally concluded in part V which presents our proposed framework for stepwise validation of specifications VTA along with a conclusion and some future works.

Part I

BACKGROUND

Chapter 2

Domain engineering, requirements engineering & formal methods

Contents

2.1 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: the major entities and their inter-relationships, the static and dynamic properties, the functions, the events, and the behaviors.

Requirements engineering is a methodology to specify all the known requirements related to the development of a particular system. There are numerous reasons to perform domain engineering prior to requirements 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 requirements conform to essential properties, and so on. Furthermore, domain engineering in a formal framework gives practitioners an effective grasp on notions, such as verifiability and validity of domain facts.

In this chapter, we present a short survey about three very important pillars of software engineering: domain engineering, requirements engineering and formal methods. We present their general philosophies along with their supporting methods.

The chapter is organized as follows: Section 2.2 presents what is a domain, how to engineer it, and what are the different methods of this engineering. Section 2.3 answers the questions related to requirements engineering, such as what are requirements, how to engineer them, and how to classify them. Section 2.4 presents a brief introduction to formal methods, their advantages, disadvantages, myths, and guidelines. The chapter is finally concluded in section 2.5.

2.2 Domain engineering

2.2.1 Domain

In literature, we find two different point of views towards the notion of domain. We present them in the following as definition 1 and 2. Please note that the definition 1 has been subsequently used throughout this thesis unless stated otherwise.

Definition 1: In the first point of view, a domain is a universe of discourse, an area of nature subject to the laws of physics, or an area of human activity subject to its interfaces with other domains and to nature [Bjørner 2009b]. In this context, a domain can be seen as the real-world [Simos 1997]. A domain therefore incorporates the processes of an environment irrespective of specific system requirements. A model of a domain behaves as the conceptual model and identifies the facts, terminologies, laws, behaviors and functionalities of the particular domain. This definition corresponds to the vertical domains as defined by Czarnecki [Czarnecki 2000]. Examples of such domains are transportation, financial service industry, banking, insurance, health care, etc.

Definition 2: In the second point of view, a domain can be seen as a set of systems and application families [Simos 1997]. The scope of the domain, in this fashion, is drawn according to the similarities between the applications falling into the domain. It models their commonalities which later contributes to product-line engineering. Czarnecki [Czarnecki 2000] calls them horizontal domains. Examples of such domains are word-processors, database systems, code libraries, workflows, etc.

2.2.2 Domain engineering

Domain engineering is an activity that concerns the documentation of facts related to one domain. It is a discipline to identify, model, construct, catalog, and disseminate a set of software artifacts that can be applied to existing and future software in a particular application domain [Pressman 2005]. The activity of domain engineering helps understand the environment where these systems are assumed to operate so that later on this understanding can be transformed into better requirements specification. In addition, it also makes possible to deliver new products in shorter time and at lower costs.

Domain engineering helps in several aspects, such as acquiring better insight of the environment where systems are supposed to work, better specification of system requirements, engineering of reusable software, i.e., libraries, frameworks or tools, knowledge management, i.e., continuous maintenance and update of knowledge in a domain, etc.

The process of domain engineering starts with the identification of the domain to be analyzed. This process of identification includes the activities, such as examination of existing applications, consultation with domain experts and different stakeholders. Based on these elements of information, a domain model is constructed. A domain model, which is the deliverable of the process of domain engineering, describes the key concepts and vocabulary pertaining to the domain being engineered. This document has numerous advantages: it describes and constrains the system boundary, it helps verify and validate that the problem domain is well understood, and it acts as a communication tool between various stakeholders of the system.

2.2.3 Domain engineering methods

Surveys, like [Wartik 1992], [Ferr'e 1999], [Harsu 2002], and [Ahmad 2004], introduce many techniques and methods for engineering domains. In this subsection, we present four such methods: Draco, Feature-Oriented Domain Analysis (FODA), Organization Domain Modeling (ODM), and Formal domain engineering. While the first three methods dominantly relate to the second definition of domain discussed earlier in this section, the last method is related to the first definition of domain and has been later employed in this thesis. Following is an abridged introduction to these domain engineering methods:

Draco

Draco [Neighbors 1980], considered as the first domain engineering method, was proposed by James Neighbors in 1980. The key idea of this approach is to organize the knowledge helpful to construct software into related domains.

The resulting domains of Draco consist of the following items: Parser to transform a system description to an internal form in a domain language, domain language which is implemented by a parser, pretty printer which is used to describe certain aspects of an application, display which describes how to display internal

forms as texts and graphics for implementation purposes, optimizations which describe the rules of exchange between statements in the domain language and may be procedural or source-to-source, components which specify the semantics of the domain through implementation of the domain language constructs in the domain language of other domains and provide refinements for these constructs, generators which describe transformational programs for the statements in a domain language for their conversion into another statement in the same domain, analyzers which describe programs that gather information about the internal form for use by the other domain parts, and strategies and tactics which describe refinement plans based on available refinements and domain interconnections [Neighbors 1989].

Draco domains, which encapsulate the knowledge for solving certain class of problems, can be classified into application domains, modeling domains, and execution domains. Application domains gather knowledge for building specific applications, e.g., transportation, finance, health, etc. Modeling domains encapsulate knowledge which is required for producing parts of systems. These reusable artifacts are further sub-divided into application-support domains, e.g., navigation, accounting, numerical control, etc., and computing technology domains, e.g., multitasking, transactions, communications, graphics, databases, user interfaces, etc. Execution domains are the refinements of application domains into more specific areas. Concrete target languages, such as C, C++, or Java are part of these execution domains and are used to realize application domains into targeted systems [Neighbors 1989].

FODA

FODA [Kang 1990] is a domain engineering method which is centered around feature modeling. The actual concept is to represent the standard features and their interrelationship of a particular domain in the systems of the domain. Features are the desirable functionalities of the applications and span over both common (mandatory) and variable (optional) parts of the system families. The domain model produced by FODA also includes the differences between related applications along with their standard ingredients.

There are two major stages for FODA: context analysis, and domain modeling. The phase of context analysis is used to draw the boundaries of the domain and its relationship with the external environment. The outcome of the context analysis phase is a context model which states these phenomena with the help of structure and data-flow diagrams.

The stage of domain modeling is further divided into feature analysis, information analysis, and operational analysis. Feature analysis helps finding the features, i.e., modeling end users' perception about the core capabilities in the domain along with its optional and alternative capabilities. This later helps in the identification of families of features. The main purpose of information analysis is to capture domain knowledge in the form of domain entities and the relationships between them. The result of information analysis is an information model. Operational analysis produces the operational model which shows the workflow of the application by

documenting the relationships between the objects in the information model and the features in the feature model. Another important artifact of this phase is the domain dictionary which defines the terminology used in the domain.

ODM

ODM [Simos 1995], is generally considered as a method for domain engineering of legacy systems, but can also be applied to the modeling of new systems. It is a customizable and configurable process which can be integrated with other software engineering stages. The main phases of this method are planning, modeling a domain, and engineering asset-base.

The phase of planning is further divided into three sub-phases which are setting objectives, i.e., determining who the stakeholders are and what their objectives are; domain scoping, i.e., characterizing and selecting the focus of a domain based on stakeholders' objectives; and domain definition, i.e., setting domain boundaries and identifying the main features of the domain.

Domain modeling includes acquiring domain information, i.e., planning, acquiring and integrating the data; describing domain, i.e., developing a lexicon, and modeling concepts and features; and refining domain, i.e., integrating descriptive models, feature selection rationale and interpretation of domain models.

The phase of engineering asset-base includes scoping the asset-base, i.e., correlation of features to customers, their priorities and their selection for implementation; architecting the asset-base, i.e., determining internal and external architecture constraints, and defining an asset-base architecture based on these constraints; and implementing the asset-base, i.e., planning asset base implementation, and implementation of assets and supporting infrastructure [Simos 1996].

Formal domain engineering

The use of formal specification, which is based on a formal language, axiomatic constraints and inference rules, is famous in software engineering since many years. For the specification of domain models, it was probably first used by [Srinivas 1991]. The principle idea of this approach is to document domain knowledge with the help of formal notations, such as mathematical models and algebraic specifications.

The idea of stepwise development, verification and validation of models is the main highlight which differentiates formal domain engineering from others. With the help of mathematical constructs, we can precisely document the facts of a domain. The mathematical model also helps us in assessing its correctness by various means, such as proving the consistency for verification or executing the model for validation.

The formal approach of modeling domains is similar to Draco in nature apart from the difference that this approach is based on mathematical theories, such as the algebraic specification theory, or the category theory. The idea of refinements and rigorous development towards implementation are common features of both approaches [Czarnecki 2000].

Formal specifications can also be related one to another using specification morphisms. Specification morphisms define translations between specification languages which preserve the theorems that can be derived from the axioms using the inference rules. Therefore, in the formal approach, a domain model can be represented in a number of formal languages including translations between them. The formal models can later also be directly translated into codes in several programming languages.

In recent times, Dines Bjørner's work is most notable in the field of formal domain engineering. He has authored many articles on domain engineering using this approach, e.g., [Bjørner 2010, Bjørner 2009a, Bjørner 2009b, Bjørner 2008b, Bjørner 2008a, Bjørner 2007, Bjørner 2006, Bjørner 2005, Bjørner 2004]. He uses RAISE Specification Language (RSL) [rai 2002] for the description of domains and concentrates towards the formalization of as much domain facts as possible. His main areas of interest are transportation, oil pipelines, container line industry, etc.

Our research differs from his work on two main fronts. First, we head towards the enrichment of domain models while paying as much attention to verification and validation as to specification. Second, 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. The choice of the formal method can be another difference. We compare both of them, Event-B and RAISE [Group 1993b], in section 4.7.

2.3 Requirements engineering

2.3.1 Requirement

The IEEE standard glossary of software engineering terminology [IEEE 1990] defines requirement as "a condition or capability needed by a user to solve a problem or achieve an objective." Alternatively, it is defined as "a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents."

Requirements express the needs of stakeholders (system users, organization, community, government bodies and industry standards). They identify the attributes, capabilities, characteristics, or quality of an operational system so that it can provide some utility to stakeholders.

Ideally, requirements are independent of design, i.e., they shall only demonstrate that "what" the system should do, rather than "how" it should be done. However, in reality, this is not always the case. Drawing the line between "what" and "how" has always been considered as a challenge of requirements engineering. One of several reasons for this is that the meanings of "what" and "how" vary from person to person [Davis 1990].

2.3.2 Requirements engineering

In a broader sense, the overall goal of the requirements engineering process is to create and maintain system requirements documents. It is a process to understand what are the customers' requirements, analyzing these requirements, assessing their feasibility, negotiating a reasonable solution in case of any conflict, specifying these requirements unambiguously, validating them, and managing them until they are transformed into an operational system [Thayer 1997].

According to [Zave 1997b], "requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints of software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families."

Laplante [Laplante 2009] defines the phase of requirements engineering as a sub-discipline of system and software engineering which is related to determining the goals, functions, and constraints of hardware and software systems.

According to [Sommerville 2006], the requirements refer to the services provided by the system and its operational constraints and requirements engineering is a process to discover, analyze, document and assure these services and constraints.

According to [Loucopoulos 1995], requirements engineering is a systematic process of developing requirements through an iterative co-operative process of analyzing the problem, documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained.

2.3.3 Classification of requirements

Sommerville [Sommerville 2006] classifies software requirements into two categories: user requirements and system requirements. User requirements, which are the high level abstract requirements, describe, either in plain language or graphically, the services and constraints of the system. Whereas, system requirements, which are the detailed description of the system, precisely describe the functions, services and operational constraints of the system in details, and acts as an agreement between users and developers. Following is the brief classification of the different software system requirements:

Functional requirements

A functional requirement is a software requirement that specifies the function of a system or of on of its component. The primary objective of functional requirements is to define the behavior of the system, i.e., the fundamental processes or transformations that software and hardware components of the system perform on input to produce output. The plan for implementing functional requirements is detailed in the system design. Example of functional requirements of a transportation system may be: vehicles move, they cross hubs, they traverse path, etc.

Non-functional requirements

A non-functional requirement is a software requirement that specifies the criterion to judge the behavior of a system, i.e., it describes how the software should perform rather than what it performs. The plan for implementing non-functional requirements is detailed in the system architecture. Non-functional requirements become part of requirements documents under several names, such as quality requirements, quality attributes, quality goals, quality of service requirements, quality constraints, and non-behavioral requirements. Examples of non-functional requirements are collision avoidance, fuel efficiency, timing, etc. The figure 2.1 shows a detailed classification of non functional requirements.

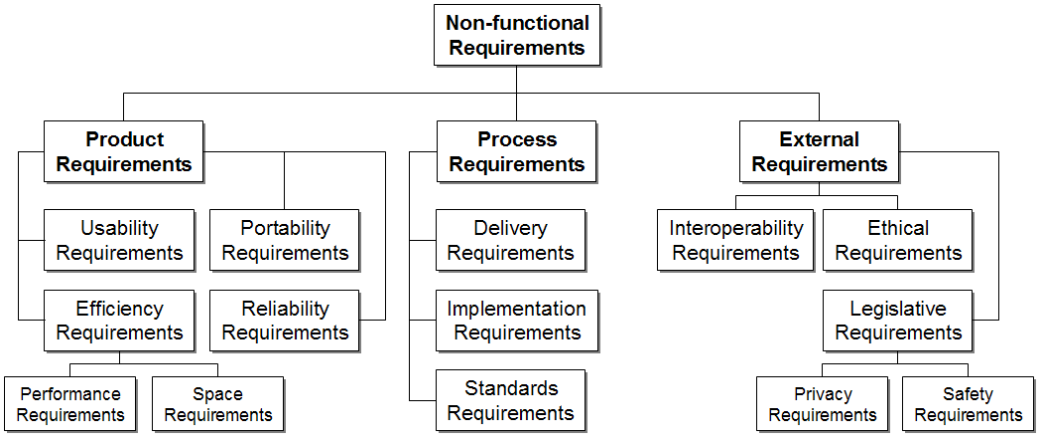

Figure 2.1: Types of non-functional requirements [Sommerville 2006]

Functional versus non-functional requirements

The notion of non-functional requirement may become quite fuzzy sometimes. The functional requirement of one system may be the non-functional requirement of another. For instance, a system requirement states that a vehicle moves from one place to another. An adjunct non-functional requirement may be associated to this which states that this movement should be collision free. However, in another system the functional requirement may state that a vehicle moves from one place to another without having any collision with another vehicle. In this case, the property of collision avoidance becomes the functional requirement of the system. In another scenario, a user requirement of security may be a non-functional requirement. However, when the system is developed this requirement may give birth to a requirement related to user authentication which may be a functional requirement. The bottom line is that the line which separates requirements from functional to non-functional is very thin and their classification sometimes becomes not as clear-cut as the definitions suggest.

2.3.4 Phases of requirements engineering

Different authors include different sub-processes as part of requirements engineering but the common primary activities during different requirements engineering processes are elicitation, analysis and documentation, negotiation, verification and validation, change management, and requirements tracing. Following is a brief introduction to each of them:

Requirements elicitation

Requirements elicitation or requirements gathering is the process of obtaining the requirements of a system from different stakeholders. Apart from stakeholders, which are the main source of requirements, requirements may also come from engineering needs, safety constraints, legislation, cultures, lessons learnt, etc. An additional process of context analysis [Ross 1977], which highlights the main motives behind the construction of the system, may also give new insights about the system.

The common activities during the requirements elicitation phase, according to [Zowghi 2005], are understanding the application domain, identifying the source of requirements, analyzing the stakeholders, selecting the techniques, approaches, and tools to use, and eliciting the requirements from stakeholders and other sources. The gathered requirements may be collected in several forms, such as pictures, sketches, scenarios, uses cases, goals, natural language statements, and formal models. The common methods used to gather requirements from stakeholders are interviews, questionnaires, observations, workshops, brainstorming, use cases, prototyping, ethnography, etc.

Besides the problem of completeness, i.e., the acquisition of all necessary requirements, according to [Christel 1992], there are also few other notable problems associated with the phase of requirements elicitation, such as scoping, i.e., ill-defined boundary of the system or un-necessary technical details from customers, understanding, i.e., customers's lack of assurance with respect to his requirements, poor understanding towards capabilities and limitations, insufficient understanding of the problem domain, trouble in communicating needs to the system analysts, etc, and volatility, i.e., change in requirements over the period of time.

Requirements analysis & negotiation

Requirements analysis is a process of categorization and organization of requirements into related subsets, exploration of relationships among requirements, examination of requirements for consistency, omissions and ambiguity, and ranking requirements based on the needs of customers [Pressman 2005]. The structured analysis of the requirements can be achieved by analysis techniques, such as requirements animation, automated reasoning, knowledge-based critical analysis, consistency checking, analogical and case-based reasoning.

It is common during the requirements analysis phase that different customers propose conflicting requirements, which from their point of view are essential for the

system. The analyst must reconcile these conflicts through a process of negotiation. During the process of requirements negotiation, stakeholders are asked to rank their requirements, risks associated with each requirement are identified and analyzed, rough cost estimates are made for each conflict, and then, depending upon the requirement priorities, requirements are eliminated, combined or modified.

Requirements specification[1]

Once requirements have been elicited, analyzed and negotiated, they need to be documented. The phase of requirements specification serves this purpose. There are several ways to specify software requirements, either through a document written in natural language, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. The three most common classes of languages for requirement specifications are informal, semi-formal and formal languages. Informal languages, such as natural language, pictures, etc., are expressive and can be understood by all stakeholders including non-technical customers but are ambiguous. Semi-formal languages, such as Entity Relationship Diagrams (ERD) [Chen 1976], Data Flow Diagrams (DFD) [Yourdon 1979], State Transition Diagrams (STD) [Booth 1967], etc., are easy to understand and provide a good overview of the system. Formal languages, such as B [Abrial 1996], Z [Spivey 1989], Vienna Development Method (VDM) [Jones 1986], specify requirements very precisely but are costly and hard to read for non-technical stakeholders.

Any language of specification can be chosen depending upon the project's needs however it should be noted that expressiveness and precision should be the key factors while making this choice. Each of these languages is popular for the development of particular systems: for large systems, a written document in natural language supported by few graphics may be the best approach, for relatively smaller systems, scenario based approaches may provide better results, and for safety-critical systems, use of formal methods is considered as industry standard. IEEE [IEEE 1998b] suggests a standard template for writing software specifications in a consistent and understandable manner. However, these are just guidelines and sometimes remaining flexible during system specification is advisable.

Software requirements specification is a part of the overall system specification document. The former is the foundation for software and the latter serves as the base for the whole system including software, hardware, database, etc. It constrains the computer-based system and define its operations. System specification also describes the information (data and control) that is input to and output from the system. Both requirements engineers and system analysts participate in the development of this document.

[1] A detailed discussion on the subject of specification can be found in chapter 3.

Requirements verification & validation[2]

Correctness, as defined by [McCall 1977], is the extent to which a program satisfies its specifications and fulfills the customer's mission objectives. Verification and Validation (V&V) are the means to establish that a specification is correct with respect to customer's requirements.

According to the IEEE Standard Glossary of Software Engineering Terminology [IEEE 1990], verification is defined as "the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."

Validation, as defined by IEEE Standard Glossary of Software Engineering Terminology [IEEE 1990], is "the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements."

V&V is an ongoing process throughout the software life cycle, however its incorporation earlier during the development has always been a good idea. V&V activities examine the specification to ensure that all system requirements have been stated unambiguously, consistently, completely, and correctly. A comprehensive V&V process includes both static and dynamic techniques. Static techniques correspond to analysis of requirements documents, design documents and the program source code. Dynamic techniques, commonly known as testing, corresponds to the execution of the artifacts, such as prototyping, animation, simulation, etc. Its task is to show that requirements model in some easily comprehensible form to customers. IEEE [IEEE 2004] proposes a comprehensive verification and validation plan for the software development life-cycle.

Requirements management

Requirements engineering is an ongoing process. But then a question arises, if this process is ongoing, then when can we start building the system? The problem is hard to define "fully," therefore requirements of most of the systems are also incomplete. The understanding of the stakeholders constantly change as we proceed with software development. Even when the software is installed, new requirements may emerge.

The answer to all aforementioned problems lies in requirements management. Requirements management is a set of activities during which we identify, control, and track any possible changes to requirements at any time during the life of the project. A track list must be kept for new requirements and also between dependent requirements because if one requirement changes, it may have effect on several other related requirements. During elicitation, requirements can be classified as enduring requirements, i.e., relatively stable requirements or volatile requirements, i.e., likely to change requirements. With this classification it becomes relatively easy to predict which requirement is prone to change. Since change management is quite expensive,

[2] A detailed discussion on the subject of V&V can be found in chapter 3.

it must be guided by a standard and well-defined process, and it should be planned at the beginning of the requirements elicitation phase. Use of traceability policy to define and maintain the relationships among requirements is often advised along with Computer Aided Software Engineering (CASE) tool support for requirements management.

2.3.5 Requirements engineering methods

Viewpoint-oriented approaches

The development of a complex system description or model involves many agents, i.e., participants, actors, etc. These agents have different views of the system under consideration. These views are, in general, partial and incomplete pictures of the system because these agents have different responsibilities and roles. These responsibilities and roles may be organizationally defined, following some defined structure, or may reflect different modeling or descriptive capabilities. The combination of the agent with his views of the system is termed a viewpoint. The study of viewpoints embraces the relations between views, between views and agents, and between agents [Finkelstein 1996]. The notable viewpoint-oriented requirements engineering methods are COntrolled Requirements Expression (CORE)) [Mullery 1979], Structured Analysis and Design Technique (SADT) [Marca 1987], Process and REquirements VIEWpoints (PREview) [Sommerville 1997], etc.

Goal-oriented approaches

A goal is an objective that the system under construction should achieve [Lamsweerde 2001]. Goals represent intended properties and can cover both functional concerns that the new system should provide, and non-functional concerns related to its quality of service, such as security, safety, etc. Contrary to requirements, there are multiple agents involved in the achievement of a goal. Goals can be used for multiple tasks, for instance, a goal can provide rationale for some requirements, it can be used to assess the completeness and relevance of a requirement, or it can be used for requirements identification. For instance, a requirement is justifiable and relevant if it leads to satisfaction of a goal, and requirements are complete if all goals are satisfied with the set of defined requirements. Some notable goal-oriented requirements engineering methods are the NFR Framework [Mylopoulos 1992] or Knowledge Acquisition in autOmated Specifications (KAOS) [Dardenne 1993], i∗ [Yu 1997].

Scenario-based approaches

According to [Sutcliffe 2003], scenarios are descriptions of the real-world and stories narrated by stakeholders used for modeling and specifying system requirements. From the point of view of requirements, scenarios are examples of real-world experiences, expressed in natural language, pictures, etc. From the point of view of

specification, scenarios are like models, such as use cases, threads through use cases, and other descriptions of sequence of events.

By another definition [Jarke 1998], a scenario is a description of a possible set of events that might happen. The main goal for constructing scenarios is to stimulate thinking about possible occurrences, assumptions relating these occurrences, possible opportunities, possible risks, and possible courses of action.

Some notable scenario-based requirements engineering approaches are Unified Modeling Language (UML) [Booch 1998], Class-Responsibility-Collaboration (CRC) card approach [Beck 1989], Scenario Based Requirements Elicitation (SBRE) [Holbrook 1990], and Negative Scenarios and Misuse Cases Method [Alexander 2003] [Alexander 2004].

Problem frames

The Problem Frames (PF) approach is based on the decomposition of complex problems into structured sets of simpler interacting subproblems with understandable interfaces in order to deal with their complexity. The solution to the original problem then lies in the combined descriptions and solutions of these sub-problems. The rationale behind the key point of the PF approach, i.e., decomposition of complex problems into familiar sub-problems, is that if a similar problem has been solved in the past, it is beneficial to use it in solving the current problem. Thus, the approach sets out to identify the common simple problems which can be used as patterns onto which the complex problems should be decomposed. These classes of common problems are identified from the body of work on problem analysis for software development, in the same way as design patterns are identified from software design work. The PF mostly target the specification of functional requirements [Jackson 2001].

2.4 Formal methods

The term formal methods refers to the particular kind of mathematical techniques which are employed for the specification, verification and validation of software requirements. Formal methods are based on a variety of theoretical computer science fundamentals, such as formal languages, automata theory, program semantics, logic calculi, algebraic data types, etc.

The use of formal methods enables software engineers to produce software artifacts (domain models, requirements specifications) which can be considered consistent and unambiguous as compared to those produced using other conventional software engineering methods. Once specified, these artifacts then undergo a series of verification and validation steps, in order to ensure their consistency, correctness, and completeness. The use of refinement further strengthens the prescription power of formal methods. This natural style of specification development, in which the understanding of the system by the specifier increases with the passage of time, also helps correct specification of the models. The power of automated tools facilitates developers on issues like learning curves, time to market, automated code generation, etc.

Formal methods are generally considered as expensive. One of the cost-effective way to use them may be to specify a system formally and then to develop programs from this informally. However, in high-integrity systems where safety or security is of utmost importance, a software can be developed in in more formal way. For example, proofs of properties or refinement from the specification to a program may be undertaken. Theorem provers or model checkers can then be used to ensure program verification.

Like for programming languages, the semantics of formal methods can also be classified into denotational, operational and axiomatic. In denotational semantics, the meaning of a system is expressed in the mathematical theory of domains. For example, a system might be represented by partial functions, by actor event diagram scenarios, or by games between the environment and the system. In operational semantics, the meaning of a system is expressed as a sequences of computational steps. These sequences are then the meaning of the program. For example, in the context of functional programs, the final step in a terminating sequence returns the value of the program.

In axiomatic semantics, which we use to model our domain, the meaning of a system is expressed in terms of preconditions and postconditions which are true before and after the system performs a task, respectively. Such formal specification typically consists of state variables, constraints on these variables known as invariants, and functions/operations/events. The variables of the system contains the latest state information which can only be altered by defined operations. The invariant specify the conditions for variables which must remain valid throughout the execution of a function.

2.4.1 Advantages of formal methods

Following are the main advantages to employ formal methods in software development:

- *Formal methods help better specification of requirements.* Formal specifications are consistent and unambiguous, thus making the overall requirements engineering process better than conventional methods.

- *Formal methods are compatible with several phases of software engineering.* From domain models to testing, the integration of formal methods make software more dependable.

- *Formal methods are supported by tools.* The supporting tools operate on formal specifications and assist in automated verification, testing and code generation.

- *Formal methods help better construction of software.* They also increase its efficiency. Better software means its requirements are consistent, complete and chances of bugs are less, thus increasing software's productivity.

- *Formal methods help execute the specifications.* It means customers can be involved into the development right from the start and requirements errors can be rectified right on the spot.

2.4.2 Disadvantages of formal methods

Besides their several advantages, the use of formal methods are still not popular in industry. Some of the possible reasons for their unpopularity are:

- *The use of formal methods is expensive.* It requires lots of time, skilled labor and changes in development attitude.
- *Formal methods are difficult.* Not all stakeholders feel comfortable with the complexity of formal methods, not even software engineers, let alone customers.
- *Not all formal methods enjoy good tool support.* Even if the tools are there, they are still crude, may contain bugs and are difficult to use.
- *Formal methods lack success stories.* Apart form safety-critical systems, there are not much success stories in other domains of software which certainly impede the penetration of formal approach in industry.
- *The expressiveness of formal methods is questionable.* It is difficult to specify the complete behavior of the system with formal methods; non-functional properties of performance and reliability are vivid examples of this.
- *Formal methods are notorious.* Different people have different prejudices towards formal methods. Most of these preconceived opinions are myths which are covered in the following section.

2.4.3 Myths of formal methods

In two separate papers, Hall [Hall 1990] and Bowen [Bowen 1995a] present several myths associated to the use of formal methods. Some of these myths are concisely discussed as follows:

- *Formal methods can guarantee perfect software.* This assumption is wrong as formal methods only model real-world phenomena and system requirements. The model can still contain many misunderstandings, errors and omissions. However, if applied carefully and in proper fashion, formal methods can significantly improve the process of specification, verification and testing.
- *Formal methods are all about program proving.* This assumption is also wrong as the role of formal methods is evident in domain modeling, requirements specification, testing, etc. Yes, verification is one of the powerful facet of formal methods, but it's not the only one.

- *Formal methods are unacceptable to users.* This myth is also not correct as visualization techniques, such as animation can be used to demonstrate the requirements to users to get their feedback.

- *Formal methods are not used on real, large-scale software.* The use of formal methods is actually recommended for large-scale software because of their time-scale and high costs. The development of some software, like mission and safety-critical software, nowadays, is not complete without the use of formal methods.

- *Formal methods increase the cost of development.* It depends! Using formal methods for design of an inventory control system may not be the best idea where time to market is short and requirements are relatively straightforward. However, in safety-critical systems, formal methods can help managing the costs by investing less on flawed requirements, design change and bug fixing.

- *Formal methods are not supported by tools.* Most of the formal methods, nowadays, are supported by tools. However, this is another debate whether these tools are suitable for industrial usage or not. In fact, tools like Safety Critical Application Development Environment (SCADE) [3], Isabelle [Paulson 2008], etc. are quite successful in industry.

- *Formal Methods only apply to software.* In fact, the use of verification techniques is very popular in hardware industry too, such as the use of Isabelle at Hewlett-Packard in the design of the HP 9000 line of server's runway bus [Camilleri 1997], or the use of HOL at Intel's lines of processors [Harrison 2003].

2.4.4 Guidelines for formal methods

In a famous paper "Ten commandments of formal methods," Bowen et al. [Bowen 1995b] present some general guidelines, which they reassert after a decade in the paper [Bowen 2006], to be employed while using formal methods. A brief summary of their papers is discussed here:

- *Choose appropriate notation.* Since a modeling notation plays a very important role in the formal specification of requirements, therefore it must be chosen carefully while keeping in mind its advantages, as well as its limitations. Some desirable language features could be vocabulary, tool support, relevance with the system under consideration, etc.

- *Do not over formalize.* Not every component of the system needs to be formalized mathematically. The modeler should assess the nature of requirements and should take a decision about the specification procedure. Generally, safety-critical and fault-tolerance aspects of the system are given priority for formal specification.

[3] http://www.esterel-technologies.com/products/scade-suite/

- *Estimate costs.* Formal methods are notorious for higher startup costs, staff training, acquisition of supporting tools, consultants, etc., which increase the overall cost of the project. A feasibility study and cost analysis must be done at the start of the project against the expected Return On Investment (ROI).

- *Have a supervised formal development.* Development of projects with formal methods sometimes become tricky. For novices, it is recommended to have it supervised under the guidance of some expert, consultant, guru, etc. Desired results are hard to reap with un-supervised first time experience with formal methods.

- *Document sufficiently.* Formal methods are known for their concise, unambiguous, and consistent specification documents. However, it is recommended that a natural language explication of formal specifications should also be there for further reinforcement of the reader's understanding of the system.

- *Do not compromise on quality.* The use of formal methods is not a "silver bullet" [Brooks 1987] for software crisis. However, if applied in the appropriate way, formal methods can achieve higher system integrity but still, perfect software can not be guaranteed. Therefore, it is better to employ other measures too to ensure that software complies with its quality standards.

- *Do not be dogmatic.* Formal methods are no panacea either, therefore it must be clear that their employment would not provide 100 percent assurance of correctness. So, a formal development may also contain small omissions, minor bugs, and other attributes that do not meet expectations.

- *Do not abandon traditional development methods.* The best results can be obtained by the incorporation of formal methods into the existing traditional software development, object-oriented, for example.

- *Test sufficiently.* Testing has always been considered as a successful manner to find bugs in a software. This integral part of traditional software development should be continued as a standard operating procedure even during the development inspired by formal methods. It will further improve its quality.

- *Reuse.* In the longer run, when it comes to cost of development and quality, the only rational answer to this question is reusability. This is the same case, even with formal software development.

2.5 Summary

In this chapter, we have presented the triptych of modern software development which is based on the pillars of domain engineering, requirement engineering and formal methods. We have defined what are domains and how to model them. We then relate domain engineering with requirements engineering because in a broader

spectrum the former is the part of the latter. We saw the different types of requirements, and phases and methods for their modeling. In the last section of the chapter, we shed some light on formal methods and discuss their advantages, disadvantages, myths and guidelines.

Chapter 3

Domain specification, verification & validation

Contents

3.1 Introduction

The three important phases of a software development life cycle, inspired by formal methods, are specification, verification and validation. While the purpose of the specification is to document the requirements of the system to be built either in natural language, graphically or formally, the later two are employed to measure the correctness of the system. As the complexity of software systems keeps increasing, we clearly see two paradigm shifts. First, traditional verification and validation methods goes from informal to semi-formal or fully formal. Second, they are introduced in development as early as possible. Since domain engineering precedes requirements engineering, it may be the first phase subjected to all these software quality assurance activities.

Though both verification and validation amount to measure the correctness of software, yet they are distinct activities. One simple difference between them, according to [Boehm 1978], is that verification answers the question "are we building the product right?", whereas validation answers "are we building the right product?" The outcome of each activity is a different artifact. Figure 3.1 explains this

fact where a verification technique, proving, yields a verified specification and a validation technique, animation, produces a validated specification. However, the desired result is to produce a unique document which is amenable to both verification and validation. This is one of the scientific questions which we later address in this thesis.

Figure 3.1: Verification vs. Validation

In this chapter, we explain, in detail, the phenomena of domain specification, verification and validation as well as different approaches towards these techniques. The organization of the chapter is as follows: Section 3.2 describes the process of domain specification. Section 3.3 discusses the idea of domain verification and its two techniques: model checking and theorem proving. Section 3.4 introduces the technique of domain validation and its supported methods. The chapter is finally summarized in section 3.5.

3.2 Domain specification

Domain specification is a complete description of the behavior and properties of a domain. This document intends to specify the domain's intrinsic and extrinsic properties, facts, laws, enforceable requirements, etc. While the specification of a domain can be written in natural language, it may be ambiguous, error-prone and difficult to verify and validate.

Formal methods come to the rescue at this point as formal specifications, written in some precise formal and mathematical language, express domains at same level of abstraction for all stakeholders. The precise description of its facts gives stakeholders a deeper understanding of the domain being specified. This practice is helpful for uncovering design flaws, inconsistencies, ambiguities and incompletenesses. Such specifications contribute to closing down the communication gap between customers, analysts, designers, developers, and testers.

In order to specify domains, according to [Lamsweerde 2000], a formal specification language must be composed of three integral components: a syntax, i.e., rules for determining the grammatical well-formedness of sentences, a semantics, i.e., rules for interpreting sentences in a precise way, and a proof theory, i.e., rules for inferring useful information from the specification. Structuring relationships, such as specialization, aggregation and instantiation, in addition to integral components, are also highly recommended. The other desirable properties of a good specification language are: adequacy, i.e., its ability of stating the problem adequately, consistency,

i.e., its ability of semantic interpretation that makes true all specified properties taken together, etc.

The use of formal methods for the specification of domains depends upon the nature of the systems it models. Some formal methods, such as B, Event-B, Z, VDM, focus on specifying the behavior of sequential systems. States of the system are expressed in terms of mathematical structures like sets, relations, axioms, functions and events, and transitions among states are expressed with pre and post-conditions. Some formal methods, such as Communicating Sequential Processes (CSP) [Hoare 1985], Calculus of Communicating Systems (CCS) [Milner 1982], Statecharts [Harel 1987], Temporal Logic [Pnueli 1981], Language of Temporal Ordering Specification (LOTOS) [Logrippo 1990], focus on specifying the behavior of concurrent systems. States of the system are either expressed by simple domains like integers or are left uninterpreted, and behavior is expressed with sequences, trees or partial orders of events. Some formal methods, such as RAISE, facilitate the description of richer state spaces in combination with concurrency mode. These specification languages are usually closer to programming languages.

Some specification languages (not strictly formal though), such as UML and Specification and Description Language (SDL) [Olsen 1994] are sometimes considered for the specification of general purpose or process-based concurrent systems respectively.

3.3 Domain verification

Domain verification is a process which is employed to judge the ability of the domain specification to comply with regulations/conditions incepted at the time of its writing. Used in a broader sense, i.e., related to the overall software development, the goal of verification is to assure that the software fully satisfies all the expected requirements. Techniques for such kind of verifications are classified into two categories: dynamic verification which is also known as testing and is good for finding bugs, and static verification which is also known as analysis and is useful for proving correctness of a domain.

Prior to domain verification, it is imperative to determine the verification approach and that the facts are adequate. Once a desirable verification technique has been selected then it is applied over the specification to check if the facts defined in it are consistent. The purpose of verification at this phase is to verify the correctness, completeness and consistency of the domain model. Inspections or reviews can be performed to determine that all the facts have been stated (this is more related to validation than verification and would be discussed in detail later in the chapter).

The process of formal verification, based on mathematical techniques, verifies that a specification conforms to some precisely expressed notion of functional correctness [Bjesse 2005]. In formal verification, the goal is to search for some input patterns that the environment could generate that will violate the given properties that must hold.

Formal verification, which proves or disproves the correctness of a model with respect to certain formal specification or property using formal methods, is equally suitable for a variety of software and hardware systems ranging from cryptographic protocols and safety-critical systems to digital and combinational circuits. The underlaying rationale, as described above, is to provide a formal proof of the correctness of an abstract mathematical model of the system.

According to [Huth 2000], formal verification techniques generally consist of three main components: a framework for modeling, i.e., a specification language, a property specification language, i.e., a temporal logic, and a verification method supporting the checking of properties in the model. The verification techniques employed, such as model checking or theorem proving, depend upon certain factors, such as tools which are sometimes completely automated or sometimes require human intervention, specification of the model which ranges from high-level to very fine grained, and application domains, i.e., concurrent systems, sequential systems, reactive systems.

According to [Clarke 1996], two well established formal verification approaches are model checking, i.e., the exploration of the whole state space to verify a property, and theorem proving, i.e., rigorous mathematical reasoning about system properties. These formal verification techniques, though effective, also have some limitations, such as the problem of state space explosion with model checking, and the problem of complexity and noticeable human interaction with theorem proving. Following is a brief introduction to both techniques.

3.3.1 Model checking

To employ the technique of model checking, a formal model of the domain is constructed in the first place and then it is checked if certain properties hold in this model. This is done by an exhaustive search of the state space, therefore some kind of finiteness is mandatory. A finite model of large size is still a problem, therefore states of the model must be reduced to some reasonable size. Two general approaches widely practiced in model checking, according to [Clarke 1996], are temporal model checking [Clarke 1982], i.e., modeling with state transition diagrams where properties are specified by temporal logic [Pnueli 1981], and automata where both the model and its properties are specified as automata and are compared for property verification by notation, such as refinement orderings [Cleaveland 1993], observational equivalence [Roy 1992], etc.

The advantages of the technique of model checking are the automated tool support, relatively fast verification, generation of counter examples, support for partial specifications, ease of use, etc. The disadvantages of this technique are that it may not scale up because of state explosion (there also exists some solutions to this problem like Binary Decision Diagrams [Bryant 1986]), etc.

Some notable model checkers are Simple Promela INterpreter (SPIN) [Holzmann 1997], KRONOS [Bozga 1998], Software, Languages, Analysis, and Modeling (SLAM) [Ball 2004], and Berkeley Lazy Abstraction Software verification Tool (BLAST) [Henzinger 2003].

3.3.2 Theorem proving

To employ the technique of theorem proving, the domain and its properties are first documented using (first or higher-order) logical statements and then it is shown through mathematical proofs that some of the statements, such as conjectures, are logical consequences of a set of other statements, such as axioms and hypotheses. These statements, defined with precise semantics, i.e., there is no ambiguity in problem statement, are complemented by inference rules. The technique of theorem proving can be used for domain verification either in automated mode, i.e., with automated tools, or interactive mode, i.e., proofs by hand. The technique of theorem proving is more popular in the domain of safety-critical systems for both of the hardware and the software designs [Clarke 1996].

The advantages of the technique of theorem proving are its application to infinite state space, the existence the proof object, etc. The disadvantages of this approach are considerable human interaction, slower results, learning curves, the problem of deciding the validity of a formula (depending upon the underlying logic), etc.

Some notable theorem provers are Isabelle [Nipkow 2002], Higher Order Logic (HOL) [Gordon 1993], A Computational Logic for Applicative Common Lisp (ACL2) [Kaufmann 1996], and Coq [Dowek 1993].

3.4 Domain validation

Once a specification is written and verified, the next step to consider is its validation. Verification alone is not sufficient to guarantee the correctness of the model because verification does not check whether the specification documents the useful requirements. Here is the point where validation comes into action and answers the correctness and completeness problems in the specification and demonstrates that the domain is fully expressed. The process of validation is an interrelated and complementary process to verification. Also, it provides comprehensive approach towards the testing in general and evaluation, review, inspection, and assessment in particular.

The process of domain validation is similar to the process of software validation where it is established by examination and provision of objective evidence that all the user, system, software and other requirements have been captured correctly and completely in the requirements specification document.

Following is a brief introduction to some of well-known validation techniques. See Gemino [Gemino 2004], Yousuf et al. [Yousuf 2008], and Raja [Raja 2009] for more details.

3.4.1 Prototyping

Domain prototypes are representations, simulations or demonstrations of a particular domain which help stakeholders discover problems in the model. The underlaying rationale behind domain prototyping is to understand the domain requirements as

they are. This method is equally good for domain elicitation as well as its validation. A well-designed prototype helps engineers understand the intended behavior of the domain. This exercise also pays off in the form of reduction of development efforts by the integration of these prototypes into the final product.

Prototypes provide a channel for communication between developers, users and management. Customers which become part of the domain validation process, help in discussing particular problems, clarifying particular questions or preparing particular decisions. Explaining the requirements to implementers, which is generally considered as a difficult task, becomes relatively simplified by employing the technique of prototyping [Budde 1990].

3.4.2 Animation

Animation, according to [Marc 1993], is a form of domain visualization that is related to dynamic and interactive graphical displays of its specification's fundamental operations. This technique is well-adapted for the concept of validation where animation demonstrates the behavior of a domain through a visual interface which reflects the state of events in graphical and (where appropriate) textual formats [Lalioti 1993]. It is a kind of rapid prototyping. The benefit over here is that we can convert the specification into a prototype without translating it into code.

In a typical animation session, some scenarios are created and then a walk-through of a specification fragment is conducted in order to follow the scenario and analyze the behavior of the domain. Any event and conceptual information are abruptly reported to the user through the animator. Animation, like this, can be used to determine causal relationships embedded in the specification or simply as a mean of browsing through the specification to ensure adequacy and accuracy by reflection of the specified behavior back to the user [Lalioti 1993].

The technique of animation focuses on the behavior of the system [Van 2004]. The principle is to simulate an executable version of the model and to visualize the simulation in some form appealing to stakeholders. This point of view is well adapted to the paradigm of events and event-based specification languages, such as Event-B. Animators use finite state machines to generate a simulation process which can be then observed with the help of UML diagrams, textual interfaces, or graphical animations [Ponsard 2007]. If deployed for the validation of event-based systems, animation checks if the events are fired in a sequence that follows the protocol described by the original requirement document. When applicable, the computation of values can also be used to check that the state of the model evolves in a way consistent with the desired intentions.

Animation can be used early during the elaboration of the specification; there is no need to wait until it is finished. As a relatively low-cost validation activity, animation can be frequently used during the process to validate important refinement steps. It is a validation process which is consistent with the refinement structure of the specification languages. This property of animation is of interest due to several reasons. One of them is the fact that problems are detected close to the point where

their cause was introduced. This facilitates the understanding of the cause. Another reason is the fact that an unforeseen behavior may be associated with a specific refinement. If refinement is seen as the formalization of a requirement, then it hints an indication that some interactions between requirements need to be investigated.

The concept of visualizations of domains is not new. Program visualizations have been previously used for designing, developing, monitoring and debugging software. Some notable visualization environments spanning across different areas of interest are graphics interface development [Clemons 1985], visualization of concurrent processes [Roman 1989], etc.

3.4.3 Reviews

Review is a process during which domain models are reviewed by a group of people (e.g., users, customers, analysts, specifiers). These reviews are either formal, although not in the mathematical sense, i.e., governed by agreed rules, such as walkthroughs, technical reviews, software inspections, or informal, i.e., relatively unstructured activities, such as "buddy checking." IEEE Standard 1028-1997 [IEEE 1998a] defines formal structures, roles, and processes for each of the aforementioned formal review processes.

Formal reviews are generally preferred over informal reviews because they are more cost-effective, less time-consumption, and better focused on the discovery of real defects and their repair. These reviews help customers and developers to resolve problems at early stages of development. The time consumed during this activity eventually pays back by minimizing the changes and alteration in the software.

Though [IEEE 1998a] defines a set of general activities around formal reviews, yet there is no standard review process for domain engineering. Customized review processes can be designed according to needs, customers and markets. However, as mentioned by [Kotonya 1998], the review process may consist of steps, such as planning a review by the selection of a team, a time and a place for the review meeting, distributing documents among review team members, holding review meeting by discussing problems and their possible solutions, follow-up actions to check if the agreed upon actions are taken place or not, and revising the final document for the acceptance or re-reviews.

While each review technique has its merits, the extent to which an informal approach can be applied to a large complex specification is questionable. Furthermore, the use of formal technique, sometimes, requires the modeler to take certain design decisions for implementation of the domain.

A validation method very similar to reviews, known as inspection [Fagan 1976], is also used for validation purposes. However, it is not a common practice in industry due to the cost which is associated to it.

3.4.4 Structured walkthroughs

Structured walkthrough [Yourdon 1978] is a validation method for identifying inaccurate and inconsistent facts present in the specification. A walkthrough is a meeting for evaluation where domain specifications are inspected for inconsistencies. It can be effectively utilized for the validation of informally defined textual specifications and small-scale formal specification documents. The main ingredients of a walkthrough are the review body including independent experts of the domain, the distribution of review material, the problems identification, and the follow-up meeting (if required). A walkthrough is generally organized and supervised by the modeler, however any interested or technically qualified person can be part of the review process. The detailed process of walkthrough is documented in [IEEE 1998a].

Walkthroughs, often also known as design reviews, differ from technical reviews because of their flexibility in the structure of the method and objectivity. They are also different from inspections because of their ability to hint about explicit changes in the model. Lack of direct focus on training and process improvement is another measure which differentiates them from inspections and makes them a less popular validation method.

3.5 Summary

The three very important cornerstones of domain engineering are domain specification, verification and validation. While the former captures and documents the domain facts, laws and properties, the latter two are used to prove their correctness. This chapter throws light on these three activities. We have defined what they are, how they can be used and what are the different techniques to perform them. The techniques of theorem proving and animation, which have been used extensively in this thesis, are also discussed.

Part II

SPECIFICATION

Chapter 4

Event-B

Contents

4.1 Introduction

Formal development can be complex; that is why sophisticated and easy-to-use tools are needed. They facilitate the application of formal methods by abstracting away some of the difficulties behind convivial interfaces and also by making easier the specification, verification and validation processes.

Event-B [Abrial 2007] is a formal language for modeling and reasoning about large reactive and distributed systems. Event-B is based on set theory with the ability to use 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].

In this chapter, we discuss the formal language Event-B. It is organized as follows: Section 4.2 presents Event-B's structuring mechanism. Section 4.3 introduces its concept of refinement. Section 4.5 discusses its idea of decomposition. Section 4.6

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

discusses its tool support, Rodin. Section 4.7 compares Event-B with RAISE, a formal method often used for domain engineering, discusses its previous involvement in transportation sector and presents some reflections about its limitations. The chapter is summarized in section 4.8.

4.2 Structuring mechanism

An Event-B model is composed of two constructs: machine and context. A typical machine, as shown by figure 4.1, defines the dynamic element of the model and contains the system variables, invariants, variants, and events. Variables are typed, their values may be integers, sets, relations, functions or any other set-theoretic construct. Invariants define the state space of the variables and their safety properties. Variants are related to the correction of refinements.

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. A typical context, as shown by figure 4.2, defines the static elements of the model and contains carrier sets, constants, axioms, and theorems. The last two are predicates expressed within the notation of first-order logic and set theory.

MACHINE
Movement0
SEES
StartState
VARIABLES
location
INVARIANT
inv1 location $\in$ Vehicles $\rightarrow$ GlobalLocations
EVENTS
INITIALISATION $\widehat{=}$
BEGIN
act1 location := startVehicleLocation
END
travel $\widehat{=}$
ANY
vehicle , newLocation
WHERE
grd1 vehicle $\in$ Vehicles
grd2 newLocation $\in$ GlobalLocations
grd3 newLocation $\neq$ location(vehicle)
THEN
act1 location (vehicle) := newLocation
END
END

Figure 4.1: Machine `Movement0`

CONTEXT
Location
EXTENDS
Net
SETS
GlobalLocations
CONSTANTS
hubLocations, obsHubLocations, connectionLocations, obsConnectionLocations
AXIOMS
tec1 GlobalLocations $\neq \varnothing$

typ1 hubLocations $\subseteq$ GlobalLocations
typ2 connectionLocations $\subseteq$ GlobalLocations
typ3 obsHubLocations $\in$ Hubs $\rightarrow \mathbb{P}$(hubLocations)
typ4 obsConnectionLocations $\in$ Connections $\rightarrow$ $\mathbb{P}$(connectionLocations)

pro1 hubLocations $\neq$ connectionLocations
pro2 hubLocations $\cup$ connectionLocations = GlobalLocations
pro3 $\forall$ h1, h2 . h1$\in$ Hubs $\wedge$ h2 $\in$ Hubs $\wedge$ h1$\neq$h2 $\Rightarrow$ obsHubLocations(h1) $\cap$ obsHubLocations(h2)=$\varnothing$
END

Figure 4.2: Context `Location`

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 and concrete description of the model. A context can extend multiple contexts. It contains the static pieces of information of a model associated to a refinement. A machine can see, that is, use the names and properties of several contexts and a context can be seen by several machines.

4.3 Refinement

Refinement is a process to add details to a model in a stepwise manner. The advantage of this technique is that it allows the analysis of the model at reduced complexity. Event-B embeds the notion of refinement as the basic element of a specification development process. The abstract-refinement relationship needs to be proved when a new refinement step is introduced to the model.

A refinement consists of introducing either new variables, new events, or new invariants. When appropriate, an abstraction invariant, often called *gluing-invariant,* relates the new variables to the abstract ones. Individual events can be refined too by strengthening their guards and adding actions on the new variables. The same abstract event can be refined into several concrete ones. When a concrete event refines an abstract one which is parameterized, then all abstract parameters must receive a value in the concrete event. Such values are called the witnesses.

New events can also be introduced in the refinement process. Formally, they are refinements of the `SKIP` event. Most often, new events express how an abstract event is realized 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 prevent 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.

4.4 Proofs

The correctness of refinement is assessed with the help of 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.

Proofs also ensure that specifications meet essential system properties, such as safety and well-definedness. The proof obligations generated by Rodin are required to be discharged using proof tools, either automatically or interactively.

The typical proofs related to a context are about well-definedness of an axiom or a derived axiom. Proofs for the verification of a machine are more varied. They are about the well-definendness of invariant, derived invariant, guard, and

action, invariant preservation, feasibility of a non-deterministic event action, etc. The proofs related to refinement are guard strengthening, action simulation, equality of a preserved variable, etc. The proofs related to variants and witnesses are well-definedness of variant, decreasing of variant, well-definedness of witness, etc.

4.4.1 Proof of invariant preservation

In order to prove that a machine maintains the invariant, we need to prove the following condition:

$$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \wedge R(s, c, v, v') \Rightarrow I(s, c, v')$$

Here v denotes the variables of the machine (v' is the value after execution) which sees the context C with sets s and constants c. The properties of the constants are denoted by $P(s, c)$ and the invariant by $I(s, c, v)$. G(s, c, v) is the guard of an event whose before-after predicate is defined by $R(s, c, v, v')$.

4.4.2 Proof of event refinement

In order to prove that an event refines an abstract event, we need to prove the following condition:

$$\begin{aligned} &P(s, c) \wedge I(s, c, v) \wedge Q(s, t, c, d) \wedge J(s, t, c, d, v, w) \wedge \\ &H(s, t, c, d, w) \wedge S(s, t, c, d, w, w') \Rightarrow \\ &\quad G(s, c, v) \wedge \exists v'.(R(s, c, v, v') \wedge J(s, t, c, d, v', w')) \end{aligned}$$

Here d, t, w represent new constants, sets and variables respectively, $J(s, t, c, d, v, w)$ and $Q(s, t, c, d)$ are the gluing invariant and new axiom respectively, and $H(s, t, c, d, w)$ is the guard of new event with before-after predicate $S(s, t, c, d, w, w')$.

4.4.3 Proof to introduce new events

In order to introduce a new event in the refinement, we need to prove the following three conditions:

1. Skip refinement

   In Event-B, every new event must be the refinement of the `SKIP` event. The following condition needs to be checked:

   $$\begin{aligned} &P(s, c) \wedge I(s, c, v) \wedge Q(s, t, c, d) \wedge J(s, t, c, d, v, w) \wedge \\ &H(s, t, c, d, w) \wedge S(s, t, c, d, w, w') \Rightarrow J(s, t, c, d, v', w') \end{aligned}$$

2. Non-divergence

   New events should not prevent older events from happening. The following condition needs to be checked:

$$P(s,c) \wedge I(s,c,v) \wedge Q(s,t,c,d) \wedge J(s,t,c,d,v,w) \wedge$$
$$H(s,t,c,d,w) \wedge S(s,t,c,d,w,w') \Rightarrow$$
$$V(s,t,c,d,w) \in \mathbb{N} \wedge V(s,t,c,d,w') < V(s,t,c,d,w)$$

Here $V(s,t,c,d,w)$ represents a variant.

3. Deadlock

   Concrete machine must not deadlock before its abstraction. The following condition needs to be checked:

   $P(s,c) \wedge I(s,c,v) \wedge Q(s,t,c,d) \wedge J(s,t,c,d,v,w) \wedge (G_1(s,c,v) \vee ... \vee G_n(s,c,v)) \Rightarrow$
   $(H_1(s,t,c,d,w) \vee ... \vee H_n(s,t,c,d,w))$

   Here G_i and H_i represent the abstract and concrete guards respectively.

4.5 Decomposition

The process of refinement can significantly enlarge the model because several new state variables and events are introduced as part of the refinement. The complexity of the model thus becomes hard to manage and the model needs to be cut into smaller and manageable units.

The idea of decomposition is to divide the large event system into smaller pieces; each piece is easily manageable as compared to the whole system and can be refined independently. However, the constraint that must be satisfied by the process of decomposition is that these independently refined pieces must be, later, transformable into q single large unit. Recomposition is then the process which results into the reunification of decomposed pieces into a system that could have been obtained directly without the decomposition.

Models in Event-B can be decomposed on two bases: event-based [Butler 1996] and state-based [Abrial 2007]. In event-based decomposition, a model is separated on the basis of its events. We encapsulate the events or part of them (in the form of new events), and their related variables in a separate component. The split events are then synchronized in order to recreate the functionality of the original model. This idea is also known as parallel composition.

In state-based decomposition, variables are split between different components; some of them are shared which are then duplicated. Events are added to components to simulate how the shared variables are used in other components. The components are then refined while keeping shared variables and events synchronized. The system can be rebuilt into a single model at the end of the refinement process.

4.6 Tool

Rodin is the tool that supports modeling in Event-B. Rodin is built upon the Eclipse platform[2] which allows it to be extended by plug-ins. Rodin supports the specification of machines and contexts, their refinement, and their consistency checking by automatically generating the proof obligations. Proofs can be discharged either automatically, or with the help of third-party theorem provers, or interactively.

A snapshot of the Rodin platform is illustrated in figure 4.3. On the left hand side, we have the projects of several specifications including machines (blue) and contexts (violet). In the middle, we have the main window. Here, we can write specifications in *Edit* mode or can see them in *Pretty Print* mode. On the right hand side, we can see the elements of a particular machine in the *Outline* menu. The menu *Proving* on the top right side enables the verification mode where proof-obligations can be seen and discharged. The section under the main window is used to show the errors and warnings. On the base of the right hand side, we have mathematical symbols which ease specification writing.

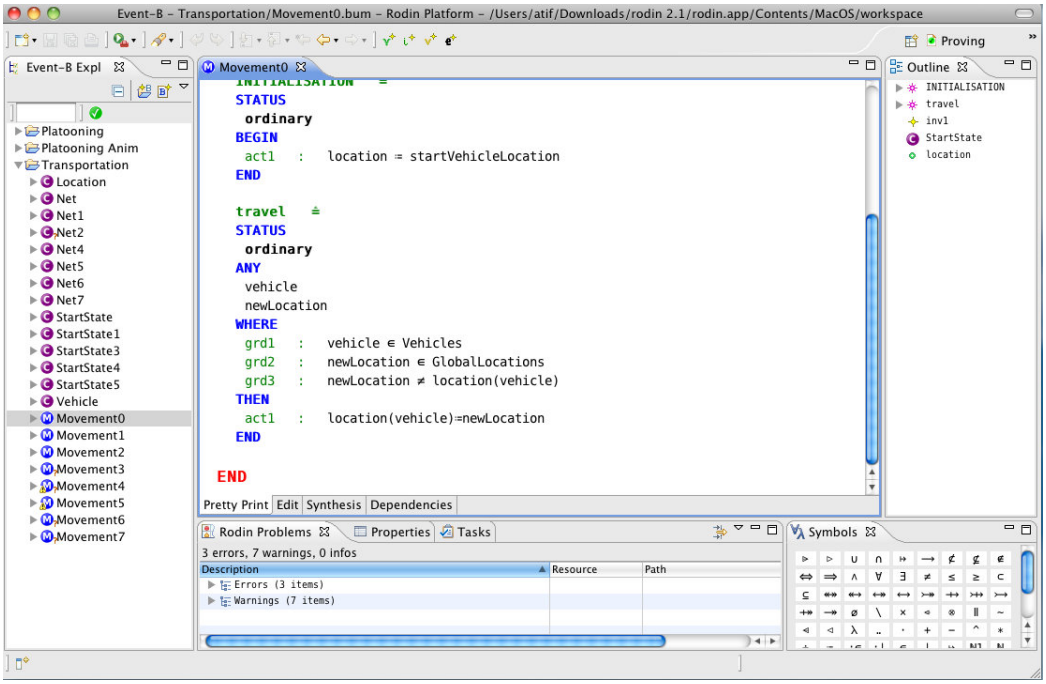

Figure 4.3: Rodin platform

There are several plug-ins available for the Rodin platform, such as model checkers, animators, theorem provers, decomposition plug-ins, etc. The Rodin platform does not inherently support decomposition/recomposition. This is work under progress.

[2] http://www.eclipse.org

4.7 Related work

4.7.1 Event-B versus RAISE

Rigorous Approach to Industrial Software Engineering (RAISE) [Group 1993b] aims to provide facilities for the industrial usage of formal methods in the development of software systems. RAISE consists of RSL which is a powerful specification and design language, a comprehensive development method [Group 1993a], and an extensive tool support. Its method is based on the stepwise refinement paradigm which produces each refinement step using *a posteriori* verification, i.e., a refinement step is developed and then the relation with previous step is verified. Verification under RAISE is generally obtained by the technique of formal proof which is assisted by tools. However, less formal techniques, such as rigorous arguments, can also be used if this suits the practical and economic constraints.

Just as B [Abrial 1996] evolved into Event-B, RAISE is considered as extended version of VDM. It includes the enhanced features of several formal techniques, such as model oriented features of VDM, algebraic features of OBJ [Futatsugi 1985], concurrency features of CSP, modularity features of Meta Language (ML) [Paulson 1996], 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 by 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 as 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 relation. Event-B, on the other hand, relaxes this strict 1-1 relation in a way 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. The refinement mechanism of Event-B has been explained in section 4.3.

RAISE achieves the notion of correctness of refinement through 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 liveness [Lamport 1977] of each machine separately, but authors like [Erasmy 1995, Gørtz 1994] have reported different experiences. According to [Gørtz 1994], the concurrent style in RSL cannot be used to specify pure progress properties, e.g., fairness, which is one of the liveness properties. According to Erasmy et al [Erasmy 1995], the justification editor of RAISE lacks rules for some important schemas to prove liveness, such as

parallel or sequential combinators. The poor ability of Event-B to describe temporal properties is discussed in details in the section 6.2.

The tool support for RAISE is 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) [Milner 1997] and Prototype Verification System (PVS) [Owre 1992], 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 parallel combinator, interaction of channel hiding and the parallel combinator, etc., yet overall its toolset is easy to use, uniform and relatively fast.

The extensive tool support for Event-B is one of its powerful aspects. Event-B is supported by the platform Rodin which helps the writing and proving of specifications. The Atelier-B [Cle 2009] provers provide additional automated proof facilities to the existing Rodin provers. Animators like ProB [Bendisposto 2008], AnimB[3] and Brama make possible the execution of specifications for their validation. UML-B plugin [Snook 2006] allows users to translate UML models into Event-B specifications for verification and validation. B2Latex plugin allows for the printing of Event-B specifications into latex for documentation purposes. One can also run B models into the Rodin platform with the help of B2Rodin plugin. There are also plugins for model decomposition, recomposition and code generation.

4.7.2 Event-B & goal models

The relation between KAOS [van Lamsweerde 2009] and Event-B is an active research field. [Bisztray 2008, Aziz 2009] propose a syntactic extension and patterns to model the notion of *obligation* introduced by a temporal model into Event-B. [Matoussi 2008, Mashkoor 2010b] explore a similar approach. Both approaches rely on systematic transformation rules to derive an Event-B model from a KAOS model. They are consistent with the idea of gradual introduction of formal elements during the specification process. They provide a tool that clarifies the relationship between Event-B and KAOS.

4.7.3 Modeling of transportation domain in Event-B

Previously, Event-B has been employed for the development of transportation systems, see for instance [Papatsaras 2002, Butler 2002, Essamé 2004], 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 transportation domain model, which will be

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

presented later in the thesis, is more general and could be used for different kinds of transportation systems, such as road, railways, conveyors, etc.

4.7.4 Refinement mechanisms in Event-B

Linear refinements are the de facto standard of specification development in Event-B. Its counter parts, other than observation levels [Mashkoor 2010a], discussed later in the thesis, are Retrenchments [Banach 1998], Feature development [Poppleton 2007] and Parallel refinements [Abrial 2007].

Retrenchment was proposed as a liberalization of the notion of refinement to capture more informal aspects of development within a formal framework. A retrenchment step from an abstract to a concrete level of abstraction allows strengthening of the precondition, weakening of the postcondition, and mixing of state and I/O information between the levels of abstraction by mediation of two extra predicates per retrenchment. In particular, it allows non-refinement-like behavior to be expressed via the weakened postcondition. This allows the specification of low level details of the model without cluttering up the formal text by unnecessary code which is mandatory to discharge proofs.

Feature oriented specification development is a mechanism to specify behavioral variability of the model. A feature, in this case, is a simple B machine which is atomic with respect to other functionalities. The composition of several machines, each highlighting a distinctive feature, will form the final model. Composition of these features is an issue. A Rodin plugin for this approach has been proposed by [Gondal 2009].

Parallel refinements is another idea of model decomposition to handle the complexity introduced by linear refinements. In this technique, a large model is cut into several smaller components. The model decomposition is either based on shared variables [Abrial 2007] or shared events [Butler 2009]. The supporting tool is presented by [Silva 2010].

Our view towards refinement of Event-B models by grouping them into observation levels is different from all of the aforementioned methods. They require intricate proof obligations to measure the correctness of the model and a change in the language. Our approach can be adopted without touching the semantics of the formalism and by just providing a visual modification at the level of the tools.

4.7.5 Specification of timing & temporal properties in Event-B

The specification of timing and temporal properties in Event-B is known to be a challenging task. The expression of such properties, a key element for utilization of formal methods in the automotive sector, is currently unsupported by existing Event-B tools. Correctness of such specifications thus becomes an issue.

In this work, we have used the timing pattern of Event-B proposed by [Cansell 2007]. Like us, [Bryans 2010] also reuses the same pattern with minor modifications. Similarly, the pattern of Joochim et al [Joochim 2010], proposes the use of 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.

Abrial et al. [Abrial 1998] proposed the use of temporal properties to model the dynamic elements of a model in Event-B specifications. In its continuation, [Groslambert 2006] proposed the use of Linear Temporal Logic (LTL) for such modeling. Like others, [Bicarregui 2008] also proposes 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. The point is that the expression of temporal properties is still a challenge in Event-B.

4.8 Summary

Event-B is a formal method for modeling and analysis. It uses set theory as a notation, refinement for incremental specification development and technique of theorem proving to verify the model.

In this chapter, we have discussed Event-B in details and described its structuring mechanism, its process of refinement, its proof system, its decomposition structure, and its tool support, Rodin. We have also compared this with RAISE, another formal method often used for domain engineering. Along with other related work, some of its limitations have also been discussed.

Chapter 5

Engineering of a domain

Contents

5.1 Introduction

An engineered system never works in isolation. On the contrary, it must operate within an environment which has its own laws, properties and constraints. Thus, a complete specification should include both descriptions: the system and the domain. Techniques, tools, methods are analogous in both types of specifications, but they also differ on important aspects.

In this chapter, we focus on the specification of a domain and we describe the environment which provides laws and constrains behaviors of systems of the domain. We provide a consistent, formal and effective domain model of land transportation, written in the Event-B formal specification language. The stepwise refinements of

the model deal with independent motions of vehicles and specify two properties: collision avoidance and travel time. A special attention is devoted to the verification of the model. Consequently, we provide a model with all proof obligations discharged.

The chapter is organized as follows: Section 5.2 presents an overview of the transportation domain. Section 5.3 specifies the Event-B model of the domain. Section 5.4 illustrates the hierarchy of the model. Section 5.5 discusses the verification of the domain. The chapter is finally concluded in section 5.6.

5.2 Domain overview

Our work takes place within the framework of the projects TACOS[1] and CRISTAL[2]. These projects aim at studying new transportation systems using autonomous and self-service vehicles known as CyCabs [Baille 1999]. 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 the last mode, several CyCabs assemble as a train without material connections between the cars. Except for the leader which can be manually driven, platoon 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 transportation 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 transportion domain is aimed at providing us with the formal expression of these properties.

The term "transportation" refers to the movement of people or goods by vehicles from one location to another. Many important concepts appear in this definition of transportation, which must be addressed during the transportation domain description, such as vehicles, locations, movement, etc. Following are the intrinsic concepts of the transportation domain which relate to the movement of vehicles:

[1] http://tacos.loria.fr
[2] http://www.projet-cristal.org

5.2.1 Locations

Each vehicle captures a location at any given moment. In this work, a location refers to the logical localization of a vehicle in contrast to its physical localization with geographical coordinates.

5.2.2 Nets, hubs & connections

We assume the transportation network which is constituted of a set of hubs connected by connections. A hub is the abstraction of a place roods meet (junctions) or where a vehicle can stop (stations). A connection is an abstraction of a directed road link between two hubs. Figure 5.1 shows a typical net with hubs and connections.

5.2.3 Junctions & stations

We consider two kinds of hubs: junctions and stations. Junctions model road intersections and other crossings where safety requires special attention. Stations model places where passengers can hop in and off vehicles, and where vehicles can be parked. Stations, but not junctions, are valid destinations of a travel.

5.2.4 Paths & routes

Operative connections between hubs are built on top of a physical network. At a physical level, adjacent hubs are connected through paths, which are, in fact, directed edges connecting vertices. A connection will then often be realized as a sequence of paths, which is called a route.

5.2.5 Properties

The general properties we want to express concerning transportation are collision avoidance and travel time. They are defined as follows.

Collision Avoidance

The term collision refers to the situation where two vehicles are at the same place at the same time. Collisions can be classified in three types: front, rear and side.

Time

Time is a very important notion in the domain of transportation. This is related to the fact that travel time is at the root of nearly all decisions made around transportation, either individually or socially. 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 which controls the behavior of vehicles.

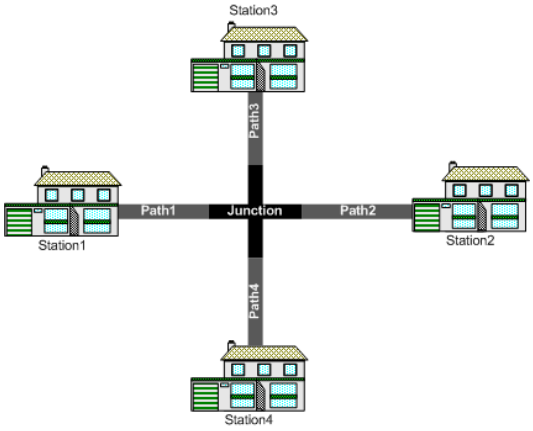

Figure 5.1: An abstract transportation network

5.3 Stepwise Event-B specification

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

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 crossings events corresponds to a change of observation level. Figure 5.2 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`.

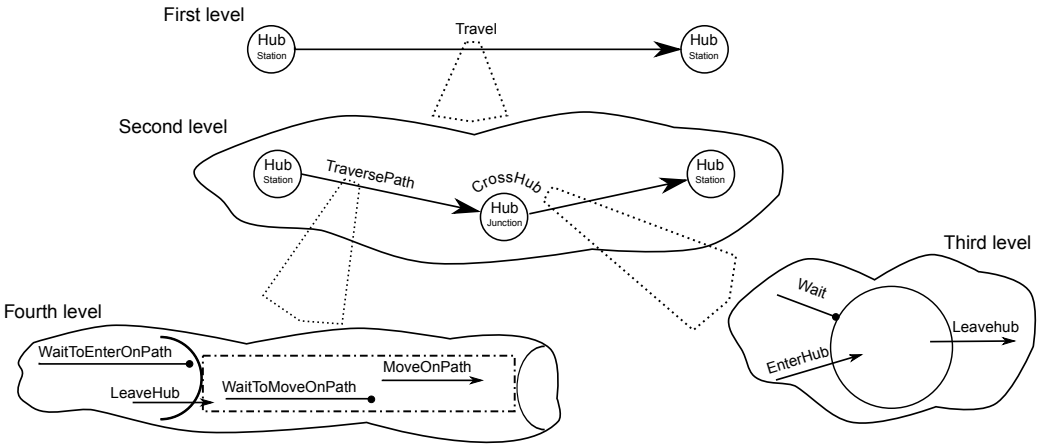

Figure 5.2: Levels of observations

New observation levels were introduced when a property could not be expressed within the existing levels. We can also go back and forth between these levels when we feel that a new property to be introduced is better suited to some other level of observation. This practice helps us to better integrate new properties into the existing model. For example, when we wanted to introduce the notion of time into the model, we did that in two steps. First, the vocabulary and abstract constraints (time is ever increasing, for instance) were defined at the first level of observation because this concerned only travels. Then, we jumped directly to the fourth level to define the computation because durations could be associated to events at this level. On the other hand, the property of collision avoidance was the reason to go from the second to the third and then from the third to the fourth level of observation.

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. 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. We also introduce the notion of travel time at this level.

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 hubs. Such collisions are abstractly defined as the presence of too many vehicles on a hub at the same time. This leads 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 on the hub).

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 vehicles. We also associate the notion of time to each event at this level.

It should be noted that Event-B lacks an explicit notion of time. We then decided to use the timing patterns for Event-B proposed by Cansell et al [Cansell 2007]. In this technique one use natural numbers to represent units of time and events to simulate a clock.

To specify contexts, three types of axioms have been used in the model: typing axioms, which set the type of the constants, property axioms, which specify the constraints and properties of the constants, and technical axioms, which specify the technical axioms required to discharge proof-obligations.

5.3.1 Initial model

This abstract model of the specification is based on contexts `Vehicle`, `Location`, `StartState`, `Net`, and machine `Movement0`.

At this level, the context `Vehicle` only introduces the notion of vehicles into the model. Later, when the model will evolve, properties related to vehicles will be added to this context.

The context `Location` introduces the concept of locations into the model. Locations are places which are occupied by vehicles. For our work, we consider two kinds of locations: `hubLocations` and `connectionLocations`. Both of these locations are subset of `GlobalLocations`, a set which models all locations of a network. All locations, either on a hub or on a connection, are observable via functions `obsHubLocations` and `obsConnectionLocations` respectively. Context `Location` is shown in figure 4.2.

Context `StartState` and its refinements are defined for ease of proofs; they do not bring new information into the model. They define the initialization statements for the variables of the machines which see them. For example, at this level, context `StartState` simply defines a constant `startVehicleLocation` which states that every vehicle has a starting global location and is then assigned to the variable `location` in machine `Movement0`.

Context `Net`, shown by figure 5.3, is modeled to specify the general transportation network facts, which are the following: A transportation net contains the notions of hubs and connections. This fact is represented in the context, for a hub, with the help of a relation between the sets of hubs and nets (`typ1`), and for a connection, with the help of a total function between the sets of connections and nets (`typ2`). The origin and the destination of each connection is always a hub (`typ3`

and typ4). The constant hubConnection refers to the fact that each hub has one or more connections attached to it, i.e., disconnected hubs are not possible in the domain (typ5). Reflexive connections are not permitted in the domain, so both the origin and destination hubs of the connections should be different (pro3). Each network is composed of at least two hubs and one connection (pro4). All hubs and connections connected to each other must belong to the same net (pro5 and pro6). A connection always belong to the same net (pro7), whereas a hub may belong to more than one net (pro8).

The basic definition of movement is "change in position." Machine Movement0, shown by figure 4.1, specifies this fact in an abstract event, called travel, in which a vehicle changes its current position to a new position. The position of a vehicle is specified with the variable location. An invariant is specified over this variable which states that a vehicle can hold any GlobalLocation over a net. As this is an abstract event, so we do not add any further information to it.

CONTEXT
Net
SETS
Nets, Hubs, Connections
CONSTANTS
obsNetHubs, obsNetConnections, connectionOrigin, connectionDestination, hubConnection
AXIOMS
tec1 finite (Nets)
tec2 finite (Hubs)
tec3 finite (Connections)
tec4 finite (obsNetConnections)
tec5 finite (obsNetHubs)
tec6 $Nets \neq \varnothing$
tec7 $Hubs \neq \varnothing$
tec8 $Connections \neq \varnothing$

typ1 $obsNetHubs \in Hubs \leftrightarrow Nets$
typ2 $obsNetConnections \in Connections \rightarrow Nets$
typ3 $connectionOrigin \in Connections \rightarrow Hubs$
typ4 $connectionDestination \in Connections \rightarrow Hubs$
typ5 $hubConnections \in Hubs \rightarrow \mathbb{P}(Connections)$

pro1 $dom(obsNetHubs) = Hubs \land ran(obsNetHubs) = Nets$
pro2 $dom(obsNetConnections) = Connections \land ran(obsNetConnections) = Nets$
pro3 $\forall c \cdot c \in Connections \Rightarrow connectionOrigin(c) \neq connectionDestination(c)$
pro4 $\forall n \cdot n \in Nets \Rightarrow card(obsNetHubs^{-1}[\{n\}]) \geq 2 \land card(obsNetConnections^{-1}[\{n\}]) \geq 1$
pro5 $\forall c \cdot c \in Connections \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionOrigin(c)\}] \land obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionDestination(c)\}]$
pro6 $\forall h, c \cdot h \in Hubs \land c \in hubConnections(h) \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{h\}]$
pro7 $\forall c \cdot c \in Connections \Rightarrow card(obsNetConnections[\{c\}]) = 1$
pro8 $\forall h \cdot h \in Hubs \Rightarrow card(obsNetHubs[\{h\}]) \geq 1$
END

Figure 5.3: Context Net

5.3.2 First refinement

The goal of this first refinement step is to add more constraints to the basic `travel` event. The movement is now seen as a change in the position from one hub to another. Since our model permits only stations to be the starting and the ending points of the vehicles, so in our opinion, this is the next logical step we should take. Therefore, `travel` is refined to state that a vehicle changes its location from an origin hub to a destination hub as a consequence of this event. The following guards are added to the event, where both `origin` and `destination` are `stations`:

$$location(vehicle) \in obsHubLocations(origin)$$
$$newLocation \in obsHubLocations(destinations)$$

The definition of stations and junctions is given by context `Net1` which is a refinement of context `Net`. In this refinement, we state that hubs can be partitioned into two categories: stations and junctions. Stations are places where goods and passengers enter or leave a vehicle. Junctions, on the other hand, model the idea of intersections and crossings. We also state that a net is composed of two of more stations. Context `Net1` is shown in figure 5.4.

CONTEXT
Net1
EXTENDS
NET
CONSTANTS
junctions , stations , obsNetJunctions, isJunction , obsNetStations
AXIOMS
typ1 junctions $\subseteq$ Hubs
typ2 stations $\subseteq$ Hubs
typ3 obsNetJunctions $\in$ Nets $\rightarrow \mathbb{P}$(junctions)
typ4 isJunction $\in$ Hubs $\rightarrow \mathbb{B}$
typ5 obsNetStations $\in$ Nets $\rightarrow \mathbb{P}$(stations)

pro1 junctions $\cap$ stations $= \varnothing$
pro2 junctions $\cup$ stations $=$ Hubs
pro3 $\forall$ h . h $\in$ Hubs $\wedge$ card(hubConnections(h)) = 1$\Rightarrow$ isJunction(h) = FALSE
pro4 $\forall$ h . h $\in$ stations $\Rightarrow$ isJunction (h) = FALSE
pro5 $\forall$ h . h $\in$ junctions $\Rightarrow$ isJunction (h) = TRUE
pro6 $\forall$ n . n $\in$ Nets $\Rightarrow$ card(obsNetHubs^{-1}[{n}] $\cap$ stations) $\geq$ 2
END

Figure 5.4: Context `Net1`

5.3.3 Second refinement

Vehicles in our transportation domain follow a route in order to reach their destination from their origin. This fact is the basis to introduce this refinement step. In this refinement, it is assumed that the starting location of the vehicle is the origin of the route and the destination of the vehicle is the last hub of the route. In other

words, a vehicle is moving along a route in this refinement from origin to destination hub. The following guards are added to strengthen the `travel` event:

$$r \in routes$$
$$origin = connectionOrigin(r(1))$$
$$destination = connectionDestination(r(card(r)))$$

In order to give the definition of routes, we refine context `Net1` into `Net2`. This context introduces the concept of path, a connection between two adjacent hubs, and route, which is defined as a sequence of adjacent, connected and non-cyclic paths. Context `Net2` is shown by figure 5.5.

CONTEXT
Net2
EXTENDS
Net1
CONSTANTS
paths, routes, isRoute, seqPaths
AXIOMS
typ1 paths ⊆ Connections
typ2 seqPaths = { seq | ∃ n . n ∈ N1 ∧ seq ∈ 1..n ⤖ paths ∧ finite (seq) ∧ card(seq) = n}
typ3 isRoute ∈ seqPaths → 𝔹
typ4 routes = {sp | sp ∈ seqPaths ∧ isRoute(sp) = TRUE}

pro1 ∀ r . r ∈ seqPaths ∧ ((connectionOrigin(r(1)) ∈ stations ∧
connectionDestination(r(card(r))) ∈ stations ∧
(obsNetHubs[{connectionOrigin(r(1))}] ∩
obsNetHubs[{connectionDestination(r(card(r)))}] ≠ ∅) ∧
(∀ i . i ∈ 2..card(r) ∧ connectionDestination(r(i−1)) = connectionOrigin(r(i))) ∧
connectionOrigin(r(1)) ≠ connectionDestination(r(card(r))) ∧
(∀ i1, i2 . i1 ∈ 1..card(r) ∧ i2 ∈ 1..card(r) ∧ i1 ≠ i2 ⇒
connectionOrigin(r(i1)) ≠ connectionOrigin(r(i2))) ∧
(∀ i1, i2 . i1 ∈ 1..card(r) ∧ i2 ∈ 1..card(r) ∧ i1 ≠ i2 ⇒
connectionDestination(r(i1)) ≠ connectionDestination(r(i2)))) ⇔
isRoute(r) = TRUE)
pro2 ∀ c . c ∈ Connections ⇒ (connectionDestination(c) ∈ stations ∧
connectionOrigin(c) ∈ stations ⇒ (∃ r . r ∈ routes ∧
connectionOrigin(c) = connectionOrigin(r(1)) ∧ connectionDestination(c) =
connectionDestination(r(card(r)))))
END

Figure 5.5: Context `Net2`

5.3.4 Third refinement

This refinement corresponds to the second level of observation of the model. Here, the model is witnessing an abstraction leap. Previous refinement stated that a vehicle travels from an origin hub to a destination hub following a route. Naturally, using a route means that a vehicle must traverse many intermediate hubs and paths in order to reach its destination. Thus, modeling the phenomena of crossing hubs and traversing paths is the main reason for introducing this observation level.

In machine `Movement3`, three new events `startTravel`, `crossHub`, and `traversePath` are added. While the event `startTravel` is introduced as the responsible event for initiating travel, `crossHub` and `traversePath` are convergent events and direct decompositions of `travel`. The protocol for traveling now looks like this:

$$travel \equiv (startTravel;\ (crossHub;\ traversePath)+)^{3}$$

The event `startTravel`, as the name suggests, is responsible event for starting the travel of a vehicle. The condition to trigger this event is that a vehicle should be at the starting hub of a particular route which is not already assigned to it. If this condition is met then we assign the particular route to this vehicle in the form of hubs to cross and connections to traverse.

The specification of `crossHub` event is trivial. The following guards can be set in order to trigger this event: The vehicle should currently be positioned on the hub which it needs to cross and, moreover, that hub should not already be crossed by this vehicle. If these guards are true then this hub will be subtracted from the list of those hubs which need to be crossed by this vehicle.

The event `traversePath` is expressed with the help of the following guards: The vehicle is currently positioned on the origin hub of a path and that hub has already been crossed. The destination hub of the path has not already been crossed. The path connecting both hubs is the member element of the route the vehicle needs to travel and has not already been traversed by the vehicle. If all of these guards are true then the position of the vehicle will be set as destination hub of the path and the path will be subtracted from `connectionsToTraverse` variable.

A variant is also specified to prevent these convergent events from happening forever:

$$card(hubsToCross) + card(connectionsToTraverse)$$

Figure 5.6 shows machine `Movement3` and its events. Notice that a variable `position` has replaced the variable `location` in these events. Though, the same phenomenon is represented by both variables, yet two different variables have been chosen. The reason is simple: level of observation has been changed. At the abstract level, `location` refers to the initial and final positions of vehicles, and at the lower level of abstraction, the variable `position` is used to represent the intermediate positions of a vehicle on a net.

5.3.5 Fourth refinement

This fourth refinement step corresponds to the third level of observation. The rationale for this leap in observation level is to specify `crossHub` event at a fine grained level. In fact, we want to express the collision avoidance property on hubs and for this, we control the entrance of vehicles to it. In order to do so, the `crossHub` event is decomposed into three further events:

[3] See 6.2.2 for the discussion on defining protocols

MACHINE
Movement3
REFINES
Movement2
SEES
StartState3
VARIABLES
connectionsToTraverse, hubsToCross, position , location
INVARIANT
inv1 connectionsToTraverse $\subseteq$ Vehicles X Connections
inv2 hubsToCross $\subseteq$ Vehicles X Hubs
inv3 position $\in$ Vehicles $\rightarrow$ GlobalLocations
VARIANT
card(hubsToCross) + card(connectionsToTraverse)
EVENTS
...
traversePath $\widehat{=}$
ANY
vehicle , r, p, newPosition
WHERE
grd1 vehicle $\in$ Vehicles
grd2 r $\in$ routes
grd3 p $\in$ ran(r)
grd4 position (vehicle) $\in$ obsHubLocations(connectionOrigin(p))
grd5 newPosition $\in$ obsHubLocations(connectionDestination(p))
grd6 newPosition $\neq$ position(vehicle)
grd7 vehicle $\mapsto$ p $\in$ connectionsToTraverse
grd8 vehicle $\mapsto$ connectionOrigin(p) $\notin$ hubsToCross
THEN
act1 position (vehicle) := newPosition
act2 connectionsToTraverse := connectionsToTraverse $\setminus$ {vehicle$\mapsto$p}
END
crossHub $\widehat{=}$
ANY
vehicle , hub
WHERE
grd1 vehicle $\in$ Vehicles
grd2 hub $\in$ Hubs
grd3 position (vehicle) $\in$ obsHubLocations(hub)
grd4 vehicle $\mapsto$ hub $\in$ hubsToCross
THEN
act1 hubsToCross := hubsToCross $\setminus$ {vehicle$\mapsto$hub}
END
startTravel $\widehat{=}$
ANY
vehicle , r
WHERE
grd1 vehicle $\in$ Vehicles
grd2 r $\in$ routes
grd3 position (vehicle) $\in$ obsHubLocations(connectionOrigin(r(1)))
grd4 vehicle $\notin$ dom(connectionsToTraverse)
grd5 vehicle $\notin$ dom(hubsToCross)
THEN
act1 hubsToCross := hubsToCross $\cup$ {i . i $\in$ 1..card(r) | vehicle $\mapsto$ connectionOrigin(r(i))}
act2 connectionsToTraverse := connectionsToTraverse $\cup$ {p . p $\in$ ran(r) | vehicle $\mapsto$ p}
END
END

Figure 5.6: Machine `Movement3`

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

To facilitate the specification of these events a new invariant is specified which is called `hubLoad`. `hubLoad` indicates the current number of vehicles on a hub. It is specified that each hub is capable of hosting, simultaneously, a certain number of vehicles which is given by constant `hubCapacity`. The invariant specified with the help of `hubLoad` guarantees that a vehicle would not enter into a hub, if the hub is already full. A state marker `vehicleState` is also initialized at this level to track and control the movement of vehicles on and off hubs. This marker reflects the current state of a vehicle on a hub, such as `entering`, `onHub`, `leaving`, etc.

The context `Net4`, at this level, defines the notion of `hubCapacity` for hubs and junctions. The capacity of a junction is always 1, i.e., only one vehicle can be present on a junction at any given time, and the capacity of a hub can be initialized with any natural number which corresponds to the capability of the hub to host vehicles. A set `States` is also defined in this context. This set is composed of states which are used in the model as state markers for vehicles, i.e., `vehicleState`. Whereas, the context `StartState4` initializes the load of the hubs according to the vehicles which are currently present on them.

The event `enterHub` is triggered when the state of a vehicle allows it to enter into a particular hub whose `hubLoad` is lower than its capacity and it is not previously traversed by the vehicle. Once the event is triggered, the load of the hub is increased by 1 and the state of the vehicle is marked as `onHub`. A vehicle must not enter a hub if the load of the hub is equal to its capacity. In this scenario, it must wait for its turn. This case is modeled with the help of the event `wait`.

In the transportation domain, situations like gridlocks, i.e., traffic jams, occur often which may prevent all vehicles from moving. Since we are modeling the domain, so we allow them in the specification. The aim of introducing `wait` is to let vehicles wait in such situations, in order to avoid collision which may occur if they proceed in these situations.

The event `leaveHub` triggers when a vehicle leaves a hub which is already crossed by it. Upon the triggering of this event, the load of the hub is decreased by 1 and the state of the vehicle for this hub is marked as `crossed`.

Figure 5.7 shows the invariants and the newly introduced events of machine `Movement4`.

5.3.6 Fifth refinement

This refinement introduces the notion of time into the model. This is a small refinement step and in this refinement we do not add much details to the model. In this machine, we introduce three new variables: `time`, `travelTime` and `startTime`. The variable `time` represents the actual time in the model. The variable `startTime` shows the starting time of the vehicle and is noted when a vehicle starts its travel. The variable `travelTime` is noted when a travel is complete. It is calculated as the difference between the actual time of the clock and the starting time of the vehicle.

MACHINE
Movement4
...
VARIABLES
hubLoad, vehicleState, connectionsToTraverse, hubsToCross, position, location
INVARIANT
inv1 hubload $\in$ Hubs $\rightarrow$ N
inv2 $\forall$h.h $\in$ Hubs$\Rightarrow$hubLoad(h) $\leq$ hubCapacity(h)
inv3 vehicleState $\in$ Vehicles X Hubs $\rightarrow$ States
inv4 $\forall$v.card({h. vehicleState (v$\mapsto$h) = onHub $\vee$ vehicleState(v $\mapsto$ h)= leaving | h}) $\leq$ 1
inv5 $\forall$v,h. vehicleState (v$\mapsto$h)=onHub $\Rightarrow$ position(v) $\in$ obsHubLocations(h)
inv6 $\forall$v,h. vehicleState (v$\mapsto$h) = leaving $\Rightarrow$ position(v) $\in$ obsHubLocations(h)
EVENTS
...
enterHub $\widehat{=}$
ANY
vehicle, hub
WHERE
grd1 vehicle $\in$ Vehicles
grd2 hub $\in$ Hubs
grd3 position (vehicle) = obsHubLocations(hub)
grd4 hubLoad(hub)<hubCapacity(hub)
grd5 vehicleState (vehicle $\mapsto$ hub) = entering
THEN
act1 position (vehicle) := position(vehicle)
act2 hubLoad(hub) := hubLoad(hub) + 1
act3 vehicleState (vehicle $\mapsto$ hub) := onHub
END
leaveHub $\widehat{=}$
ANY
vehicle, hub, r
WHERE
grd1 r $\in$ routes
grd2 vehicle $\in$ Vehicles
grd3 hub $\in$ Hubs
grd4 position (vehicle) $\in$ obsHubLocations(hub)
grd5 vehicle$\mapsto$hub $\notin$ hubsToCross
grd6 vehicleState (vehicle$\mapsto$hub) = leaving
grd7 hubLoad(hub)$\geq$1
THEN
act1 hubLoad(hub) := hubLoad(hub)$-$1
act2 vehicleState (vehicle$\mapsto$hub) := crossed
END
wait $\widehat{=}$
ANY
vehicle, hub
WHERE
grd1 vehicle $\in$ Vehicles
grd2 hub $\in$ Hubs
grd3 position (vehicle) $\in$ obsHubLocations(hub)
grd4 hubLoad(hub) $\geq$ hubCapacity(hub)
grd5 vehicleState (vehicle$\mapsto$hub) = entering
THEN
act1 position (vehicle) := position(vehicle)
act2 vehicleState (vehicle$\mapsto$hub) := entering
END
END

Figure 5.7: Machine `Movement4`

In order to do so, event `travel` is refined as shown in figure 5.8. A global clock is introduced into the model with the event `ticTac` which is shown in figure 5.9.

```
  travel  ≙
REFINES
  travel
ANY
  vehicle , newLocation, r , origin , destination
WHERE
  grd01 r∈routes
  grd02 vehicle ∈Vehicles
  grd03 origin ∈stations
  grd04 destination ∈stations
  grd05 origin ≠destination
  grd06 origin =connectionOrigin(r(1))
  grd07 destination =connectionDestination(r(card(r)))
  grd08 location ( vehicle )∈obsHubLocations(origin)
  grd09 newLocation∈obsHubLocations(destination)
  grd10 newLocation≠location(vehicle)
  grd11 position ( vehicle )=newLocation
  grd12 vehicleState ( vehicle ↦destination)=onHub
  grd13 time ≥ startTime(vehicle)
THEN
  act1 location ( vehicle ):=newLocation
  act2 travelTime( vehicle ):= time−startTime(vehicle)
END
```

Figure 5.8: Event `travel`

```
  ticTac ≙
ANY
  t
WHERE
  grd1 t ∈ N
  grd2 t > time
THEN
  act1 time := t
END
```

Figure 5.9: Event `ticTac`

5.3.7 Sixth refinement

This refinement introduces the fourth level of observation into the model. It specifies the protocol to traverse a path. Event `traversePath` is decomposed as follows:

$$traversePath \equiv (waitToEnterOnPath*;\ leaveHub;$$

$$(waitToMoveOnPath|\ moveOnPath)*)$$

In order to traverse a path, we must avoid the risk of collision between vehicles. In order to do so, we introduce two new variables: `vehiclePath` and `vehiclePosition`. The first denotes the path on which a vehicle is currently running and the latter refers to the position the vehicle is currently occupying on the path. Then, with the help of these two variables, we define our non-collision property. The intention is that two vehicles must not occupy the same position on the same path at the same time. To ensure this, we introduce the notion of critical distance among vehicles which is the idea that a vehicle must maintain some safety distance, `criticalDistance`, with the vehicle in front of it. The following invariants are introduced:

$$
\begin{gathered}
vehiclePath \in Vehicles \pfun paths \\
vehiclePosition \in Vehicles \pfun \mathbb{N}_1 \\
dom(vehiclePath) = dom(vehiclePosition) \\
\forall v1, v2.v1 \in Vehicles \land v2 \in Vehicles \land v1 \neq v2 \land \\
v1 \in dom(vehiclePosition) \land v2 \in dom(vehiclePosition) \land \\
vehiclePath(v1) = vehiclePath(v2) \Rightarrow \\
vehiclePosition(v1) \neq vehiclePosition(v2)
\end{gathered}
$$

As the protocol for traversing a path has been enriched, so the following new events are added to the specification:

- Event `waitToEnterOnPath`, which is similar to event `wait`. The difference is that event `wait` allows a vehicle to wait if the hub is full, whereas event `waitToEnterOnPath` allows a vehicle to wait on a hub if the distance between this vehicle and the vehicles already present on the path does not fulfill the safety requirement. This safety condition is given by `grd11` of this event. The event `waitToEnterOnPath` is shown in figure 5.10.
- Event `moveOnPath`, which is triggered when the vehicle is in the position to move on a path safely, i.e., the distance between this vehicle and its predecessor is always more than the allowed safety distance. The vehicle continues moving until it is stopped by safety concerns or the end of the path, `pathLen`, is reached. A local variable `move` is added to its position at the end of the event which shows its current position on the path. The event `moveOnPath` is shown in figure 5.11.
- Event `waitToMoveOnPath`, which allows a vehicle to wait on a path if its distance from its predecessor is less than the safety distance. In this case, the vehicle waits until there is room on the path for a safe movement. The event `waitToMoveOnPath` is shown in figure 5.12.

The event `leaveHub` also goes under refinement at this level. The property we check, like all other newly introduced events of this refinement, is that when a vehicle leaves a hub there should be enough distance between this vehicle and all other vehicles currently present on the path. If this vehicle has no predecessor then it can choose its position arbitrarily. This is shown by figure 5.13.

5.3.8 Seventh refinement

The rationale for this refinement is to properly integrate the timing property into the model. In refinement 5, we introduced the notion of time. At that level, we could not fully express this property because the protocol of traversing a path was not fully described. In the previous refinement, we defined this behavior and now we are able to fully express the timing property.

In order to integrate the notion of time with events, we modeled the technique used in simulating queue systems. We introduced a timed event queue,

waitToEnterOnPath $\widehat{=}$
ANY
vehicle , path, route, vehiclesOnPath
WHERE
grd01 route $\in$ routes
grd02 vehicle $\in$ Vehicles
grd03 path$\in$paths $\wedge$ path$\in$ran(route)
grd04 position (vehicle) $\in$ obsHubLocations(connectionOrigin(path))
grd05 vehicle$\notin$dom(vehiclePath)
grd06 vehicle$\mapsto$path $\in$ connectionsToTraverse
grd07 vehicle$\mapsto$connectionOrigin(path) $\notin$ hubsToCross
grd08 vehicleState (vehicle$\mapsto$connectionOrigin(path))=leaving
grd09 vehiclesOnPath $\subseteq$ Vehicles
grd10 vehiclesOnPath={v.v$\in$Vehicles $\wedge$ v$\in$dom(vehiclePosition) $\wedge$ v$\in$dom(vehiclePath) $\wedge$ vehiclePath(v)=path|v}
grd11 $\neg$($\forall$v.v$\in$ vehiclesOnPath $\Rightarrow$ vehiclePosition(v) $>$ criticalDistance)
THEN
SKIP
END

Figure 5.10: Event `waitToEnterOnPath`

moveOnPath $\widehat{=}$
ANY
vehicle , path, vehiclesOnPath, move
WHERE
grd1 vehicle $\in$ Vehicles
grd2 hub $\in$ Hubs
grd3 vehicle $\in$ dom(vehiclePath) $\wedge$ vehiclePath(vehicle) = path
grd4 vehiclePosition (vehicle)<pathLen(path)
grd5 vehiclesOnPath $\subseteq$ Vehicles
grd6 vehiclesOnPath={v.v$\in$Vehicles $\wedge$ v$\in$dom(vehiclePath) $\wedge$ vehiclePath(v)=path|v}
grd7 $\forall$v . v$\in$vehiclesOnPath $\wedge$ vehiclePosition (v) $>$ vehiclePosition (vehicle) $\Rightarrow$ vehiclePosition (v) $-$ vehiclePosition (vehicle) $>$ criticalDistance
grd8 {v.v$\in$vehiclesOnPath $\wedge$ vehiclePosition (v) $>$ vehiclePosition (vehicle) | v} $\neq \varnothing \Rightarrow$ move$\in$1..((min({v.v$\in$vehiclesOnPath $\wedge$ vehiclePosition (v)$>$vehiclePosition (vehicle)| vehiclePosition (v)})$-$ vehiclePosition (vehicle))$-$ criticalDistance)
grd9 {v.v$\in$vehiclesOnPath $\wedge$ vehiclePosition (v) $>$ vehiclePosition (vehicle) | v} $= \varnothing \Rightarrow$ move$\in$1..(pathLen(path)$-$vehiclePosition(vehicle))
THEN
act1 vehiclePosition (vehicle) := vehiclePosition(vehicle) + move
END

Figure 5.11: Event `moveOnPath`

```
  waitToMoveOnPath ≙
ANY
  vehicle , path, vehiclesOnPath
WHERE
  grd1 vehicle ∈ Vehicles
  grd2 path ∈ paths
  grd3 vehicle ∈dom(vehiclePath) ∧ vehiclePath( vehicle ) = path
  grd4 vehiclesOnPath ⊆ Vehicles
  grd5 vehiclesOnPath={v.v∈Vehicles ∧ v∈dom(vehiclePath) ∧ v∈dom(vehiclePosition) ∧
       vehiclePath (v)=path|v}
  grd6 ∃v.v∈vehiclesOnPath∧v≠vehicle ∧ vehiclePosition (v)> vehiclePosition ( vehicle ) ∧
       vehiclePosition (v) − vehiclePosition ( vehicle )≤ criticalDistance
THEN
  SKIP
END
```

Figure 5.12: Event `waitToMoveOnPath`

```
  leaveHub ≙
REFINES
  leaveHub
ANY
  vehicle , hub, r , p, vehiclesOnPath
WHERE
  grd01 r∈routes
  grd02 vehicle ∈Vehicles
  grd03 hub∈Hubs
  grd04 position ( vehicle )∈obsHubLocations(hub)
  grd05 vehicle ↦hub∉hubsToCross
  grd06 vehicleState ( vehicle ↦hub)=leaving
  grd07 hubLoad(hub)≥1
  grd08 p∈paths ∧ p∈ran(r)
  grd09 hub=connectionOrigin(p)
  grd10 vehicle ↦p∈connectionsToTraverse
  grd11 vehiclesOnPath⊆Vehicles
  grd12 vehiclesOnPath={v.v∈Vehicles ∧ v∈dom(vehiclePath) ∧ vehiclePath(v)=p|v}
  grd13 ∀v.v∈vehiclesOnPath⇒ vehiclePosition (v) > criticalDistance
THEN
  act1 vehicleState ( vehicle ↦hub):=crossed
  act2 hubLoad(hub):=hubLoad(hub)−1
  act3 vehiclePath ( vehicle ):=p
  act4 vehiclePosition ∈|(∃pos. pos∈1..pathLen(p)∧vehiclePosition ' = vehiclePosition ⩤
       {vehicle ↦pos} ∧ (∀ v . v ∈ vehiclesOnPath ∧ v ≠vehicle ⇒
       vehiclePosition '(v) − vehiclePosition '( vehicle ) ≥ criticalDistance ))
END
```

Figure 5.13: Event `leaveHub`

`activationTime`, which contains the time at which a moving vehicle must perform an event. The following invariants are introduced:

$$activationTime \in Vehicles \nrightarrow \mathbb{N}$$
$$activationTime \neq \emptyset \Rightarrow time \leq min(ran(activationTime))$$

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

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

The action part of the event modifies the event queue accordingly. The refined timing pattern, specified by the event `ticTac`, is shown in figure 5.15. An example of use of this pattern is shown in figure 5.14.

```
  enterHub ≙
REFINES
  enterHub
ANY
  vehicle , hub
WHERE
  grd1 vehicle ∈ Vehicles
  grd2 hub ∈ Hubs
  grd3 position ( vehicle ) ∈ obsHubLocations(hub)
  grd4 hubLoad(hub) < hubCapacity(hub)
  grd5 vehicleState ( vehicle ↦ hub) = entering
  grd6 vehicle ∈ dom(activationTime) ∧
       time=activationTime(vehicle)
THEN
  act1 position ( vehicle ) := position( vehicle )
  act2 hubLoad(hub) := hubLoad(hub) + 1
  act3 vehicleState ( vehicle ↦ hub) := onHub
  act4 vehiclePath := {vehicle} ⩤ vehiclePath
  act5 vehiclePosition := {vehicle} ⩤ vehiclePosition
  act6 activationTime := activationTime ⩤
       {vehicle ↦ time + hubCrossingTime(hub)}
END
```

Figure 5.14: Event `enterHub`

```
  ticTac ≙
ANY
  tic
WHERE
  grd1 activationTime ≠ ∅
  grd2 tic = min(ran(activationTime))
  grd3 tic > time
THEN
  time := tic
END
```

Figure 5.15: Event `ticTac`

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

Earlier in the specification, we allowed gridlock situations to happen. Consequently, this decision, combined with the introduction of time, raised up some unexpected proof-obligations. Since our model is based on timed event queues, i.e., happening of one event allows the next, so blocking one vehicle would imply the whole system to come to a halt. In order to avoid such cases, we introduce three new events into the model. The description of these events is as follows:

- Event `lockOut` is triggered 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." Event `lockOut` is shown in figure 5.16.

```
  lockOut ≙
REFINES
  wait
ANY
  vehicle , path, vehiclesOnPath
WHERE
  grd1 vehicle ∈ Vehicles
  grd2 hub ∈ Hubs
  grd3 position ( vehicle )∈obsHubLocations(hub)
  grd4 hubLoad(hub)≥hubCapacity(hub)
  grd5 vehicleState ( vehicle ↦hub)=entering
  grd6 vehiclesOnHub = {v|v∈Vehicles∧position(v)∈obsHubLocations(hub) ∧
       ( vehicleState (v↦hub)=leaving ∩
        vehicleState (v↦hub)=onHub)} ◁ activationTime
  grd7 vehiclesOnHub ∅
  grd8 vehicle ∈ dom(activationTime) ∧ time=activationTime(vehicle)
THEN
  act1 position ( vehicle ) := position( vehicle )
  act2 vehicleState ( vehicle ↦ hub) := entering
  act3 activationTime := {vehicle} ⩤ activationTime
  act4 blockedVehicles := blockedVehicles ∪ { vehicle }
END
```

Figure 5.16: Event `lockOut`

- Event `lockIn` is used when a vehicle needs to enter a path which is full of other (stationary) vehicles. This vehicle is then "locked in." Event `lockIn` is shown in figure 5.17.

- Event `lockOnPath` is used when a vehicle is blocked on a path after havin begun the traversal of the path. It is then "locked on path." Event `lockOnPath` is shown in figure 5.18.

Such vehicles are then added to the list of blocked vehicles and subsequently removed from `activationTime`. The flow of the system is then easily controlled.

5.4 Hierarchy of the model

Figure 5.19 presents the Event-B hierarchy of our domain model. It contains all the contexts and machines specified in the model. Extension between contexts and refinements between machines are shown by single arrow lines, whereas use of contexts by machines is depicted by double arrow lines. An important point to notice about this hierarchy is its three column format and the indication of observation levels.

The first column of the hierarchy is constituted of a series of refinements of the context `Net`. This series of refinements has been used to model the general properties of a transportation network. The second column depicts the refinements of `StartState` contexts. These contexts contain the starting values of variables used in the machines and hence are deployed for initialization purposes. The last column

lockIn $\widehat{=}$
REFINES
waitToEnterOnPath
ANY
vehicle , path, route, vehiclesOnPath
WHERE
grd01 route $\in$ routes
grd02 vehicle $\in$ Vehicles
grd03 path $\in$ paths $\wedge$ path$\in$ran(route)
grd04 position (vehicle) $\in$ obsHubLocations(connectionOrigin(path))
grd05 vehicle $\notin$dom(vehiclePath)
grd06 vehicle$\mapsto$path $\in$ connectionsToTraverse
grd07 vehicle$\mapsto$connectionOrigin(path) $\notin$ hubsToCross
grd08 vehicleState (vehicle$\mapsto$connectionOrigin(path))=leaving
grd09 vehiclesOnPath={v.v$\in$Vehicles $\wedge$ v$\in$dom(vehiclePath) $\wedge$ vehiclePath(v)=path|v}
grd10 $\neg$ ($\forall$v. v$\in$ vehiclesOnPath $\Rightarrow$ vehiclePosition (v) $>$ criticalDistance)
grd11 vehiclesOnPath $\lhd$ activationTime = $\varnothing$
grd12 vehicle $\in$ dom(activationTime) $\wedge$ time = activationTime(vehicle)
THEN
act1 blockedVehicles := blockedVehicles $\cup$ { vehicle }
act2 activationTime := {vehicle} $\lhd\!\!+$ activationTime
END

Figure 5.17: Event `lockIn`

lockOnPath $\widehat{=}$
REFINES
waitToMoveOnPath
ANY
vehicle , path, vehiclesOnPath
WHERE
grd1 vehicle $\in$ Vehicles
grd2 path $\in$ paths
grd3 vehicle $\in$dom(vehiclePath) $\wedge$ vehiclePath(vehicle) = path
grd4 vehiclesOnPath={v.v$\in$Vehicles $\wedge$ v$\in$dom(vehiclePath) $\wedge$ v$\in$dom(vehiclePosition) $\wedge$ vehiclePath (v)=path|v}
grd5 $\exists$v.v$\in$vehiclesOnPath$\wedge$v$\neq$vehicle $\wedge$ vehiclePosition (v)$>$ vehiclePosition (vehicle) $\wedge$ vehiclePosition (v) $-$ vehiclePosition (vehicle)$\leq$ criticalDistance
grd6 vehiclesOnPath $\lhd$ activationTime = $\varnothing$
grd7 vehicle $\in$ dom(activationTime) $\wedge$ time=activationTime(vehicle)
THEN
act1 blockedVehicles := blockedVehicles $\cup$ { vehicle }
act2 activationTime := {vehicle} $\lhd\!\!+$ activationTime
END

Figure 5.18: Event `lockOnPath`

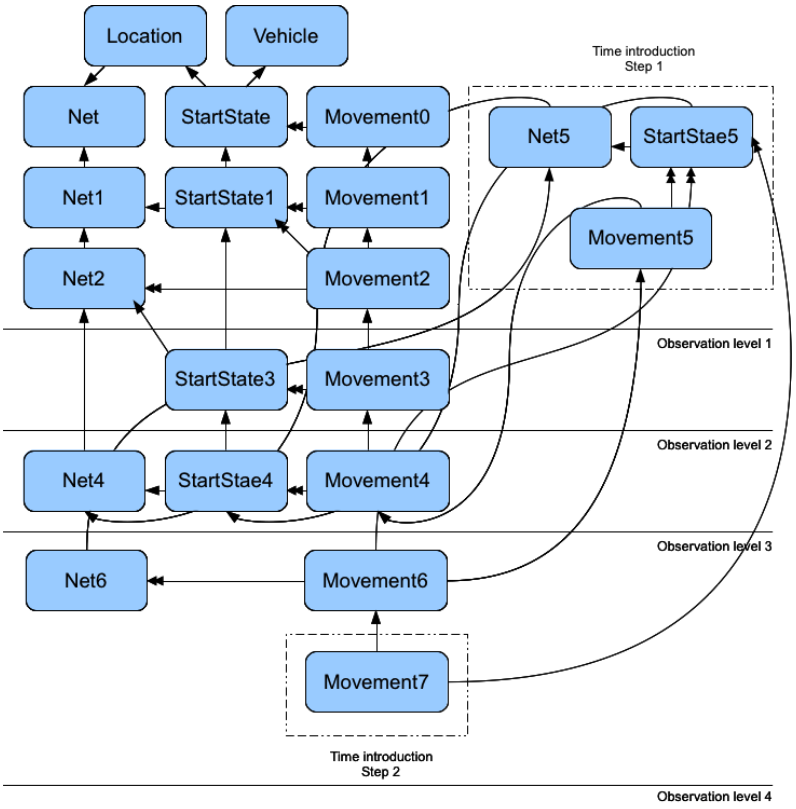

Figure 5.19: The hierarchy of the domain model

of the hierarchy shows the series of refinements of machines. These machines build the main construct of the land transportation domain model.

Figure 5.19 also indicates four different levels of observations. The abstract model, the first two refinements and the fifth refinement belong to the first level of observation. Though technically realized as the refinement of `Movement4`, the fifth refinement step is logically situated at the first level of observation; it introduces time and concerns only the events at the first level. The third and fourth refinement machines belongs to the second and third level of observations respectively. The sixth and seventh refinements model the fourth level of observation. Machine `Movement7` completes the introduction of time into the model and concerns the events at this level, hence its situation.

5.5 Verification of the model

We have given a great deal of attention to the verification of our model. Consequently, we now have an Event-B specification of transport domain in which all proof-obligations have been discharged. In the following table, we present some statistics about the proofs of our model.

	Total	Automatic	Interactive
Location	1	1	0
Vehicle	0	0	0
Net	6	5	1
*Net*1	4	4	0
*Net*2	4	1	3
*Net*4	2	0	2
*Net*5	0	0	0
*Net*6	0	0	0
StartState	0	0	0
*StartState*1	1	0	1
*StartState*3	0	0	0
*StartState*4	3	3	0
*StartState*5	1	0	1
*Movement*0	3	1	2
*Movement*1	3	1	2
*Movement*2	2	2	0
*Movement*3	18	4	14
*Movement*4	66	8	58
*Movement*5	8	4	4
*Movement*6	54	15	39
*Movement*7	74	22	52

Naturally, the number of proof-obligations increases when more details are added to models by refinement steps. Though, as shown by the table, most of the proofs were discharged interactively, yet most of them were not difficult to prove. In fact, many of the proofs just required either one or a few clicks. Though, not a real annoyance, this practice soon become distracting and boring. We have used Rodin 0.9 for specification and verification of the model, as it was the latest version at the time when we started development of the model. We have observed that provers of Rodin 2.x are much more efficient concerning the verification of the specification.

The real difficult proofs are associated to the notion of state markers. In machine `Movement4`, we introduced `vehicleState` to track and control the movement of vehicles on and off hubs. Invariant 4 of this machine states that a vehicle can only be present on one hub at a time. In all concerned events of this machine, for example, `enterHub`, `leaveHub`, `crossHub`, we had to prove that vehicle is not currently present on any other hub. Due to these intricate proofs, the number of interactively discharged proof-obligations in this machine are significantly higher than any other machine of the model.

5.6 Summary

Understanding the application domain is important for specification of useful system requirements. When engineering domains formally, all the vocabulary and key concepts of the problem domain is precisely specified. This, then, can be effectively used to verify and validate the understanding of the problem domain among various stakeholders.

In this chapter, we have specified the intrinsic laws and behaviors of transport domains in general, and properties of collision avoidance and timing in particular. We have developed our model in the formal specification language Event-B. We have liberally used refinements, both of machines and contexts. We have also presented the idea of "observation levels" to group refinements in such a way that facilitates the introduction of new properties into the model. We have also given a great amount of attention to proofs and all refinement steps are fully verified.

Chapter 6

Guidelines for domain engineering with Event-B

Contents

6.1 Introduction

In the last chapter, we presented a consistent, effective and verified formal model of land transport domain. During this development, we were confronted with several challenging characteristics of the domain, such as

- there is no centralized control-system. On the contrary, each vehicle is autonomous and the communication between vehicles is minimal,
- there are important functionalities that must be expressed at the system-level: absence of collisions, gridlock, etc.,
- time is an important concept at several levels.

Among the issues which were raised, the problem of different "flavors" of time is very interesting. The flavor of time used by kinematics laws is continuous, the flavor used to measure travel-time is discrete; when discretized, the former flavor requires very small units, while the other requires much bigger units.

As a complex and reasonably well-known domain, land transportation is a good candidate to test and assess specification methods and processes. We have conducted a thorough analysis of the use of Event-B for this case-study, and more generally, for specifying domains. The results of this study are presented in this chapter which mainly deals with three questions: What to specify, how to refine, and how to verify using Event-B.

The chapter is organized as follows: Section 6.2 presents an overview of the ingredients of a domain model which must be formally specified. Section 6.3 and 6.4 discuss how it should be refined and verified respectively. We present some observations about the language and the tool in section 6.5. The chapter is finally concluded in section 6.6.

6.2 What to specify?

6.2.1 Model assumptions

One of the main reasons to use mathematical formalisms and tools is to define explicitly 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 [Zave 1997a].

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 then be 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. Please note that this classification is purely methodological, just for the ease of specification.

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) or that hubs are

partitioned into stations and junctions. 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 them 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.

6.2.2 Define protocols

A domain exhibits several protocols. Once events are decomposed into smaller events, it is crucial that these events be fired in a strict order so that the protocol can be properly followed. 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 implicit controls to express this protocol. Instead, we must make an 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 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 the vehicle's current 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 through 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) [Jackson 1983] or formalism like CSP could be used.

6.2.3 Specify time

Unsurprisingly, modeling time raised many questions. We used the timing patterns for Event-B proposed by Cansell et al [Cansell 2007] 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:

- pick an event to "time"
- add the guard $\mathsf{vehicle} \in \mathsf{dom}(\mathsf{activationTime}) \land \mathsf{time} = \mathsf{activationTime}(\mathsf{vehicle})$
- add the action $\mathsf{activationTime} := \mathsf{activationTime} \lhd\!\!\!\!- \{\mathsf{vehicle} \mapsto \mathsf{time} + \mathsf{timeInc}\}$. $timeInc$ is, of course, dependent on the particular event. It can be a constant, an arbitrary value or a computation on the event queue.

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 introduction of abstract kinematics behavior did not pose many problems. The basic idea was to use the pattern presented above with a kind of "fixed tick." This idea answers the second question.

6.2.4 Express temporal properties

Temporal properties, such as safety and liveness, are essential to most domains and systems, particularly those which are safety-critical. However, they do not play the same role in their specifications.

Safety

A safety property asserts that nothing bad happens [Lamport 1977]. 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 on a certain part of the network 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 [Leuschel 2003].

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 proofs, technically preventing their discharge. We have tackled the situation by modeling special `(lockXXX)` events. These events clearly show the conditions of the blockage. 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 `(lockXXX)` 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" [Lamport 1977]. 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 [Yadav 2009], 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 the abstract level it is enabled at the 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 desti-

nations. 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 these kind of properties. We consider this as an important shortcoming.

6.3 How to refine?

Refinements and observation levels are distinct concepts. Refinement is the cornerstone of the B method. It serves two purposes: methodologically, it allows specifiers to concretize the specification, and technically, it induces proof obligations which guarantee the correctness of the development. It gives the development a flat structure which may impair its readability.

When modeling a domain, we prefer to see refinements as process steps where a new piece of information is added to the model. We find useful to classify refinements in two categories:

1. State enrichment: a new concept or a new constraint on the state is added. Contexts are extended and the events concerned by the novelty are refined (guards and actions are changed). The structure of the specification is not changed.

2. Event decomposition: a "large" event is decomposed into several "smaller" ones. The structure of the model is altered.

The second kind of refinement corresponds to a change of "observation level." 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 that level.

6.3.1 Refine slowly

We advise to use small incremental steps while developing domain models. Ideally, only one new fact should be introduced per refinement.

Recording rationales for refinements is essential for later understanding of the formal text. Currently, Rodin provides only minimal abilities in this domain: simple comments. They are not well fitted for long explanations. We make use of this small-step approach to introduce the notion of transportation network, for instance.

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

typ obsNetConnections ∈ Connections → Nets
pro ∀ c . c ∈ Connections ⇒ card(obsNetConnections[{c}] = 1

or as

typ obsNetConnections ∈ Connections ⤔ Nets // total injection

The assumption is less conspicuous in the second expression.

Domain models are reference documents. So, they will often be read by people who need to check some intuitive assumptions 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.

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, understanding the protocols which prevent collisions require to consider the guards of two different sets of events. Expressing each type of collision by one specific refinement eases the work.

Last, expressing only one feature of the domain at a time helped us finding patterns, such as the time introduction presented earlier, in the refinements of events. Again, this kind of regularity in the expression makes later analysis of the text easier.

We should also note that small refinement steps are well supported by Rodin. They do not cost much.

6.3.2 Refine unconventionally

Another piece of advice is to use the notion of observation levels to organize the development rather than the implicit linear view of refinements.

Event-B has inherited from B the view that a development is a sequence of refinements. This conception is adequate in B, less so in Event-B. Plugins like Feature Composition Plugin [Gondal 2009], Parallel Composition Plugin [Poppleton 2008], or Shared Event Composition Plugin [Silva 2010] attest the importance of this observation and may help domain modelers in the future. However, their slow progress seems to indicate that strong formal difficulties may restrict their applicability.

Organizing the introduction of a new feature along the observation levels (at least, one refinement per level) has several advantages:

- we make "small" steps, focusing on a small and specific set of events,
- we can relate more easily failures during the proofs to incompatibilities between the feature's definition and the existing model,
- the levels point "naturally" to the feature's facets we need to analyse.

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 6.1. 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
```

Figure 6.1: 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 6.2 and to introduce in all other events a dummy action of the form:

meter $\in|$ 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
```

Figure 6.2: Introduction of Energy consumption: what we have

6.4 How to verify?

The verification of a domain amounts to assert that the specified facts, laws and properties about the domain are consistent, checkable and provable. A verified domain model is, therefore, considered as a consistent set of assumptions about the domain. In fact, this hypothesis does make sense as an unprovable model, of course, can not be trusted.

Refinements serve different purposes in system development than in domain modeling. In the former case, the refinement is a more concrete description of the same model. We need to prove that the new description enjoys the same functional properties. In the latter case, the refinement is an enrichment of the model. We need to show that the new feature is consistent with the previous ones. Although based on the same set of proof-obligations, the proving process needs to be observed from another point of view.

6.4.1 Beware of easy proofs!

Asserting the consistency of an assumption expressed as an invariant is relatively easy. Either the invariant-related proof-obligations can be discharged or not. This is safe.

However, when an assumption is expressed as an axiom, it is hard to prove its consistency. Proof obligations assert well-formedness and well-typing, but not consistency. We are then always at a risk to introduce a fact which is in contradiction with the rest of the model. Although hard, the ability to detect contradiction among axioms is crucial for the correctness of the model.

Unfortunately, Rodin does not warn us when axioms are inconsistent. One should always keep a close eye on the proofs. If proofs become mysteriously easy to discharge, beware! This point is further elaborated in the section 6.5 as an observation on the tool.

6.4.2 Beware of obvious truth!

Sometimes, discharging a proof-obligation may not be possible. A proof which can not be carried out in Event-B is not always an indication of an error in the specification. We can make "formal approximations" in such situations. Rodin allows this by declaring the goal as "reviewed." Model is then considered as trustworthy.

While legitimate in certain situations, in particular due to shortcomings of current provers, reviewing an "obvious" goal may lead to a surprise. For instance, assuming $x(y/z) = (xy)/z$ seems natural, except that this is true in $\mathbb{R}$, but not in $\mathbb{N}$. Though the difference becomes actually negligible when numerators are much bigger than denominators, yet this approximation is formally incorrect.

One should always be careful while making approximations and should at least test them. Dynamic testing techniques, such as animation, with realistic values, can give insight on the validity of the approximations and on the solidity of the model.

6.4.3 Use animation to complement provers

Animation allows specifiers to check the behavior of the specification by observing its execution. Non-provable safety and liveness properties can be assessed by analyzing state values and event-enabledness status. We can make solid observations on the behavior of a model by a trace analysis of the simulated scenarios.

Animation cannot show that a property always hold, but it may help to generate counter examples which show that the model is partly incorrect. Animation plays the same role as model-checking, in fact, a tool like ProB offers both functions.

A very useful side effect of animation is to help get better insights on the model. Implicit properties and unexpected behaviors, either good or bad, will become apparent. We did even use it as a validation and prototyping tool to understand and fix the expression of tricky protocols. The next section of this thesis is dedicated to animation where it is discussed in more details.

6.5 Observations on tool & language

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 6.3), 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.

6.6 Summary

The domain of transportation exhibits several interesting features, such as high levels of non-determinism, complex interactions, stringent safety properties, multifaceted timing attributes, etc. The formal representation of these features with Event-B has been a challenging task.

6.6. Summary

In this chapter, based on our experience, we have critically analyzed the capability of Event-B as a domain engineering tool. We have identified the areas where modelers can struggle while specifying domains and presented some guidelines to avoid such pitfalls. In this context, we have answered three pertinent questions: what to specify? how to refine? and how to verify? Apart from the areas such as temporal properties, where more work is still needed, we have found Event-B as a mature and effective tool for engineering domains.

Part III

VALIDATION

Chapter 7

Validation of specifications by animation

Contents

7.1 Introduction

The intrinsic complexity of the transport domain raised quickly the issue of the validation of our model. "Is the model close enough to the reality it pretends to represent?" is an instance of the general quality assurance question "Do we build the right product?" Proofs and formal treatments are insufficient. We have to look towards validation techniques for answers.

Successful validation strategies are often based on users observing and playing with prototypes. A similar strategy, such as animation, can be used in a formal development but with a constraint: the prototype must be rigorously derived form the formal specification. Some tools are provided to animate Event-B specifications, Brama [Servat 2006], for instance. However, these tools put drastic constraints on the class of specifications they can animate: limited non-determinism,

non-constructive definitions easily implemented by enumerations, quantifications amenable to simple iterations, etc. The catch is that a well written specification is likely to exhibit contrary features at its early development stages.

Specifications can be classified into two major categories: animatable and non-animatable. In order to achieve the animation of non-animatable specifications, we have to change their class. We achieve this goal by their transformation but with a constraint that their behavior must remain the same, possibly at the expense of other properties, such as provability.

The chapter is organized as following: In section 7.2, we discuss the validation of specifications using animation. In section 7.3, we discuss the benefit of using animation in the incremental development of specifications. In section 7.4, we introduce the animator Brama, its structure, its working principle, etc. Then, in section 7.5, we discuss the distinction between animatable and provable specifications. In section 7.6, we talk about the limitations of the animator Brama. Finally, in section 7.7, we discuss how can we obtain the animatability of specifications by changing their class. Section 7.8 then concludes the chapter.

7.2 Validation by animation

Once a model has been formally specified and verified, an important question arises: does it accurately capture the requirements? While proof tools guarantee the consistency of the specification (verification), they are of little help to check if the specification models the desired behavior (validation).

There are several ways to validate a specification: prototyping, structured walk-through, transformation into a graphical language, animation, and others. All concur to the same goal: to evaluate the specification in order to assess its conformance to the original requirements which later contribute in the demonstration that the software fulfills users' needs.

To answer the question of validation, we use the technique of animation. The main goal behind animation is to demonstrate the requirements mentioned 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 a set of possible 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 are approaching the intersection.

The behavioral scenarios, which define the functional behaviors of the system through a sequential executions of events, are animated by feeding some initial values to animators at startup. These startup values may not be required by animators which can compute possible values from the specification itself. 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 selected scenarios, which are considered critical for the validation of the specification, may be an effective approach.

7.3 Stepwise animation

Waiting for a final specification to begin the validation leads to the same difficulties as proving a program against its specification: costly, very complex, soon unmanageable. The strategy which works with the verification of specifications built by a stepwise refinement process, i.e., to break down the proof of correctness of the implementation into smaller proofs associated with each refinement step, can also work for validation. In a similar spirit, the technique of animation can be used at each refinement step to break its validation into smaller assessments. It then proves as a validation tool consistent with the refinement structure of the specification process.

We used animation while developing our specifications: we did not wait until their construction was finished. The strategy of stepwise validation of specifications which we have adopted has several advantages. One of them is the fact that problems are detected close to the point where their cause was introduced. This facilitates the understanding of the cause. Another advantage is the fact that an unforeseen behavior may be associated with a specific refinement. If we see a refinement as a formalization of an assumption, then we have an indication that some interactions between assumptions need to be investigated.

7.4 Brama: The animator

Once a model has been specified using the Event-B specification language in Rodin, Brama, an Eclipse based animation plug-in for the platform Rodin, can be exploited to execute it for its validation.

7.4.1 Working principle

In Brama, 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 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).

While animating specifications, Brama determines whether the invariant clause has been violated. If this is the case, it indicates the violated part of the invariant. It picks values and tries to evaluate the guard. If it finds good values, the guard is evaluated to TRUE, otherwise FALSE.

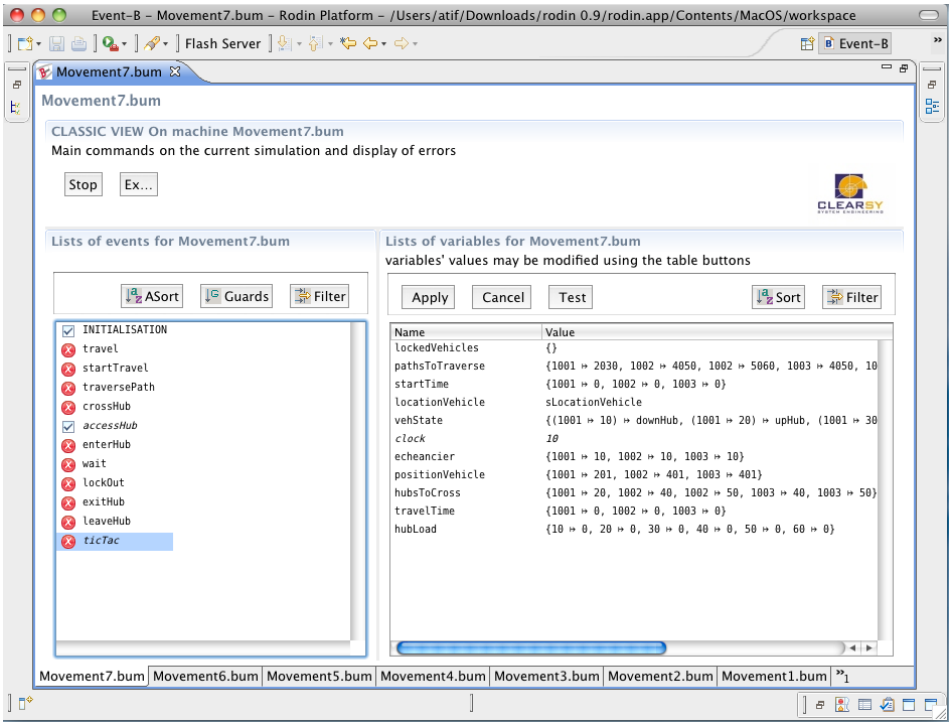

Figure 7.1: The Brama animator for RODIN

Figure 7.1 shows the "classic" interface of Brama. 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 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.

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[1] graphical animation through a communication server; it then acts as the engine which controls the graphical effects. A mechanism of observers is provided. Expressions and predicates can be individually monitored and their value is communicated to the Flash program each time they change. Last, a scheduler mechanism allows for the automatic firing of events.

7.4.2 Structure

Brama is built on an animation engine which, in fact, is a predicate solver. Other components of Brama are a visualization tool for variables and events, a management module for the automatic linking of events, a management module for variables, predicates and observed expressions, and a module to communicate with external

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

graphical environment, such as Flash.

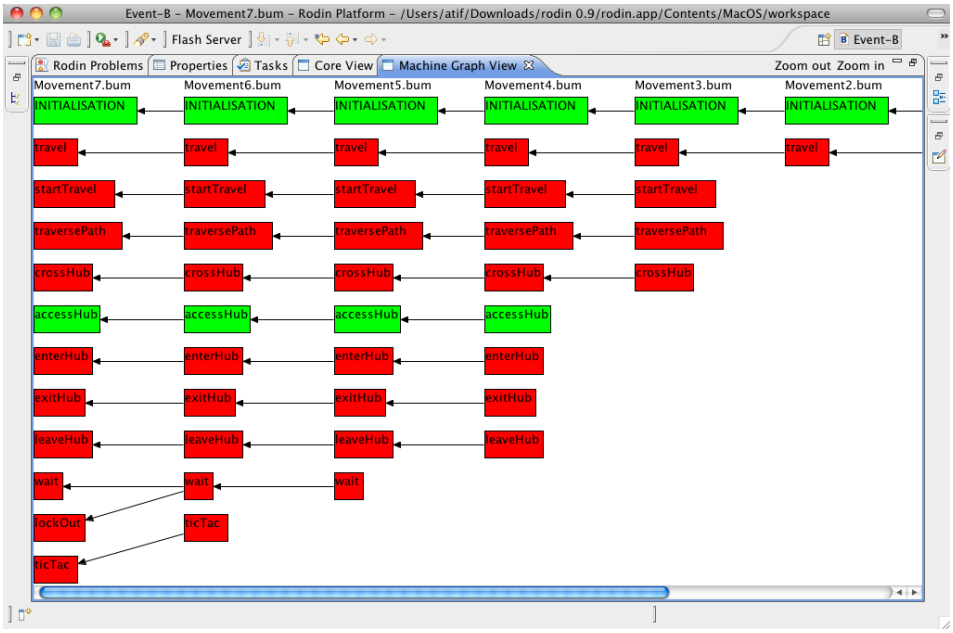

Figure 7.2: Machine-graph-view generated by Brama

The animation of a model obtained by Brama can be seen from different perspectives. While the variable-view, machine-view and event-view show the animation of the model from their respective point of views, the most interesting is the machine-graph-view. As depicted by figure 7.2, this view shows the animation of all the refinements of the model in one glance. Green boxes signal enabled events and red boxes signal disabled events. All the refinement levels are animated concurrently. However, it is worth noting that a refined event may be animatable while its abstraction may not.

7.4.3 Related animators

ProB [Leuschel 2003] and AnimB[2] are two other animators which are also capable to animate Event-B models. While there is little scientific or technical documentation available for AnimB, ProB has emerged as a mature and well-documented animator as well as model checker for the Event-B specifications.

Values to constants in Brama and AnimB explicitly need to be provided, whereas ProB does this automatically using constraint-solving techniques by finding proper values that satisfy all axioms. Pure enumeration techniques, which both Brama and AnimB employ, are limiting as compared to constraint-solving and pose some problems when evaluating complicated predicates. In fact, they were the main reason which forced us to transform our specifications in order to achieve their animation.

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

Like Brama, ProB also fails to address some of the known animation problems. For example, it also requires that all sets should be given a finite cardinality or mathematical integers cannot be enumerated outside their upper and lower bounds. Basic animation challenges for B specifications and ProB limitations are discussed in details in [Leuschel 2008].

It should be noted that our choice of tool, Brama, is contingent. At the time when we started our development, it was the only one able to animate Event-B specifications; AnimB appeared later and ProB required the translation of Event-B code into B. While our proposed heuristics (discussed in chapter 8) should surely be adapted to these specific tools, we suspect that the general philosophy of animation we have adopted is still valid. Different set of heuristics would be required for different animators.

7.5 Classes of specifications

As shown by figure 7.3, we can characterize specifications along two axes: provable and animatable. A specification may therefore fall into one of four categories:

1. non-provable and non-animatable,
2. non-provable but animatable,
3. provable but non-animatable,
4. provable and animatable.

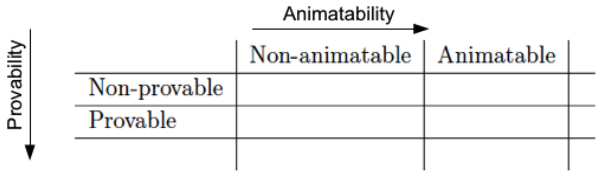

Figure 7.3: Classes of specifications

Like a "bad" program can be executed, an incorrect specification can also be animated. Of course, neither would be an admissible solution to the problem they are meant to solve, but the observation of the execution can give precious information about the correct solution. Some important ingredients of specifications, such as non-constructive definitions, infinite sets, or complex quantified logic expressions, are among the list of constructs which are non-animatable. Unfortunately, well written specifications often use these traits. Indeed, it is even advised that early specifications be highly abstract and non-constructive. Animation, by nature, heavily depends on tools. So any limitation of the tool will also be a restriction on the class of animatable specifications.

To validate a specification which does not belong to the animatable class, we need to "bring it in." We do this by applying the transformational heuristics which are designed to keep the behavior unaltered, possibly at the expense of other properties, such as provability.

While it would be interesting for theoreticians to know whether some tools' limitations come from implementation features or have a deep mathematical reason; we, as practitioners, are more interested in designing practical rules for one particular tool. However, it is important to have an explicit rule design technique so that the current effort can be leveraged and transposed to other tools.

The first error message one is likely to encounter with the animator Brama is "Brama does not support finite axioms." Since these technical axioms are mandatory to discharge the well-formedness proof obligations of carrier sets, the case was settled. Beyond the anecdote (removing such technical axioms do not change the essence of the specification), this feature of Brama gave us the essential insight to dissociate proofs and animations. We could then focus on the transformational heuristics which preserve behavior without bothering about preserving proofs.

One can wonder why we do not produce a specification at first which belongs to the animatable class. The reason is that such specifications are likely to be less readable and, more importantly, unprovable. The elements which are necessary to discharge proof obligations are sometimes altered or suppressed by elements which make specifications animatable. While specifying, verification should be given preeminence over animation. However, once we are assured of its verifiability, we can then proceed towards its animatability.

The process of bringing specifications into the animatable class may "downgrade" them. By compromising on proofs, we are at a risk of generating inconsistent specifications. In fact, sometimes we cannot prove within the formal Event-B rules that a transformation does not modify the original behavior. This implies that the correctness of these transformations must be asserted through other means. We have then chosen to follow the mathematical tradition of providing rigorous and convincing arguments as a proof of the preservation of the behavior for each transformation heuristic.

7.6 Limitations of Brama

The situations where Brama cannot animate a specification can be arranged in a typology of five typical cases:

I Brama does not support the finite clause in axioms

II Brama must interpret quantifications as iterations

- **II.1** Brama only operates on finite sets
- **II.2** Brama cannot compute finite sets defined in comprehension with nested quantification

II.3 Brama explicitly requires typing information of all those sets over which iteration is performed in an axiom

III Brama cannot compute dynamic functional bindings in substitutions

III.1 Brama does not support dynamic mapping of variables in substitutions

III.2 Brama does not support dynamic function computation in substitutions

IV Brama does not compute arbitrary functions

IV.1 Functions with analytical definitions in context cannot be computed in events

IV.2 Functions using case analysis can not be expressed in a single event

IV.3 Invariants based on function computations can not be evaluated

V Brama has limited communication with its external graphical animation environment

We have gathered these observations about Brama while animating two specifications which will be discussed in details later in chapter 9. The non-animatability of specifications has two main reasons: either it is the limitation of the animator or the expression itself is too complicated to be executed.

7.7 Changing the class of a specification

In order to animate a specification, we have to bring it into the right class. We change its class primarily by reformulating its expressions and adding some constructive elements to it. Some of such techniques are usage of extension for a finite domain, definition of upper and lower bounds to ensure termination, simplification of complex formulas, rewriting of complex non-constructive expressions into executable format, inline/macro expansion of the formula instead of calling the function, decomposition of events to include all the cases defined by functions, etc. Our main class changing constructive techniques, depicted by figure 7.4, are as following:

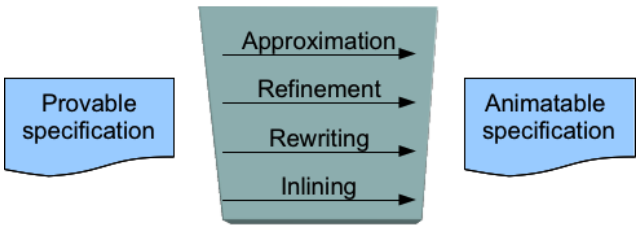

Figure 7.4: Types of class changing transformational heuristics

7.7.1 Approximation

Approximation is a standard mathematical technique to represent something close enough to the original value. It is a reasonably fast solution to be useful for computation and execution. In our transformations, we use two types of approximations: under-approximation and over-approximation. The former is the idea of taking a reasonable subset of the original expression, whereas the latter takes the superset. These approximation techniques are based on abstract interpretation [Cousot 1977] and are often used to address state explosion problems in model checking.

We use under-approximation to address the termination problem. This is a specific kind of termination which is based on enumeration of values. When a formula is based on an unbounded value it becomes non-terminal because animator will continue enumerating it infinitely. Consequently, Brama fails to execute such expressions. Therefore, we define their upper and lower bounds which settles the case.

In other cases, where we have to deal with complex data structures, such as sequences or lists, we exploit the over-approximation technique. The primitive data type of sequence is not provided in Event-B, consequently, we use its standard definition which is based on nested quantifications (see section 8.2 for more details). As we have already discussed that (un-bounded) quantifications cause problems during animation, so, apparently, it seems that under-approximation can again be the solution and defining bounds will solve the problem. This case is different because expressions involving nested quantifications become very complex for the animator to execute. Therefore, we simplify the expression using over-approximation technique to achieve its animatability.

The idea behind this transformation is twofold: first, to simplify a formula to replace non-executable elements with something executable, second, to employ an approach to quickly analyze and establish that if some property exists in the abstract (over-approximate) specification then it holds in the concrete specification that it abstracts. However, if the property does not hold in the former, we do not know if the latter violates this property.

7.7.2 Refinement

Refinement is an established formal activity to transform an abstract (high-level) formal specification into a concrete (low-level) executable program. This is exactly how we use this technique and transform our non-executable high-level non-constructive formulas and expressions into lower-level animatable and executable elements. Although transformations achieved by approximations can also be discussed in terms of abstract-refinement relationship, yet the solutions to Brama's problems II.3, IV.2 and specially V are cases of refinements.

The following proof obligation must be proved in order to define the abstract-refinement relationship between the original and transformed specifications:

$$P(s,c) \land I(s,c,v) \land Q(s,t,c,d) \land$$
$$J(s,t,c,d,v,w) \land H(s,t,c,d,w) \land S(s,t,c,d,w,w') \Rightarrow$$
$$G(s,c,v) \land \exists v'.(R(s,c,v,v') \land J(s,t,c,d,v',w'))$$

Where v defines the variables of the abstract machine and w defines the variables of refined machine, s and c define the sets and constants of the abstract context, and t and d define the sets and constants of refined context. The axioms on the sets and constants of the abstract context are denoted by $P(s,c)$ and on the refined context by $Q(s,t,c,d)$. The invariant of abstract machine is denoted by $I(s,c,v)$ and of the refined machine by $J(s,t,c,d,v,w)$. $G(s,c,v)$ is the guard of the abstract event whose before-after predicate is defined by $R(s,c,v,v')$. The corresponding concrete event of the refined machine has the guard $H(s,t,c,d,w)$ and the before-after predicate $S(s,t,c,d,w,w')$.

7.7.3 Rewriting

Rewriting is a method to replace formulas and expressions with their equivalent counterparts. The rationale behind using this technique for transformation is same as others: to replace non-executable elements with their equivalent but executable counterparts.

In generalized substitutions, dynamic functions whose parameters are passed at runtime (non-deterministically) and depend upon the computations performed by guards, are hard for Brama to compute. The same hardships are also faced when Brama has to compute sets of tuples in generalized substitutions. This is due to the inability of the animator to perform some standard operations. Our approach towards these animation problems is to reformulate the non-computable formula by its counterpart in set algebra. While less readable, it has the same meaning and is easy enough for animation.

7.7.4 Inlining

Inline/macro expansion is an optimization technique to replace the call of a function by its body. While writing specifications, this is a common practice to use functions for readability and simplifying proofs. These functions are defined in contexts using axioms. A function based on a case-analysis has multiple definitions and cannot be enumerated straightforwardly. Consequently, specification fails to execute.

This problem can be solved by using inline expansion technique, i.e., to replace the function call by its body. We take the function body from the context and replace it in events where they are called. Like this, we do not have to pre-define values at compile time and animator gives the values to guards at run-time by itself, so problem is solved and animation is possible.

Inline expansion technique, in fact, is based on two previously defined transformational techniques: rewriting and refinement. It is rewriting because we are replacing the function call by its body which means semantically both expressions

are equivalent, of course, proper care has to be exerted with the use of involved variables. It can be defined as the refinement of the original machine if we can prove the enabledness preservation of the involved events. The following proof obligation must be discharged:

$$\forall S, C, Sr, Cr, V, Vr, x, xr.A \wedge Ar \wedge I \wedge Ir \Rightarrow (Gr \Rightarrow G)$$

Where S and C respectively represent the sets and constants of the abstract context, Sr and Cr respectively represent the sets and constants of the refined context, V is the variables of the abstract machine, Vr is the variables of the refined machine, x is the local variables of the abstract event, xr is the variables of the refined event, and A, Ar, I, Ir, G, Gr are the axioms, invariants and guards of the abstract and refined machines respectively.

7.8 Summary

Proofs alone are not sufficient to ensure that the specification is an adequate model of the problem we want to solve; they need to be complemented by validation techniques, such as animation. However, several ingredients of well written specifications, such as abstractness, non-determinism, non-constructive definitions, are non-animatable. Provability and animatability are two different sets of classes of a formal specification. In order to be validated, a specification must belong to the latter class. This transformation can be achieved by employing techniques, such as refinement, rewriting, etc.

In this chapter, we have explained what are the issues which may impede the animation of a formal specification. We have discussed the differences between provable and animatable specifications. We have introduced the animator Brama and also noted down its limitations for animation. In the end, we have presented some techniques which help obtain the animatability of non-computable traits of formal specifications.

Chapter 8

Transformational heuristics & formal semantics

Contents

8.1 Introduction

The distinction between the classes of non-animatable and animatable specifications leads to the emergence of a set of heuristics which assists in their transformation from the former class to the latter. These transformational heuristics are designed to keep the behavior of the specifications unaltered, possibly at the expense of other formal properties, such as provability. Their correctness then becomes an issue.

We address this issue in two steps. In the first step, we present the heuristics in a descriptive frame and give rigorous arguments to justify their use, notably that they are applied to the already verified formal text. In the second step, we give a formal proof of their correctness. The proof indicates under which conditions both original and transformed specifications are behaviorally equivalent, i.e., provided the same

values, the same sequences of events can be followed on both specifications. Some of the proof obligations cannot be discharged within the formal framework of Event-B. The general behavior preservation property states that whatever we observe on the animation of the transformed specification would have been observed on the animation of the original specification.

The chapter is organized as follows: In section 8.2, we discuss the heuristics to change the class of a specification. In section 8.3, we present the formal semantics of the transformations which assist in defining the theorem of behavioral equivalence of non-animatable and animatable specifications. In section 8.4, we give the formal proof of behavioral equivalence of each heuristic. Section 8.5 concludes the chapter.

8.2 Transformational heuristics

The aforementioned animation problems of Brama, discussed in section 7.6, lead us to design 10 transformational heuristics, one for each case. We designed the heuristics to preserve the behavior of the specification, not its formal properties. In particular, the transformation may not be provable within Event-B formal logic system. The correctness of the transformation is then a crucial issue.

One of the strategies to address the issue is relying on the mathematical tradition of rigorous arguments. For this to work, we need a basic assumption: the initial specification text must have been formally verified. It provides us with a safeguard that correct modification of the formal text would not affect the behavior of the specification. Following this, we have proposed a rigid pattern, shown in figure 8.1, which describes each heuristic in a standard format.

For each heuristic, we first describe the **symptom**, i.e., in which particular case this heuristic should be used. The symptom generally relates to the animation problems discussed in section 7.6. It also indicates the construct of Event-B model, such as, axiom, guard, or generalized substitution, where the problem lies and which is susceptible to modification. The **transform** explains how the original statement must be transformed in order to be animatable. Each transform is based on the execution techniques discussed in section 7.7. **Caution** is the description of the applicability conditions, the hypothesis to check, the possible effects, and the precautions to follow. In the **justification** part, we provide a rigorous argument about the validity of the transformation; we describe why this solution works. Although not strictly formal, yet this rigorous and clear description frame allows us to use animation safely to validate specifications. Some heuristics are also associated to a proof-obligation, discussed later in section 8.4, which once discharged ensures the soundness of the application of the heuristic.

Heuristic 1: Remove the axiom `finite` from the specification

Symptom: Error message about the keyword `finite` not being supported. The problem lies in the axioms of the model.
Transform: Remove all the instances of the axiom `finite` from the specification.

Heuristic pattern	
Symptom:	What reveals the situation, i.e., Brama error message
Transform:	The expression schema of the original specification and its transformed counterpart
Caution:	Description of the application conditions, hypothesis to check, possible effects and precautions to follow
Justification:	A rigorous argument about the validity of the transformation

Figure 8.1: The heuristic pattern

Caution: Removal of the axiom `finite` invalidates many well-formedness proof-obligations.
Justification: Axioms like finiteness and non-emptiness can be considered as purely technical axioms which are just used to discharge related proof obligations. They do not bring much information into the specified system whose implementation will necessarily be finite, even if it could conceptually be infinite. These technical axioms are required by the inference rules used by the provers. Since all the sets of values will be defined by extension, the animation will work upon necessarily finite values. The behavior is trivially maintained.

Heuristic 2: Specify the finiteness of a quantified domain

Symptom: Error message about the dependent variables which do not have an iterator. The problem lies in the axioms of the model.
Transform: Limit the range of the list.

Original $n.n \in \mathbb{N} \Rightarrow expression(n)$

Transformed $n.n \in min..max \Rightarrow expression(n)$

Caution: The range must be wide enough so that the values of n computed during the animation never fall outside it. Some proof obligations may become impossible to discharge (e.g., $n + 1 \in \mathbb{N}$).
Justification: This heuristics is the opposite of the heuristic 1; the argumentation on the necessary finiteness of the values during animation holds. The major difference with the heuristic 1 is the necessity to check during the whole animation that the range is always wide enough. If this condition is ensured then the behavior is unchanged.

Decidability is a common animation problem. Our solution to ensure it via typing and stating that any variable, parameter or constant can only take finitely possible values is inspired by the solution proposed by ProB.

In a broader formal framework spectrum, this is an example of refinement. The newly constructed expression is a refined version of the original expression which contains lesser but more precise values. From a more focused abstraction frame-

work's point of view, this is under-approximation which allows us to quickly check the state reachability by exploring the subset of the reachable states.

Heuristic 3: Generalize expressions involving complex iterations

Symptom: Error message about the impossibility to build the iterators of the predicate. The problem lies in the axioms of the model.
Transform: Take the super-set of the expression.
Original $var = \{x | \exists n.n \in \mathbb{N}_1 \land x \in 1..n \rightarrow y\}$
Transformed $var \in \mathbb{P}(\mathbb{N} \nrightarrow y)$
Caution: This transformation loosens the constraints on the values, some maybe essential to the behavior (for instance, the property that all integer between 1 and the length of the sequence belong to the range of the function). Brama cannot ensure anymore that the property holds. The burden of the check is passed onto the input of the values. It must be ensured that animation is performed on a shared set of values between the original and transformed specifications.
Justification: On the subset of shared values (that is, those values respecting the constraints left out by the generalization), both specifications must have the same behavior. Two cases must be considered:

- the value is associated to a constant: it does not change during the animation and it keeps its properties,
- the value is associated to a variable: at least one of the proof obligations in the initial specification deals with proving that the result of the computation belongs to the set. Since the initial specification is verified, the values in the modified specification have the same property.

This is an example of abstraction because the transformed formula is an abstraction of the original formula. In abstraction framework, this technique is known as over-approximation.

Heuristic 4: Explicitly provide the typing information of all sets used in an axiom

Symptom: Error message about the impossibility to build the iterators of the predicate. The problem lies in the axioms of the model.
Transform: Always provide the type of variables.
original $x.expression(x)$
Transformed $x.x \in X \Rightarrow expression(x)$
Caution: The type provided must be consistent with the type inferred by the provers.
Justification: Brama does not use the information derived by the provers. The provided set is actually redundant. Brama needs it to set up the iteration process. Two cases must be considered:

- if the type is equal to a carrier set, or a subset, the modified expression is just a redundant form of the initial expression,
- if the type is an infinite set, such as $\mathbb{N}$, then heuristic 2 should also be applied with caution and justification.

Heuristic 5: Avoid expressions involving mapping of variables in substitutions

Symptom: Error message: "Default number can not be casted to IMapplet." Brama does not compute sets of tuples in substitutions. The problem lies in the substitutions of the model.
Transform: Rewrite the substitution to avoid mapping.
Original $\{x, y.x \in X \land y \in Y | x \mapsto y\}$
Transformed $\{x \in X | x\} \times \{y \in Y | y\}$
Justification: The transformation is simply a rewriting of the initial expression as a formula in set algebra. While less readable, it has the same meaning. This heuristic can also be used in guards and axioms.

Heuristic 6: Avoid dynamic function computation in substitutions

Symptom: Error message: "Related invariant is broken after executing the event." Brama cannot apply a function defined by its graph in a substitution.
Transform: Rewrite the substitution to avoid function computation.
Original $\{x.x \in X | fun(x)\}$
Transformed $\{ran(\{x.x \in X | x\} \lhd fun)\}$
Justification: The transformation is simply a rewriting of the initial expression as a formula in set algebra. While less readable, it has the same meaning.

Heuristic 7: Inline in events the functions defined in contexts

Symptom: Startup values can not be fed to complex functions. The problem lies in the context of the model.
Transform: Substitute function calls by their inlined equivalent
Original (in Context) $\forall x.x \in S \Rightarrow f(x) = expression(x)$
Original (in Event) $f(v)$
Transformed (in Context) $true$
Transformed (in Event) Add a new guard $v \in S$ and replace $f(v)$ with $expression(v)$
Caution: All occurrences of f in the specification must be replaced; be consistent when replacing formal parameters by actual values.
Justification: This is the case of refinement. In a mathematical context, the value $f(v)$ is equal to its definition expression where v has been substituted to x; both expressions are interchangeable.

Contexts in Event-B are precisely meant to contain constants and general definitions, such as functions. Using this structure eases the proofs and provides better

legibility. As for the previous two heuristics 5 and 6, this heuristic is also strongly connected to the issue of readability and understandability of formal texts.

Heuristic 8: Replicate events which use functions defined "by cases"

Symptom: Same as the heuristic 7.

Transform:

Original (in Context) $\forall x.x \in S \Rightarrow (p(x) \Rightarrow f(x) = expression(x) \wedge q(x) \Rightarrow f(x) = expression'(x))$

Original (in Machine) EVENTS

EVENT A
WHERE ...f(v)...
THEN ...f(v)...
END

Transformed (in Context) true

Transformed (in Machine) EVENTS

EVENT A1
WHERE ...
grdC1 p(v)
THEN ...
END
EVENT A2
WHERE ...
grdC2 q(v)
THEN ...
END

Caution: This heuristic must be followed by the application of heuristic 7. Check that all cases have been covered. Be particularly careful if the function is applied to several different actual parameters; this may require several applications of this heuristic.

This heuristic entails major surgery in the specification. A blind application may introduce many copies of the events. By using the structures of the other guards (some may already prevent cases in the function definition to be used) and by grouping several functions into one transformation, it is possible to reduce the number of duplications.

Justification: This is a case of refinement. The predicates used in the "by case" definitions are equivalent to guards in events. They have the same form and are used for the same purpose. Events A1 and A2 are copies of A, except for the new guard: their union is equivalent to A. Hence, the transformed specification has the same behavior as the original specification.

Heuristic 9: Remove Invariant

Symptom: Error message about the dependent variable which does not has an iterator. The problem lies in the invariant of the model.
Transform: Remove the related invariant.
Caution: Removal of the invariant may invalidate some proof obligations.
Justification: Invariants express the conditions to which a specification must adhere. When applied to a proven specification, removal of invariant is safe because (1) invariant do not modify behaviors (they are only observed) and (2) proof obligations related to maintaining the invariant have already been successfully discharged.

Heuristic 10: Introduce observatory variables

Symptom: No error message.
Transform: Introduce observatory variables to the specification.
Caution: Use this heuristic only for observatory purposes, not for introducting a new behavior. Make sure that they are not used on the right hand side of any generalized substitution or in any guard.
Justification: The observation variables, invariants and events are introduced to the specification when we want to demonstrate a particular behavior in external flash application. Since the Flash interface is bound to the Event-B specification where the actual values are being changed so it is easier to introduce new constructs there rather than at the front end. These new constructs are purely cosmetic changes to the specification; they facilitate its graphical look, and do not define any new behavior.

8.3 Formal semantics of the transformations

Rigorous arguments can be used to justify the usage of the transformational heuristics. Though necessary, they are not sufficient to prove their correctness. This section therefore aims at defining precise semantics for these transformational heuristics. We provide the formal definitions of basic ingredients which we use to assert the correctness of our heuristics. We define what is a behavior of a specification, what is a relation between the original and transformed specification, how can we characterize that two specifications share the same set of behaviors, and more importantly, what is behavior equivalence.

These formal semantics assert the fundamental concept which we have employed to reason about our idea of transforming a provable specification into an animatable one which is "anything that is observed during the animation of the transformed specification would have been observed on the animation of the provable specification."

8.3.1 State names

State names N is a set of all legal names of variables and constants.

8.3.2 State values

State values V is a set of all legal values in Event-B which are required by an animator to perform the animation.

8.3.3 States

States is a set of mappings of names to values constrained by invariants.

$$S = N \to V$$
$$\forall s.s \in S \Rightarrow Inv(s)$$

8.3.4 Event

An event is a transition from one state value to another. Event E is made of a guard G, which is a predicate built on a state s and expresses necessary condition for the transition, and generalized substitution U which describes how the state is modified.

$$E = \ When\ G(s)\ \ Then\ U(s)\ End$$

8.3.5 Behavior

A behavior is a sequence of quadruples of an initial state, an event, state values, and a reached state. It is defined as:

$$b \in seq(S \times E \times \mathbb{P}(V) \times S)$$
$$\forall i.i \in dom(b) \wedge fourth(b(i)) = first(b(i+1))$$

where the starting state of a behavior is the initial state of the machine, E represents the set of all Es defined in section 8.3.4 and $first$ and $fourth$ are the respective projections.

We say that a state t is reached from a state s after the firing of an event e with some state values v. A simple behavior can be denoted as $s \xrightarrow{e(v)} t$. Here, e is an element of E and v represents any state values chosen non-deterministically to make the guard of e true. The pool of these possible values is represented by $\mathbb{P}(V)$.

8.3.6 Specification

Specification $Spec$ is a syntactically correct Event-B model.

8.3.7 Relation between specifications

There exists a relation between a transformed specification and its original specification. We say that for all transformed events e' in a transformed specification $Spec_t$, there exists an event e in the original specification $Spec_o$ and a relation Rel between both of these events and vice versa.

$$\forall e'.e' \in Events(Spec_t) \Rightarrow \exists e.e \in Events(Spec_o) \wedge e' \mapsto e \in Rel$$
$$\forall e.e \in Events(Spec_o) \Rightarrow \exists e'.e' \in Events(Spec_t) \wedge e' \mapsto e \in Rel$$

Where $Event(Spec)$ denotes the set of all events in the specification $Spec$.

8.3.8 Shared state values

The set of shared state values is composed of all legal values which are permissible in the original and transformed specifications. Let V_o and V_t be the sets of all legal state values in the original and transformed specification respectively, then V_c is the set of shared state values of both specifications.

$$V_c = V_o \cap V_t$$

8.3.9 Shared states

Let S_o and S_t be all the states in the original and transformed specification respectively, then S_c is the set of shared states of both specifications.

$$S'_o = \{s.s \in S_o | N_c \cap N_o \lhd s\}$$
$$S'_t = \{s.s \in S_t | N_c \cap N_t \lhd s\}$$
$$S_c = S'_o \cap S'_t$$

8.3.10 Shared behaviors

Let B_o and B_t be the sets of all behaviors of the original and transformed specification respectively. Let b_o and b_t be any two behaviors of B_o and B_t respectively, then b_o and b_t are shared if they share state values, and their events are related by Rel with their corresponding counterpart.

$$B_c = \{(b_o, b_t) | \forall i.i \in dom(b_o) \wedge b_o \in B_o \wedge b_t \in B_t \wedge$$
$$first(b_o(i)) = first(b_t(i)) \wedge third(b_o(i)) = third(b_t(i)) \wedge$$
$$fourth(b_o(i)) = fourth(b_t(i)) \wedge second(b_o(i)) \mapsto second(b_t(i)) \in Rel\}$$

Let us define the generalized relationship Rel^* between the original (B_o) and transformed (B_t) behaviors (i.e., a set of couples of b_o and b_t).

$$\forall b_o, b_t.b_o \in B_o \wedge b_t \in B_t \wedge b_o \mapsto b_t \in Rel^* \Leftrightarrow$$
$$(\forall i.i \in dom(b_o) \Rightarrow$$
$$(second(b_o(i)) \mapsto second(b_t(i)) \in Rel))$$

Now if seen from the transformed specification perspective, then

$$B^t_c = \{b_t | b_t \in B_t \wedge (Rel^{*-1}[\{b_t\}] \subseteq B_o)\}$$

and if seen from the original specification perspective, then

$$B^o_c = \{b_o | b_o \in B_o \wedge (Rel^*[\{b_o\}] \subseteq B_t)\}$$

8.3.11 Behavioral equivalence

In state-based specification languages, such as Event-B, we mainly deal with two main constructs: states and events. To compare the behavior of two specifications, we can make the following observations based on the sequence of these two constructs: enabledness, reachability and closure. Enabledness is the idea that an event is enabled at a particular state in both behaviors. Reachability is the idea that a particular state is reachable in both behaviors. Closure is the idea that from a shared state an event with a shared value always leads to another shared state.

Definition

Two specifications $Spec_o$ and $Spec_t$ are behaviorally equivalent when all behaviors starting from a shared state are shared with $Spec_o$.

$$
\begin{aligned}
&Spec_o \overset{B}{=} Spec_t \triangleq \\
&\quad \forall b_t, i.b_t \in B_t \land i \in dom(b_t) \land first(b_t(i)) \in S_c \Rightarrow b_t \in B_c^t
\end{aligned}
$$

Theorem

If two specifications related by Rel have same enabledness, reachability and closure then they are behaviorally equivalent.

$$
\begin{aligned}
&SameEnabledness(Spec_o, Spec_t) \land \\
&SameReachability(Spec_o, Spec_t) \land \\
&SameClosure(Spec_o, Spec_t) \Rightarrow \\
&Spec_o \overset{B}{=} Spec_t
\end{aligned}
$$

- $SameEnabledness(Spec_o, Spec_t)$

  If an event is enabled with a certain state value associated to a certain state in the original specification then its transform must be enabled in the transformed specification given the same state and value, and vice versa.

$$
\begin{aligned}
&SameEnabledness(Spec_o, Spec_t) \triangleq \\
&\quad (\forall s, e, v.s \in S_c \land e \in Spec_o \land v \in V_c \land enabled(e, v, s) \Rightarrow \\
&\qquad (\exists e'.e' \in Spec_t \land e' \mapsto e \in Rel \land enabled(e', v, s))) \land \\
&\quad (\forall s, e', v.s \in S_c \land e' \in Spec_t \land v \in V_c \land enabled(e', v, s) \Rightarrow \\
&\qquad (\exists e.e \in Spec_o \land e' \mapsto e \in Rel \land enabled(e, v, s)))
\end{aligned}
$$

- $SameReachability(Spec_o, Spec_t)$

  If a state is reachable in the original specification after an event with a certain state value then the same state should be reachable in the transformed specification as well given the transform of the event and the same state value, and vice versa.

$$
\begin{aligned}
&SameReachability(Spec_o, Spec_t) \triangleq \\
&\quad (\forall s, t, e, v.s, t \in S_c \wedge e \in Spec_o \wedge v \in V_c \wedge s \xrightarrow{e(v)} t \Rightarrow \\
&\qquad (\exists e'.e' \in Spec_t \wedge e' \mapsto e \in Rel \wedge s \xrightarrow{e'(v)} t)) \wedge \\
&\quad (\forall s, t, e', v.s, t \in S_c \wedge e' \in Spec_t \wedge v \in V_c \wedge s \xrightarrow{e'(v)} t \Rightarrow \\
&\qquad (\exists e.e \in Spec_o \wedge e' \mapsto e \in Rel \wedge s \xrightarrow{e(v)} t))
\end{aligned}
$$

- $SameClosure(Spec_o, Spec_t)$

  All the states reachable from a shared state, after an event with a shared state value, are shared states as well.

$$
\begin{aligned}
&SameClosure(Spec_o, Spec_t) \triangleq \\
&\quad \forall s, t, e, v.s \in S_c \wedge t \in S_o \wedge e \in Spec_o \wedge v \in V_c \wedge enabled(e, v, s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c
\end{aligned}
$$

- *Proof.* We want to prove:

$$
\begin{aligned}
&SameEnabledness(Spec_o, Spec_t) \wedge \\
&SameReachability(Spec_o, Spec_t) \wedge \\
&SameClosure(Spec_o, Spec_t) \Rightarrow \\
&\forall b_t, i.b_t \in B_t \wedge i \in dom(b_t) \wedge first(b_t(i)) \in S_c \Rightarrow b_t \in B_c^t
\end{aligned}
$$

  Let $Spec_o$ be the original specification and $Spec_t$ be the transformed specification.
  Let Rel be the relation between these specifications.
  Let $B_t = Behavior(Spec_t)$ and $B_o = Behavior(Spec_o)$
  Let $b_t, b_o.b_t \in B_t \wedge b_o \in B_o$

  Now if $SameEnabledness(Spec_o, Spec_t) \wedge$
  $SameReachability(Spec_o, Spec_t) \Rightarrow \exists B_c.b_t, b_o \in B_c$
  Same enabledness and reachability means specifications share behaviors. However, some events may lead to non-shared states, therefore we take closure to consider only the shared states of both specifications, i.e.,

  $\forall s, t, e, v.s \in S_c \wedge t \in S_o \wedge e \in Spec_o \wedge v \in V_c \wedge enabled(e, v, s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c$

  Therefore,

$$
\begin{aligned}
&SameEnabledness(Spec_o, Spec_t) \wedge \\
&SameReachability(Spec_o, Spec_t) \wedge \\
&SameClosure(Spec_o, Spec_t) \Rightarrow \\
&\forall b_t, i.b_t \in B_t \wedge i \in dom(b_t) \wedge first(b_t(i)) \in S_c \Rightarrow b_t \in B_c^t
\end{aligned}
$$

  If the specification has also the same closure (i.e., no transition leads to a non-shared state) in addition to the same enabledness and reachability (shared

behaviors) then the specifications are behaviorally equivalent, i.e., any behavior which is observed in the transformed specification would also be observed in the original specification. □

8.4 Proofs of the heuristics

In this section, we present a formal proof of behavioral equivalence of each heuristic. While some heuristics are both behavior as well as semantic preserving, some are only behavior preserving. For the latter case, we introduce a proof-obligation (PO) which ensures its behavioral equivalence: $PO \vdash Spec_o \overset{B}{=} Spec_t$.

Heuristic 1: Remove the axiom `finite` from the specification

Proof. In this heuristic no event is modified therefore *Rel* is identity.

Removal of the `finite` axiom does not introduce any change in the states of the specification, therefore

$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t)$

Though the clause of finiteness is removed, yet all the set values are defined by extension and the original specification is verified so specification will also maintain the closure property, i.e., $SameClosure(Spec_o, Spec_t)$

Therefore,

$$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t) \wedge SameClosure(Spec_o, Spec_t) \Rightarrow Spec_o \overset{B}{=} Spec_t$$

□

Heuristic 2: Specify the finiteness of a quantified domain

Proof. In this heuristic no event is modified therefore *Rel* is identity.

In this heuristic, we are limiting the range of the supplied values to the constant. These values may impact the behavior. Therefore, it is imperative to check that all the state transitions (affected by this limitation) from a shared state lead to another shared state:

$\forall s, t, e, v . s \in S_c \wedge t \in S_o \wedge e \in Spec_o \wedge v \in V_c \wedge enabled(e, v, s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c$

Following PO needs to be proved: $v \in V_c$

As the original specification is verified, all guards respect the invariants and the transformation explicitly requires the given range to be wide enough to incorporate all the legitimate values, therefore this PO would ensure that we are within the shared states, i.e. $closure(Spec_o, Spec_t)$. However, for proofs like $n + 1 \in \mathbb{N}$ or $n - 1 \in \mathbb{N}$, we can use lazy proof approach or proof by demand, i.e., always extending or retracting *max* and *min* respectively to incorporate the desired value into the range.

Since $Rel = id \wedge S_o, S_t \subseteq S_c \Rightarrow$
$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t)$
Therefore,

$$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t) \wedge SameClosure(Spec_o, Spec_t) \Rightarrow Spec_o \overset{B}{=} Spec_t$$

□

Heuristic 3: Generalize expressions involving complex iterations

Proof. In this heuristic no event is modified therefore *Rel* is identity.

In this heuristic, we are taking the superset of the original values, therefore it is imperative to check the following (closure) property which states that all the state transitions from a shared state will lead to another shared state:

$\forall s, t, e, v.s \in S_c \wedge t \in S_o \wedge e \in Spec_o \wedge v \in V_c \wedge enabled(e, v, s) \wedge s \xrightarrow{e(v)} t \Rightarrow t \in S_c$

Again, following PO needs to be checked: $v \in V_c$

As the original specification is verified, all guards respect the invariants, and the original set is replaced by its super-set which implicitly contains all the abstracted values, therefore this PO would ensure that we are within the shared states, i.e., $closure(Spec_o, Spec_t)$.

Since $Rel = id \wedge S_o, S_t \subseteq S_c \Rightarrow$
$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t)$
Therefore,

$$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t) \wedge SameClosure(Spec_o, Spec_t) \Rightarrow Spec_o \overset{B}{=} Spec_t$$

□

Heuristic 4: Explicitly provide the typing information of all sets used in an axiom

Proof. We are just supplying the missing type; the behavior remains intact. □

Heuristic 5: Avoid expressions involving mapping of variables in substitutions

Proof. The transformed expression is the rewriting of the original expression in set algebra. The behavior of both specifications is the same. □

Heuristic 6: Avoid dynamic function computation in substitutions

Proof. The transformed expression is the rewriting of the original expression in set algebra. The behavior of both specifications is the same. □

Heuristic 7: Inline in events the functions defined in contexts

Proof. This is the case of inline expansion which makes the *Rel* and therefore following PO needs to be checked:

$$\forall S, C, Sr, Cr, V, Vr, x, xr.A \wedge Ar \wedge I \wedge Ir \Rightarrow (Gr \Rightarrow G)$$

$Ir = I$ because invariants are not changed in this heuristic

$Ar \subset A$ because axioms of the transformed context are subset of the original context

Therefore, we are left to prove:

$$\forall S, C, Sr, Cr, V, Vr, x, xr.A \wedge I \Rightarrow (Gr \Rightarrow G)$$

This can be proved with the enabledness preservation proof. This PO can also be discharged within the formalism of Event-B using refinement.

Once we have proved that $SameEnabledness(Spec_o, Spec_t)$, we are now left to prove the other two properties of behavioral equivalence, i.e., same reachability and closure.

Since in this heuristics, the states and transitions are not modified, therefore

$$S_o, S_t \subseteq S_c \wedge SameReachability(Spec_o, Spec_t) \wedge \\ SameClosure(Spec_o, Spec_t) \Rightarrow Spec_o \stackrel{B}{=} Spec_t$$

□

Heuristic 8: Replicate events which use functions defined "by cases"

Proof. This is the case of inline expansion followed by event decomposition which introduces a *Rel* different from identity; therefore the following PO needs to be checked which states that the new guards cater for all the cases defined by the original guard.

$$G_e(v) \Rightarrow \exists e'.e' \in Rel[\{e\}] \wedge G'_{e'}(v) \wedge (\forall e'.G'_{e'}(v) \Rightarrow G_e(v))$$

This can be proved with the enabledness preservation proof. This PO can also be discharged within the formalism of Event-B using refinement.

Once we have proved that $SameEnabledness(Spec_o, Spec_t)$, we are now left to prove the other two properties of behavioral equivalence, i.e., same reachability and closure.

Since in this heuristics, states and transitions are not modified, therefore $S_o, S_t \subseteq S_c \wedge SameClosure(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t)$

Therefore, $Spec_o \stackrel{B}{=} Spec_t$. □

Heuristic 9: Remove Invariant

Proof. In this heuristic no event is modified therefore *Rel* is identity.

The original specification is already proven and all (invariant related) proof obligations have been discharged. Since invariants do not modify the behavior of the specification, removal of an invariant would not change the set of states and transitions on them, therefore

$$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t) \wedge SameClosure(Spec_o, Spec_t) \Rightarrow Spec_o \stackrel{B}{=} Spec_t$$

□

Heuristic 10: Introduce observatory variables/invariants/events

Proof. In this heuristic no event is modified therefore *Rel* is identity.

On shared states,

$SameEnabledness(Spec_o, Spec_t) \wedge SameReachability(Spec_o, Spec_t) \wedge SameClosure(Spec_o, Spec_t)$

The transformed specification is the refinement of the original specification. In the formalism of Event-B, it can be proved by defining the refinement-relationship between both. The set of observations on both specifications is identical.

Therefore, $Spec_o \stackrel{B}{=} Spec_t$. □

8.5 Summary

Animation of non-executable specifications can be sought by our proposed transformational heuristics. However, some of them do not ensure the provability of the specification any more. Their soundness is then an issue. In order to address this question, we have developed an ad-hoc semantics based on the behavior of the model. The behavioral preservation property states that whatever we observe on the execution of the transformed specification would have been observed on the proven specification.

In this chapter, we have first presented the heuristics which transform non-animatable specifications into animatable ones. We have devised a generic pattern and in its descriptive frame, we have discussed their symptoms, proposed transformations, caution clauses, and rigorous arguments to justify their application. We have then introduced the formal semantics of the transformations. We have formally defined the basic ingredients which are then used to prove their correctness. We have also presented the theorem of behavioral equivalence which relates the transformations to the formal semantics. In the end of the chapter, we have given a formal proof of correctness for each heuristic.

Chapter 9

Application of the heuristics on case studies

Contents

9.1 Introduction

In order to investigate the utility of our transformational heuristics, we apply them on two case studies to validate their behaviors using animation. The first case study is about the land transport domain model which we have already specified in chapter 5. The second case study is about a platooning system. Specifications of both models are animated by creating reasonable behavioral scenarios representing the protocols that must been observed in the reality. The animator is provided with startup values accordingly.

Not all refinements are animated. Some refinements based on small incremental steps are uninteresting from the animation's point of view because they do not

bring much information in terms of new behaviors. At least, one refinement per observation level was subjected to animation.

The effect of the heuristics is an animatable specification. Their application on the formal text is presented in a before-after state clearly indicating how it has been transformed. When necessary, the application of the heuristics is justified in the form of a formal proof.

The chapter is organized as follows: Section 9.2 presents the first case study about the land transport domain model. Section 9.3 presents the second case study about the model of a platooning system. Chapter is finally concluded in section 9.4.

9.2 Case study 1: The land transport domain model

The specification in this case study is about the modeling of land transportation domain. In the model, we want to express the properties that any system working within the domain is expected to meet and maintain.

In this specification effort, the focus is on the formal definition of domain's laws, protocols and properties, rather than on the implementation of a particular system. Refinement is used to introduce new notions; the proof obligations serve to guarantee the consistency of the model.

The current specification consists of 7 refinements and four observation levels. This specification exhibits several properties which call for animation, namely:

- complex data which constraint behaviors (following a route, for instance),
- protocols and iterations (travel as a sequence of hub crossing and path traversing protocols, for instance), and
- non-deterministic interaction between elements (autonomous vehicles, for instance).

To validate this specification, only first six heuristics, defined in section 8.2, were used. Then, setting up animation was easy and we used it intensively. Actually, we used animation more as prototyping rather than validation while beginning the work on refinements. It helped us understand and fix desired behaviors.

Not all refinements were animated, but we made sure to have one for each observation level. A typical example of a refinement which is uninteresting from animation's point of view was `Movement5` where the notion of time was introduced. The aim of the refinement was mainly to set the vocabulary (`travel_time, clock`) and very general properties: time always increases, clock ticks, etc. The definitions are highly non-deterministic and there is not much in terms of new behaviors. Animation of that particular refinement would not bring much information. `Movement7`, where the actual computation of time, a new complex behavior, is defined, was subjected to animation.

An interesting point to note is that a machine may not be animatable while its refinement may be; there is no monotonicity. We also followed an incremental

approach while animating. We thus ensured that the correction of an error trickles down the refinement chain.

9.2.1 Machine Movement0

This machine is the most abstract. It contains only one event, `travel`, which states that vehicles change their location on a network. The behavior is simple and does not present much interest per se. The interesting point lies in the definition of the values in the contexts. In particular, the context `Net` specifies the topology of the network (a directed graph) and the basic properties which make it a transportation network. It is at this level that the foundations of the model instance are built.

We had to apply the heuristic 1 (erasure of `finite` axioms) on `Net` to animate `Movement0`. All the sets and constants of this machine were declared as finite. As a result of the application of the heuristic 1, axioms such as:

$$\forall n.n \in Nets \Rightarrow card(obsNetHubs^{-1}[n]) \geq 2 \wedge card(obsNetConnections^{-1}[n]) \geq 1$$

which expresses that a transportation network has at least two hubs and one connection, cannot be verified. Its well-formedness proof-obligation requires `obsNetHubs` and `obsNetConnections` to be finite functions, hence their domains and codomains (`Nets`, `Hubs`, and `Connections`) be finite too. As the consequence of application of the heuristic 1, the finite axioms have been removed, thus making the machine animatable but the context unprovable.

We also applied the heuristic 1 to `Vehicle`. The set of `Vehicles` was declared as finite and we removed this axiom as a consequence of application of heuristic 1. Figure 9.1 shows the context `Net` before and after the application of the heuristic 1.

9.2.2 Machine Movement1

`Movement1` refines the model by specifying that the locations of vehicles are `Hubs` at the beginning and the end of `travel` event. Since this machine introduces only a small refinement in the model, its animation is straightforward. We were not required to apply any of the heuristics other than heuristic 1 in order to animate this machine.

9.2.3 Machine Movement2

The refinement `Movement2` introduces the notion of routes. A `route` is a sequence of `paths`; a `path` is an edge in the graph between two `hubs` which are the vertices. The event `travel` is refined to state that locations are connected by a route.

The set of routes is introduced in the context `Net2`. We had to use the heuristic 3 because we defined the notion of sequence as follows:

$$seqPaths = \{seq | \exists n.n \in \mathbb{N}_1 \wedge seq \in 1..n \rightarrowtail paths \wedge finite(seq) \wedge card(seq) = n\}$$

CONTEXT
Net
SETS
Nets, Hubs, Connections
CONSTANTS
obsNetHubs, obsNetConnections, connectionOrigin, connectionDestination
AXIOMS
tec1 finite (Nets)
tec2 finite (Hubs)
tec3 finite (Connections)
tec4 finite (obsNetConnections)
tec5 finite (obsNetHubs)
tec6 $Nets \neq \varnothing$
tec7 $Hubs \neq \varnothing$
tec8 $Connections \neq \varnothing$

typ1 $obsNetHubs \in Hubs \leftrightarrow Nets$
typ2 $obsNetConnections \in Connections \rightarrow Nets$
typ3 $connectionOrigin \in Connections \rightarrow Hubs$
typ4 $connectionDestination \in Connections \rightarrow Hubs$
typ5 $hubConnections \in Hubs \rightarrow \mathbb{P}(Connections)$

pro1 $dom(obsNetHubs) = Hubs \wedge ran(obsNetHubs) = Nets$
pro2 $dom(obsNetConnections) = Connections \wedge ran(obsNetConnections) = Nets$
pro3 $\forall c \cdot c \in Connections \Rightarrow connectionOrigin(c) \neq connectionDestination(c)$
pro4 $\forall n \cdot n \in Nets \Rightarrow card(obsNetHubs^{-1}[\{n\}]) \geq 2 \wedge card(obsNetConnections^{-1}[\{n\}]) \geq 1$
pro5 $\forall c \cdot c \in Connections \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionOrigin(c)\}] \wedge obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionDestination(c)\}]$
pro6 $\forall h, c \cdot h \in Hubs \wedge c \in hubConnections(h) \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{h\}]$
pro7 $\forall c \cdot c \in Connections \Rightarrow card(obsNetConnections[\{c\}]) = 1$
pro8 $\forall h.h \in Hubs \Rightarrow card(obsNetHubs[\{h\}]) \geq 1$
END

CONTEXT
Net
SETS
Nets, Hubs, Connections
CONSTANTS
obsNetHubs, obsNetConnections, connectionOrigin, connectionDestination
AXIOMS

tec6 $Nets \neq \varnothing$
tec7 $Hubs \neq \varnothing$
tec8 $Connections \neq \varnothing$

typ1 $obsNetHubs \in Hubs \leftrightarrow Nets$
typ2 $obsNetConnections \in Connections \rightarrow Nets$
typ3 $connectionOrigin \in Connections \rightarrow Hubs$
typ4 $connectionDestination \in Connections \rightarrow Hubs$
typ5 $hubConnections \in Hubs \rightarrow \mathbb{P}(Connections)$

pro1 $dom(obsNetHubs) = Hubs \wedge ran(obsNetHubs) = Nets$
pro2 $dom(obsNetConnections) = Connections \wedge ran(obsNetConnections) = Nets$
pro3 $\forall c \cdot c \in Connections \Rightarrow connectionOrigin(c) \neq connectionDestination(c)$
pro4 $\forall n \cdot n \in Nets \Rightarrow card(obsNetHubs^{-1}[\{n\}]) \geq 2 \wedge card(obsNetConnections^{-1}[\{n\}]) \geq 1$
pro5 $\forall c \cdot c \in Connections \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionOrigin(c)\}] \wedge obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionDestination(c)\}]$
pro6 $\forall h, c \cdot h \in Hubs \wedge c \in hubConnections(h) \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{h\}]$
pro7 $\forall c \cdot c \in Connections \Rightarrow card(obsNetConnections[\{c\}]) = 1$
pro8 $\forall h.h \in Hubs \Rightarrow card(obsNetHubs[\{h\}]) \geq 1$
END

Figure 9.1: The context `Net` before (left) and after (right) the application of the heuristic 1

This axiom was replaced by:

$$seqPaths \in \mathbb{P}(\mathbb{N} \nrightarrow paths)$$

The original axiom defines the route as a sequence of paths. There is no support of the *sequence* in Event-B data structure, so we are forced to use its definition. This definition uses double quantification which Brama is unable to support. We, therefore, take the superset of the expression in order to animate it. Since the typing information of `seqPaths` is now changed, though not in essence, therefore `pro1` and `pro2` which are based on the previous definition of `seqPaths` can also not hold. Consequently, we remove the two properties and merge their basic ingredients into the definition of `routes`. Hence, the specification is now animatable. Figure 9.2 shows the context `Net2` before and after the application of the heuristic 3.

The most important effect of the application of the heuristic is the invalidation of all the proofs, either in `Net2` or in `Movement2` and their subsequent refinements, which relied on the essential property of sequences:

$$\forall s.s \in seqPaths \Rightarrow dom(s) = 1..card(s)$$

Proof of application of the heuristic 3: We are taking the superset of the expression. The input values to `Net2` are supplied carefully while making sure that they conform to the essential property of closure and respect invariants, i.e., $\forall v.Inv(v)$. Since the original set of values is a constant, the values fed to its transformation respect original axiom, and the original specification is verified, therefore the axiom-respecting values make sure that $v \in V_c$. □

9.2.4 Machine Movement3

The refinement `Movement3` introduces the decomposition of the notion of travel as a sequence of smaller movements: paths traversings and hubs crossings. The important behavior we need to validate is that a vehicle follows exactly the same route that was assigned for its travel. Technically, this means that the events `crossHub` and `traversePath` must be fired in a strict order.

The ordering of the events is controlled by two variables, `pathsToTraverse` and `hubsToCross`, which model the movement of the vehicles along their routes. Their initialization in the `startTravel` event required us to use the heuristic 5 to transform

$$pathsToTraverse := pathsToTraverse \Leftarrow\!\!+ \{path.path \in ran(route) | vehicle \mapsto path\}$$

into

$$pathsToTraverse := pathsToTraverse \Leftarrow\!\!+ (\{vehicle\} \times \{path.path \in ran(route) | path\})$$

We also had to use the heuristic 6 to transform

CONTEXT
Net2
EXTENDS
Net1
CONSTANTS
paths, routes, isRoute, seqPaths
AXIOMS
typ1 paths $\subseteq$ Connections
typ2 seqPaths = { seq | $\exists$ n . n $\in$ N1 $\wedge$
seq $\in$ 1..n $\rightarrowtail$ paths $\wedge$
finite (seq) $\wedge$ card(seq) = n}
typ3 isRoute $\in$ seqPaths $\rightarrow$ $\mathbb{B}$
typ4 routes = {sp | sp $\in$ seqPaths $\wedge$
isRoute(sp) = TRUE}

pro1 $\forall$r. r$\in$seqPaths $\wedge$
((connectionOrigin(r(1)) $\in$ stations $\wedge$
connectionDestination(r(card(r)))$\in$stations$\wedge$
(obsNetHubs[{connectionOrigin(r(1))}] $\cap$
obsNetHubs[{connectionDestination(r(card(r)))}]
$\neq \varnothing$) $\wedge$ ($\forall$i.i$\in$2..card(r) $\wedge$
connectionDestination(r(i$-$1)) =
connectionOrigin(r(i))) $\wedge$ connectionOrigin(r(1))$\neq$
connectionDestination(r(card(r))) $\wedge$
($\forall$i1, i2 . i1$\in$1..card(r) $\wedge$ i2$\in$1..card(r) $\wedge$ i1 $\neq$ i2 $\Rightarrow$
connectionOrigin(r(i1)) $\neq$ connectionOrigin(r(i2)))
$\wedge$ ($\forall$ i1, i2 . i1$\in$1..card(r) $\wedge$ i2$\in$1..card(r) $\wedge$ i1$\neq$ i2
$\Rightarrow$ connectionDestination(r(i1)) $\neq$
connectionDestination(r(i2)))) $\Leftrightarrow$
isRoute(r) = TRUE)
pro2 $\forall$c.c$\in$Connections $\Rightarrow$
(connectionDestination(c)$\in$stations $\wedge$
connectionOrigin(c) $\in$ stations $\Rightarrow$
($\exists$r. r$\in$routes $\wedge$ connectionOrigin(c) =
connectionOrigin(r(1)) $\wedge$
connectionDestination(c) =
connectionDestination(r(card(r)))))
END

CONTEXT
Net2
EXTENDS
Net1
CONSTANTS
paths, routes, isRoute, seqPaths
AXIOMS
typ1 paths $\subseteq$ Connections
typ2 seqPaths $\in$ $\mathbb{P}$(N $\nrightarrow$ paths)

typ3 isRoute $\in$ seqPaths $\rightarrow$ $\mathbb{B}$
typ4 routes = {sp|sp$\in$seqPaths $\wedge$
1$\in$dom(sp) $\wedge$ card(sp)$\in$dom(sp) $\wedge$
connectionOrigin(sp(1)) $\neq$
connectionDestination(sp(card(sp))) $\wedge$
isRoute(sp) = TRUE}

END

Figure 9.2: The context `Net2` before (left) and after (right) the application of the heuristic 3

$$hubsToCross := hubsToCross \Leftarrow \{i.i \in 1..card(route) | vehicle \mapsto connectionOrigin(route(i))\}$$

into

$$hubsToCross := hubsToCross \Leftarrow (\{vehicle\} \times ran(ran(\{i.i \in 1..card(route)|i\} \lhd route) \lhd connectionOrigin))$$

Figure 9.3 shows the event `startTravel` before and after the application of the heuristics 5 and 6. In this refinement, we also introduced the variant into the machine because of the decomposition of the `travel` event. It is as follows:

$$card(hubsToCross) + card(connectionsToTraverse)$$

The well-formedness of this variant cannot be proved because of the absence of the finite axioms. This is the only undischarged proof-obligation of this machine.

startTravel ≙ **ANY** vehicle , r **WHERE** grd1 vehicle ∈Vehicles grd2 r∈routes grd3 position (vehicle)∈obsHubLocations(r(1)) grd4 vehicle ∉dom(connectionsToTraverse) grd5 vehicle ∉dom(hubsToCross) **THEN** act1 hubsToCross := hubsToCross ∪ {i . i∈..card(r)| vehicle ↦connectionOrigin(r(i))} act2 connectionsToTraverse := connectionsToTraverse ∪ {p.p∈ran(r)| vehicle ↦p} **END**	startTravel ≙ **ANY** vehicle , r **WHERE** grd1 vehicle ∈Vehicles grd2 r∈routes grd3 position (vehicle)∈obsHubLocations(r(1)) grd4 vehicle ∉dom(connectionsToTraverse) grd5 vehicle ∉dom(hubsToCross) **THEN** act1 hubsToCross := hubsToCross ⩤ ({ vehicle } X ran(ran({i . i∈1..card(r)i }|◁r) ◁ connectionOrigin)) act2 connectionsToTraverse := connectionsToTraverse ⩤({vehicle} X {p.p∈ran(r)|p}) **END**

Figure 9.3: The event `startTravel` before (left) and after (right) the application of the heuristics 5 & 6

9.2.5 Machine Movement4

`Movement4` introduces the concept that the crossing of a hub follows a protocol. The refinement decomposes the `crossHub` event into three events: `enterHub`, `leaveHub` and `wait`. Another property which is introduced at this level is that only a definite number of vehicles can be simultaneously present on a hub. This is an abstract definition of the property of non-collision at a hub. The contexts are enriched by the notion of `hubCapacity` and the definition of states of vehicles with respect to a hub. A variable `hubLoad` is also added to the machine.

An important invariant property which states that a vehicle can be on no more than one hub is introduced at this point as:

$$\forall v.card(\{h.vehState(v \mapsto h) = onHub \cup vehicleState(v \mapsto h) = leaving | h\}) \leq 1$$

It required us to use heuristic 4 to introduce the type of `hub`, so we rewrite:

$$\begin{array}{l} \forall v.v \in Vehicles \Rightarrow \\ \quad card(\{h.hub \in Hubs \wedge vehicleState(v \mapsto h) = onHub \cup \\ \quad vehicleState(v \mapsto h) = leaving | h\}) \leq 1 \end{array}$$

Since both `Vehicles` and `Hubs` are the inferred types, we just introduced a redundant information.

In context `startState4`, the well-formedness of the following axiom could not be verified because of the removal of the finite axiom on `obsHubLocations`.

$$\begin{array}{l} \forall h.h \in Hubs \Rightarrow \\ \quad hubCapacity(h) \geq card(\{v.v \in Vehicles \wedge \\ \quad startVehicleLocation(v) \in obsHubLocations(h) | v\}) \end{array}$$

In `Net4`, we were required to apply the heuristic 2 to the following axiom:

$$\forall s.s \in stations \Rightarrow (\exists n.n \in N \wedge hubCapacity(s) = n)$$

It now appears as:

$$\forall s.s \in stations \Rightarrow (\exists n.n \in 1..100 \wedge hubCapacity(s) = n)$$

Actually the problem comes from the unbounded value of `n` which is a member element of the set of natural numbers. Termination of such expressions is non-decidable, specially if they are not defined as finite.

Proof of application of the heuristic 2: The range of the interval *1..100* was chosen considering the reasonable values with respect to the number of vehicles simultaneously present on a hub. This range would not be superseded during the animation because to keep the scenarios manageable, original set of vehicles `Vehicles` has not been provided with that many values. Since the original set of values is constant and the under-approximated set ($1..100 \subseteq \mathbb{N}$) respect the original axiom and invariants, i.e., $\forall v.Inv(v)$, and the original specification is verified, therefore $v \in V_c$. □

In a previous version of our specification, Brama also helped us to find couple of specification errors at this level. In this previous version, the event `exitHub` changed the state of the vehicle once it exited a hub, now this is done by `leaveHub`. In `exitHub`, we forgot to specify the following guard:

$$vehicle \in dom(hubsToCross)$$

The error was detected because the omission made the event enabled before the `startTravel` event was fired. This violates a basic behavior of the system.

The second error in the same specification found by the animation was in the event `enterHub`. Its guard should have stated that it is true only if the load of the hub is strictly less than its capacity, but instead we had written:

$$hubCapacity(h) \geq hubLoad(h)$$

Interestingly, this error was not caught during the verification: formally, the expression is correct. Actually, it specifies a behavior which could be admissible in another model. The correction of the guard was obvious. The corrected specification could be proven without much problem, and the animation showed the intended behavior: vehicles waited outside when a hub was full.

9.2.6 Machine Movement5

This refinement introduces the notion of time in the model which is then used to calculate the travel time of vehicles. As argued before, the animation of this machine was trivial and uninteresting as it does not introduce much new details in the model.

9.2.7 Machines Movement6 & Movement7

This machine `Movement6` decomposes the `traversePath` event into three sub-events: `moveOnPath`, `waitToEnterOnPath` and `waitToMoveOnPath`. It enriches the protocol of movements of vehicles on paths. Whereas, the machine `Movement7` introduces the events `lockIn`, `lockOut` and `lockOnPath`, which model our fine-grained analysis of grid-lock situations.

The animation of both machines was easy, straightforward and did not induce us to apply any of the heuristics. Probably, this is because of the well-learned animation experiences which were there in the back of our minds while specifying these refinement steps. In fact, both of these refinements were particularly specified by using animation as an aide. We extensively used animation: as soon as the behavior was defined, it is animated right away. Animation, used that way, helped us a lot in the exploration of the modeling of this rather tricky behavior.

9.3 Case study 2: The platooning system

The second case study deals with the specification of a platooning system. Platooning is a mode of moving where vehicles are synchronized and follow one another closely. A platoon can be seen as a road-train where cars are linked by software, instead of by hardware. Platooning has several potential uses in an urban mobility system: augmenting throughput, herding unused cars to stations, or running transient buses, for instance.

Several platooning control systems are being developed and experimented. One locally developed is based on Situated Multi-Agent (SMA) theory. Each car has its own local control algorithm which uses a perception/decision/action loop; the platooning behavior is an emerging property [Daviet 1996, Scheuer 2008].

An Event-B specification of the local model has been written [Lanoix 2008, Colin 2008a, Colin 2008b]. Contrary to the previous case study, the structure of the development in this case study can be interpreted as a sequence of refinements toward an implementation. Each refinement decomposes some events to make explicit a part of the general computation.

The Event-B model of the specification is presented by figure 9.4. The specification consists of five machines (four refinements):

Platoon: defines platoons and sets the basic safety property. It contains only one event, `all_move`, where all vehicles change positions while keeping safe distance.

Platoon_1: decomposes the event into one which moves the leader vehicle and one which moves the followers. This organizes the basic "iteration along the platoon" of each move.

Platoon_2: computes the length of each basic move. This leads to the introduction of kinematic functions in the contexts and to the refinement of move events into several ones, each corresponding to a different situation (wether the maximum and minimum speeds are reached or not). This models the *action* part of the SMA.

Platoon_3: introduces the notion of *decision* of the SMA model into the specification. Two events, one for the leader, and one for the followers, are introduced and integrated in the control loop.

Platoon_4: introduces the notion of *perception* which allows decision events to be refined so the actual computation of the parameters of the control law (acceleration) can be performed.

Although the last refinement is very close to an implementation, in spirit if not in form, yet we decided to use animation to validate the specification for several reasons. The first was curiosity as the heavy use of functions was challenging, the second was to compare the results of the animation with the results of simulations that had been previously made, and the last was to confirm that a certain "formal approximation" was legitimate.

The last reason is a consequence of using discrete tools to model what is inherently continuous. In this case, all proof-obligations were discharged, assuming one property, namely $x(y/z) = (xy)/z$, holds. True in $\mathbb{R}$, this property is false in $\mathbb{N}$. However, the difference becomes actually negligible when numerators are much bigger than denominators. Animation with realistic values gives insight on the validity of the "approximation" and on the solidity of the model.

Although all machines have been animated, the first four are not particularly interesting. The non-deterministic definition of the parameter of the control law does not allow for long automatic run of the animation. To observe interesting behaviors, we have to feed "coherent" values to each event which is fired. This can

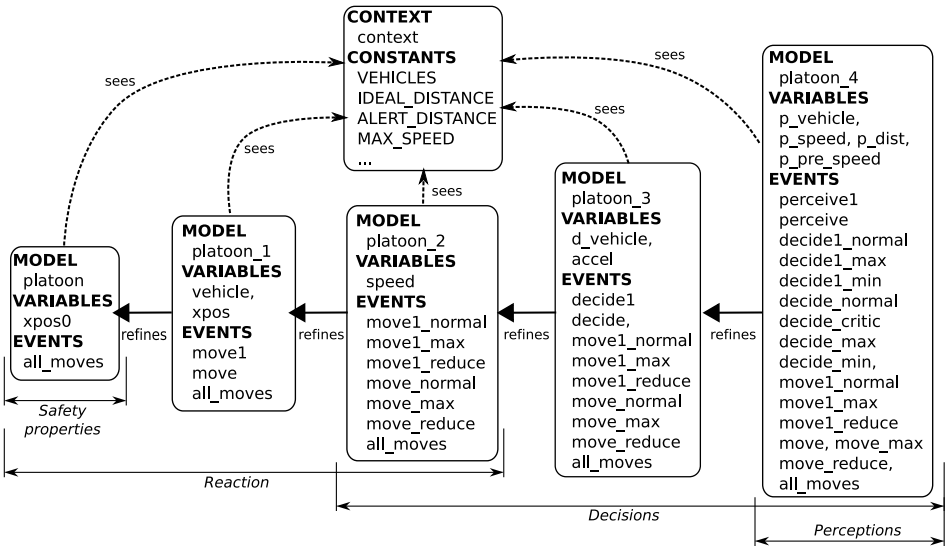

Figure 9.4: Event-B model of the platooning system [Lanoix 2008]

be useful for a quick look into the behavior, but not much more. The interesting animation was for `Platoon_4`.

9.3.1 Machine Platoon

This is the most abstract machine of the specification. It sees the context `Context`. The `Context` introduces the basic notion of a platoon, i.e., a set of vehicles, `VEHICLES`; the notion of critical distance between vehicles, `CRITICAL_DISTANCE`, which ensures the safe movement of the platoon; and initial positions of vehicles, `initial_xpos`.

The only event of the machine, `all_moves`, states that vehicles change their positions while respecting the critical distance among them. The functionality defined in this machine is too abstract to be animated.

9.3.2 Machine Platoon_1

This is the first refining machine in the specification which extends the protocol of movement in a platoon. It sees `Context_1` which extends `Context`, but it is empty. Specifiers probably have defined this context in order to be coherent with naming conventions used during the development process of the specification.

The machine `Platoon_1` is comprised of three events: `move_1` which defines the behavior of the leader of the platoon, `move` which defines the movement of the subsequent machines of the platoon, and `all_moves` which is the refinement of the abstract event with the same name. The animation of this machine is also uninteresting because of the high level of non-determinism and abstraction.

9.3.3 Machine Platoon_2

The machine `Platoon_2` sees the context `Context_2` which is the refinement of `Context_1`. This context introduces the notion of *speed* and *acceleration* into the model. Several new constants and axioms have been introduced into the context which in turn introduce the kinematics of a platooning system. The definition of the kinematics is comprised of complex mathematical functions and definitions which are non-animatable. The prime reason for their non-animatability is the complex definition of the functions which does not allow the assignment of a single start-up value to the constant. In fact, some of the functions are based on multiple definitions, each corresponding to a different case.

The first complex modification of the refinement was the definition of `new_xpos` function

$$\begin{array}{l}\forall xpos0, speed0, accel0. \\ ((xpos0 \in \mathbb{N} \land speed0 \in 0..MAX\_SPEED \land \\ accel0 \in MIN\_ACCEL..MAX\_ACCEL) \Rightarrow \\ \quad (new\_xpos(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 + speed0 + (accel0/2)))\end{array}$$

which models a kinematic law. It was used in some event guards in the form

$$nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$$

Using the heuristic 7, we rewrote the guards as

$$nxpos = xpos(vehicle) + speed(vehicle) + (magic\_accel/2))$$

Proof of application of the heuristic 7: The PO needs to be proved is $G_r \Rightarrow G$.

$nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$ (G)

The function new_xpos is defined in the context as:

$new\_xpos(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 + speed0 + (accel0/2)$

Inlining the definition of function into G with the corresponding local variables:

$nxpos = xpos(vehicle) + speed(vehicle) + (magic\_accel/2))$ (G_r)

Therefore, $G_r \Rightarrow G$. □

The most important complication came with another kinematic function `new_xpos_max` which is quite similar to `new_xpos`, except there is a case definition:

$$\begin{array}{l}\forall xpos0, speed0, accel0. \\ \quad ((xpos0 \in \mathbb{N} \land speed0 \in 0..MAX\_SPEED \land \\ \quad accel0 \in MIN\_ACCEL..MAX\_ACCEL) \Rightarrow \\ \quad\quad (accel0 = 0 \Rightarrow new\_xpos_max(xpos0 \mapsto speed0 \mapsto accel0) = \\ \quad\quad\quad xpos0 + MAX\_SPEED) \land \\ \quad\quad (accel \neq 0 \Rightarrow new\_xpos\_max(xpos0 \mapsto speed0 \mapsto accel0) = \\ \quad\quad\quad xpos0 + MAX\_SPEED - (((MAX\_SPEED - speed0) \times \\ \quad\quad\quad (MAX\_SPEED - speed0))/(2/accel0))))\end{array}$$

The events using `new_xpos_max` function had to be duplicated (heuristic 8), one with the guard `accel=0` and the other with its negation.

The prime example of such cases is the event `move1_max` which is shown in figure 9.6. The `guard3` of the original event calculates the new speed of vehicle as:

$$nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$$

The speed is then checked against the maximum allowed speed `guard4` and consequently a new position for the vehicle is determined in `guard5` as:

$$nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$$

To solve the issue, the cases defined to calculate `new_xpos_max` are broken down into two events, each catering for one particular case. The figure 9.7 shows the transformed `move1_max` event. The original and the transformed context `Context_2` that tells which functions have been relocated to machines are shown by figure 9.5.

Proof of application of the heuristic 8: The PO needs to be proved is

$$G_e(v) \Rightarrow \exists e'.e' \in Rel[\{e\}] \wedge G'_{e'}(v) \wedge (\forall e'.G'_{e'}(v) \Rightarrow G_e(v))$$

$$nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)(G_e)$$

The function new_xpos_max is defined as:

$$\begin{array}{l} If\ accel0 = 0 \Rightarrow \\ \quad new\_xpos\_max(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 + MAX\_SPEED \\ else\ if\ accel0 \neq 0 \Rightarrow \\ \quad new\_xpos\_max(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 + MAX\_SPEED - \\ \quad (((MAX\_SPEED - speed0) * (MAX\_SPEED - speed0))/(2/accel0)) \end{array}$$

Inlining the definition of function into G_e while splitting it into G' and G''

G' states:

$$\begin{array}{l} grd'\ magic\_accel \neq 0 \\ grd5\ nxpos = xpos(vehicle) + MAX\_SPEED - \\ \quad (((MAX\_SPEED - speed(vehicle)) * \\ \quad (MAX\_SPEED - speed(vehicle)))/(2 * magic\_accel)) \end{array}$$

G'' states:

$$\begin{array}{l} grd''\ magic\_accel = 0 \\ grd5\ nxpos = xpos(vehicle) + MAX\_SPEED \end{array}$$

Therefore, $G' \vee G'' \Rightarrow G_e(v)$. □

CONTEXT
Context2
EXTENDS
Context1
CONSTANTS
MAX_SPEED, MIN_ACCEL, MAX_ACCEL,
initial_speed , new_speed, new_xpos,
new_xpos_max, new_xpos_min
AXIOMS
typ01 MAX_SPEED ∈ N1
typ02 MAX_ACCEL ∈ N1
typ03 MIN_ACCEL ∈ INT

pro01 MIN_ACCEL < 0
pro02 initial_speed ∈ 1..VEHICLES →
0..MAX_SPEED
pro03 ∀ vehi0 .(vehi0∈1..VEHICLES ⇒ (∃ speed0.
(speed0 ∈ 0..MAX_SPEED ∧
initial_speed (vehi0) = speed0)))
pro04 new_speed ∈ (0..MAX_SPEED X
MIN_ACCEL..MAX_ACCEL) → INT
pro05 ∀ speed1, accel1 .
(speed1∈0..MAX_SPEED ∧ accel1∈
MIN_ACCEL?MAX_ACCEL ⇒
new_speed(speed1↦accel1) =
speed1 + accel1)
pro06 new_xpos ∈ (N X 0..MAX_SPEED X
MIN_ACCEL..MAX_ACCEL)→ N
pro07 ∀ xpos0,speed0, accel0 . ((xpos0 ∈ N ∧
speed0 ∈ 0..MAX_SPEED ∧
accel0 ∈ MIN_ACCEL..MAX_ACCEL) ⇒
(new_xpos(xpos0↦speed0↦accel0) =
xpos0 + speed0 + (accel0 / 2)))
pro08 new_xpos_max ∈ N X 0..MAX_SPEED X
MIN_ACCEL..MAX_ACCEL → N
pro09 ∀ xpos0,speed0, accel0 . (xpos0 ∈ N
∧ speed0 ∈ 0..MAX_SPEED ∧
accel0 ∈ MIN_ACCEL..MAX_ACCEL ⇒
((accel0 = 0 ⇒
new_xpos_max(xpos0↦speed0↦accel0)
= xpos0 + MAX_SPEED) ∧
(accel0 ≠ 0 ⇒
new_xpos_max(xpos0↦speed0↦accel0) =
xpos0 + MAX_SPEED −
(((MAX_SPEED − speed0) ∗
(MAX_SPEED−speed0))/(2∗accel0)))))
pro10 new_xpos_min ∈ N X 0..MAX_SPEED X
MIN_ACCEL..MAX_ACCEL → N
pro11 ∀ xpos0,speed0, accel0 . (xpos0 ∈ N ∧
speed0 ∈ 0..MAX_SPEED ∧
accel0 ∈ MIN_ACCEL..MAX_ACCEL ⇒
((accel0 = 0 ⇒
new_xpos_min(xpos0↦speed0↦accel0) =
xpos0) ∧ (accel0 ≠ 0 ⇒
new_xpos_min(xpos0↦speed0↦accel0) =
xpos0 − ((speed0 × speed0) /
(2 × accel0)))))
END

CONTEXT
Context2
EXTENDS
Context1
CONSTANTS
MAX_SPEED, MIN_ACCEL, MAX_ACCEL,
initial_speed ,

AXIOMS
typ01 MAX_SPEED ∈ N1
typ02 MAX_ACCEL ∈ N1
typ03 MIN_ACCEL ∈ INT

pro01 MIN_ACCEL < 0
pro02 initial_speed ∈ 1..VEHICLES →
0..MAX_SPEED
pro03 ∀ vehi0 .(vehi0∈1..VEHICLES ⇒
(∃ speed0 . (speed0 ∈ 0..MAX_SPEED ∧
initial_speed (vehi0) = speed0)))

END

Figure 9.5: The context `Context_2` before (left) and after (right) the application of the heuristic 7

```
  move1_max ≙
REFINES
  move1
ANY
  magic_accel, nspeed, nxpos
WHERE
  grd1 vehicle = 1
  grd2 magic_accel ∈ MIN_ACCEL..MAX_ACCEL
  grd3 nspeed = new_speed(speed(vehicle)↦magic_accel)
  grd4 nspeed > MAX_SPEED
  grd5 nxpos = new_xpos_max(xpos(vehicle)↦speed(vehicle)↦magic_accel)
WITH
  var1 magic_xpos_vehicle = nxpos
THEN
  act1 vehicle := vehicle+1
  act2 xpos( vehicle ) := nxpos
  act3 speed( vehicle ) := MAX_SPEED
END
```

Figure 9.6: The event `move1_max` before application of the heuristics 7 & 8

```
  move1_max ≙
REFINES
  move1
ANY
  magic_accel, nspeed, nxpos
WHERE
  grd1 vehicle = 1
  grd2 magic_accel ∈ MIN_ACCEL..MAX_ACCEL
  grd' magic_accel ≠ 0
  grd3 nspeed = new_speed(speed(vehicle)↦
       magic_accel)
  grd4 nspeed > MAX_SPEED
  grd5 nxpos = xpos(vehicle) +
         MAX_SPEED − (((MAX_SPEED −
         speed( vehicle )) × (MAX_SPEED −
         speed( vehicle ))) / (2 × magic_accel))
WITH
  var1 magic_xpos_vehicle = nxpos
THEN
  act1 vehicle := vehicle+1
  act2 xpos( vehicle ) := nxpos
  act3 speed( vehicle ) := MAX_SPEED
END
```

```
  move1_max_zero ≙
REFINES
  move1
ANY
  magic_accel, nspeed, nxpos
WHERE
  grd1 vehicle = 1
  grd2 magic_accel ∈ MIN_ACCEL..MAX_ACCEL
  grd'' magic_accel = 0
  grd3 nspeed = new_speed(speed(vehicle)↦
       magic_accel)
  grd4 nspeed > MAX_SPEED
  grd5 nxpos = xpos(vehicle) + MAX_SPEED
WITH
  var1 magic_xpos_vehicle = nxpos
THEN
  act1 vehicle := vehicle+1
  act2 xpos( vehicle ) := nxpos
  act3 speed( vehicle ) := MAX_SPEED
END
```

Figure 9.7: The event `move1_max` after application of the heuristics 7 & 8

9.3.4 Machine Platoon_3

This refinement decides the acceleration of the vehicles in the platoon. It sees the context `Context_3` which defines the initial acceleration of the vehicles. This context is simple and does not require any transformation.

In machine `Platoon_3`, we apply the heuristic 9 (removal of the invariant) to the following complex invariant:

$$
\begin{array}{l}
(d\_vehicle = VEHICLES + 1) \; or \; (\forall v.(v \in 2..d_vehicle - 1 \Rightarrow \\
\quad (\exists f1, f2. \\
\quad\quad (f1 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \wedge \\
\quad\quad\; f2 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \wedge \\
\quad\quad (f1(xpos(v-1) \mapsto speed(v-1) \mapsto accel(v-1)) - \\
\quad\quad\; f2(xpos(v) \mapsto speed(v) \mapsto accel(v)) > \\
\quad\quad\quad CRITICAL\_DISTANCE)))))
\end{array}
$$

The event `decide` of the machine, which actually decides the acceleration of the members of platoon other than the leader vehicle, had to be transformed in order to animate. We applied heuristic 8 and decomposed the event into four subsequent events, each catering for one particular case. The event `decide` is shown by figure 9.8. Its transformations are shown by figures 9.9 and 9.10 respectively.

```
  decide ≙
ANY
  magic_accel
WHERE
  grd1 vehicle = 1
  grd2 d_vehicle ∈ 2..VEHICLES
  grd3 magic_accel..MIN_ACCEL..MAX_ACCEL
  grd4 ∃g1,g2. (g1∈ {new_xpos, new_xpos_max, new_xpos_min} ∧
       g2 ∈ {new_xpos, new_xpos_max, new_xpos_min} ∧
       (g1(xpos(d_vehicle−1)↦speed(d_vehicle−1)↦accel(d_vehicle−1))
       − g2(xpos(d_vehicle)↦speed(d_vehicle)↦magic_accel) > CRITICAL_DISTANCE))
THEN
  act1 d_vehicle := d_vehicle+1
  act2 accel(d_vehicle) := magic_accel
END
```

Figure 9.8: The event `decide` before the application of the heuristics 7 & 8

The original machine contains 10 events which are decomposed into 17 events in the transformed machine.

9.3.5 Machine Platoon_4

In this refinement, the controllers perceive (perception step) before they decide for an acceleration by applying the decision law. This is done in `Context_4` which defines a new constant `new_accel`. Because of its complex definition, it can not

```
    decide_1 ≙
ANY
  magic_accel
WHERE
  grd1  vehicle  = 1
  grd2  d_vehicle ∈ 2..VEHICLES
  grd3  magic_accel ∈ MIN_ACCEL..MAX_ACCEL
  grd'  magic_accel ≠ 0
  grd'' accel(d_vehicle−1) ≠ 0
  grd4  ∃ g1, g2.
        (g1∈ {(xpos(d_vehicle−1) +
        speed(d_vehicle−1) +
        (accel(d_vehicle−1)/2)),
        (xpos(d_vehicle−1) +
        MAX_SPEED − (((MAX_SPEED −
        speed(d_vehicle−1)) × (MAX_SPEED −
        speed(d_vehicle−1))) /
        (2 × accel(d_vehicle−1)))),
        (xpos(d_vehicle−1) −
        (((speed(d_vehicle−1) ∗
        speed(d_vehicle−1)) /
        (2 × accel(d_vehicle−1))))} ∧
        g2∈{(xpos(d_vehicle) + speed(d_vehicle)+
        (magic_accel/2)), (xpos(d_vehicle) +
        MAX_SPEED − (((MAX_SPEED −
        speed(d_vehicle)) ∗
        (MAX_SPEED − speed(d_vehicle))) /
        (2 × magic_accel))), (xpos(d_vehicle) −
        (((speed(d_vehicle) × speed(d_vehicle)) /
        (2 × magic_accel))))} ∧
        g1− g2 > CRITICAL_DISTANCE)
THEN
  act1 d_vehicle := d_vehicle+1
  act2 accel(d_vehicle) := magic_accel
END
```

```
  decide_2 ≙
ANY
  magic_accel
WHERE
  grd1  vehicle  = 1
  grd2  d_vehicle ∈ 2..VEHICLES
  grd3  magic_accel∈MIN_ACCEL..MAX_ACCEL
  grd'  magic_accel ≠ 0
  grd'' accel(d_vehicle−1) = 0
  grd4  ∃ g1, g2 .
        (g1 ∈ {(xpos(d_vehicle−1) +
        speed(d_vehicle−1)), (xpos(d_vehicle−1)+
        MAX_SPEED),
        speed(d_vehicle−1)} ∧

        g2 ∈ {(xpos(d_vehicle)+speed(d_vehicle)+
        (magic_accel/2)), (xpos(d_vehicle) +
        MAX_SPEED − (((MAX_SPEED −
        speed(d_vehicle)) × (MAX_SPEED −
        speed(d_vehicle))) / (2 × magic_accel))),
        (xpos(d_vehicle) − (((speed(d_vehicle) ∗
        speed(d_vehicle)) / (2 × magic_accel))))}

        g1 − g2 > CRITICAL_DISTANCE)
THEN
  act1 d_vehicle := d_vehicle+1
  act2 accel(d_vehicle) := magic_accel
END
```

Figure 9.9: The event `decide` after the application of the heuristics 7 & 8

```
    decide_3 ≙
ANY
  magic_accel
WHERE
  grd1  vehicle  = 1
  grd2  d_vehicle ∈ 2..VEHICLES
  grd3  magic_accel..MIN_ACCEL..MAX_ACCEL
  grd'  magic_accel = 0
  grd'' accel(d_vehicle−1) ≠ 0
  grd4  ∃ g1, g2 .
        (g1 ∈ {(xpos(d_vehicle−1)+
        speed(d_vehicle−1)+
        (accel(d_vehicle−1)/2)),
        (xpos(d_vehicle−1) + MAX_SPEED −
        (((MAX_SPEED − speed(d_vehicle−1)) ∗
        (MAX_SPEED − speed(d_vehicle−1))) /
        (2 × accel(d_vehicle−1)))),
        (xpos(d_vehicle−1) −
        (((speed(d_vehicle−1) ∗
        speed(d_vehicle−1)) /
        (2 × accel(d_vehicle−1)))))} ∧
        g2 ∈ {(xpos(d_vehicle) +
        speed(d_vehicle−1)),
        (xpos(d_vehicle) + MAX_SPEED),
        speed(d_vehicle)} ∧
        g1 − g2 > CRITICAL_DISTANCE)
THEN
  act1 d_vehicle := d_vehicle+1
  act2 accel(d_vehicle) := magic_accel
END
```

```
  decide_4 ≙
ANY
  magic_accel
WHERE
  grd1  vehicle  = 1
  grd2  d_vehicle ∈ 2..VEHICLES
  grd3  magic_accel..MIN_ACCEL..MAX_ACCEL
  grd'  magic_accel = 0
  grd'' accel(d_vehicle−1) = 0
  grd4  ∃ g1,g2 .
        (g1 ∈ {(xpos(d_vehicle−1)+
        speed(d_vehicle−1)),
        (xpos(d_vehicle−1) + MAX_SPEED),
        speed(d_vehicle−1)} ∧
        g2 ∈ {(xpos(d_vehicle) +
        speed(d_vehicle−1)),
        (xpos(d_vehicle) + MAX_SPEED),
        speed(d_vehicle)} ∧
        g1 − g2 > CRITICAL_DISTANCE)
THEN
  act1 d_vehicle := d_vehicle+1
  act2 accel(d_vehicle) := magic_accel
END
```

Figure 9.10: The event `decide` after the application of the heuristics 7 & 8

be assigned startup value so it must be transported to the machine `platoon_4` by applying heuristics 7 and 8. The original and the resulted contexts are shown by figure 9.11.

CONTEXT
Context_4
EXTENDS
Context_3
CONSTANTS
IDEAL_SPEED, ideal_distance, new_accel
AXIOMS
typ01 IDEAL_SPEED ∈ 0..MAX_SPEED
typ02 ideal_distance ∈ 0..MAX_SPEED → N
typ03 new_accel ∈
(INT X 0..MAX_SPEED X
0..MAX_SPEED) → INT

pro01 IDEAL_SPEED < MAX_SPEED
pro02 ∀ speed0 . (speed0 ∈ 0..MAX_SPEED ⇒
ideal_distance(speed0) =
CRITICAL_DISTANCE + speed0)
pro03 ∀ p_dist1,p_speed1,p_pre_speed1 .
(p_dist1 ∈ INT ∧ p_speed1 ∈
0..MAX_SPEED ∧ p_pre_speed1 ∈
0..MAX_SPEED ⇒
new_accel(p_dist1↦p_speed1↦
p_pre_speed1) = p_dist1 −
ideal_distance(p_speed1) +
p_pre_speed1 − p_speed1)
pro04 ∀ speed . (speed∈ 0..MAX_SPEED ⇒
ideal_distance(speed) ≥
CRITICAL_DISTANCE)
END

CONTEXT
Context_4
EXTENDS
Context_3
CONSTANTS
IDEAL_SPEED, ideal_distance
AXIOMS
typ01 IDEAL_SPEED ∈ 0..MAX_SPEED
typ02 ideal_distance ∈ 0..MAX_SPEED → N

pro01 IDEAL_SPEED < MAX_SPEED
pro02 ∀ speed0 . (speed0 ∈ 0..MAX_SPEED ⇒
ideal_distance(speed0) =
CRITICAL_DISTANCE + speed0)

pro04 ∀ speed . (speed∈ 0..MAX_SPEED ⇒
ideal_distance(speed) ≥
CRITICAL_DISTANCE)
END

Figure 9.11: The context `Context_4` before (left) and after (right) the application of the heuristics 7 & 8

The consequence of the above transformation was that the event `decide_normal` had to be split into four subsequent events, just like the event `decide`. Each new event is the refinement of the `decide` event shown by figures 9.9 and 9.10. The trickiest situation was actually the two guards

$$
\begin{aligned}
&grd5\ naccel = new\_accel(p\_dist(d\_vehicle) \mapsto p\_speed(d\_vehicle) \mapsto \\
&\quad p\_pre\_speed(d\_vehicle)) \\
&grd3\ \exists g1, g2. \\
&\quad (g1 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \wedge \\
&\quad g2 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \wedge \\
&\quad (g1(xpos(d\_vehicle-1) \mapsto speed(d\_vehicle-1) \mapsto accel(d\_vehicle-1)) - \\
&\quad g2(xpos(d\_vehicle) \mapsto speed(d\_vehicle-1) \mapsto naccel) \\
&\qquad > CRITICAL\_DISTANCE))
\end{aligned}
$$

Out of three functions used in `grd3`, two are defined by case but all are used twice with different arguments. We were lucky that the cases for the function are the

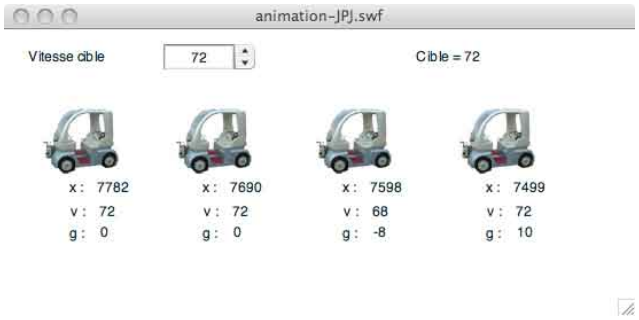

Figure 9.12: Animation of the platooning system

same (acceleration null or not), so we had only four duplications.

The last step was to setup a small graphical interface in Flash so that we could have a synthetic view of the moving platoon. Technically, we had to introduce a new refinement so that "observation" variables could be set (heuristic 10). The reason comes from the limited data types that Brama currently communicate to the Flash server: integers and boolean. As the model uses discrete functions to record current information of the vehicles, we had to split them into different variables. They are all concentrated in the `move_all` event. The end result is shown by figure 9.12.

As can be seen on the interface, the cruising speed of the platoon can be controlled. Setting this control required us to modify the specification in which the cruise speed was initially defined as a constant. From our point of view, the initial specification was unrealistic in this respect and the animation allowed us to spot this, small, inconsistency.

After the application of the heuristics, the transformed machine `Platoon_4` contains 20 events whereas the original machine has 16 events.

During the animation it has been observed that there is an oscillation in the platoon, i.e., the propagation of a wave inside the platoon without stabilization. The last vehicles of the platoon had to adjust their acceleration frequently while the ones in the front run smoothly. Animation shows that this specification needs to improve on this account as this is an undesirable feature.

9.4 Summary

In this chapter, we have presented two Event-B specifications, a transport domain model and a platooning system, which have been animated by applying our proposed transformational heuristics. In order to show their effects, we have presented both before and after states of the specification. The applications of the heuristics are justified in the form of a formal proof. The animation of the transformed specifications did not only show the behavior of the model, but also helped to spot some specification errors.

Part IV

EPILOGUE

Chapter 10

Stepwise validation of formal specifications

Contents

10.1 Introduction

With the help of well-defined syntax and semantics, formal specifications concisely express software requirements. However, due to their complex structures and mathematical notions, 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 requirement engineering phase. However, in the form of animation, we have a solution for this problem but then another question arises: *when* shall we start validating?

Verification also raises a similar question. In test-based verification procedures, we need to wait until systems are implemented and running. As the cost of correcting errors or misunderstandings in requirements increases dramatically during the development life-cycle, it makes a lot of sense to verify and validate as early as possible.

The pivotal concept of formal methods such as Event-B is the notion of refinement and its relation to correctness. The assessment of the correctness of a piece of code, its verification, is no more a unique big process step but is broken down into small pieces along with the whole development process. The proof of correctness is the sum of the proofs of small assertions (invariant preservation, well-formedness, existence of abstraction function, etc.) associated to each refinement. Problems are then detected early. While a formal refinement process does not preclude a testing

activity, the latter will be more focused on finding true implementation errors, not requirement problems.

We propose a framework, VTA, for the rigorous validation of formal specifications based on the principle of verification. We break the validation of formal specifications into smaller assessments and integrate it to their stepwise development. VTA, powered by the techniques of verification, transformation and animation, allows specifiers to validate their models early and cheaply.

Our aim to introduce validation into refinement-based processes yields results on two levels. First, early detection of problems in the requirements (say, misunderstanding about a certain behavior) should be easier and less expensive to correct. Second, users can be involved into the development right from the start.

The chapter is organized as follows: Section 10.2 presents our proposed validation framework VTA. Section 10.3 reflects our perspective towards the technique of animation. Finally, section 10.4 summarizes the chapter.

10.2 VTA: The framework

VTA (Verify-Transform-Animate) is a framework for the stepwise validation of formal specifications. In this framework, we integrate the technique of animation with each observation level of the specification to break its validation into smaller assessments in order to ensure that it represents actual requirements of customers.

In VTA, before proceeding with the actual validation of the specification, we first verify it. Later, if the specification contains some elements which are non-animatable, they are transformed with the help of the heuristics in order to achieve animation. The application of these heuristics may "downgrade" the specification to be "non-provable." Then, running the animation may uncover some mistakes. These entail the modification of the initial specification which then must be verified and transformed again for proceeding with the validation.

VTA is shown by figure 10.1 and consists of the following steps:

1. Start from a fully verified specification. This step is mandatory.

2. Transform each non-animatable element of the specification by:

   (a) Choosing the matching heuristic from the list

   (b) Checking that its applicability conditions hold

   (c) Proving its application

   (d) Applying the heuristic

3. Animate for validation. If an anomalous behavior is encountered, modify the initial specification and restart from step one.

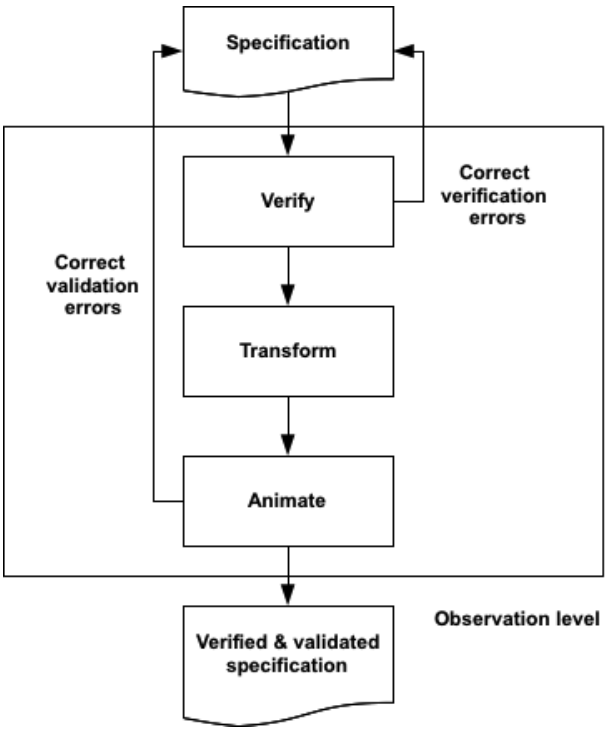

Figure 10.1: VTA: The stepwise validation framework

10.2.1 Verification step

The step 1 of our proposed validation framework is based on the verification of specifications. Our belief is that there is no point in validating a specification which could not be verified! Such a specification is a dead-end as far as formal development is concerned.

It is important to note that the order between verification and validation in VTA is the reverse of what a development relying on tests would use. In the later case, there is no point in engaging a costly series of tests on a piece of code which does not fulfill the needs of users. We give verification preeminence over validation mainly for two reasons. First, it provides us with a reasonable safeguard. Second, and more importantly, it allows us to justify some heuristics with sound arguments.

Let us consider two heuristics, for instance. The first calls for the removal of an invariant (heuristic 9). This is safe because (1) invariant does not modify behaviors (it is only observed) and (2) the proof obligations related to maintaining the invariant have been successfully discharged. The second calls for the replacement of a set defined through complex properties of its elements by a simpler super-set (heuristic 3). Provided we have exerted great care while feeding the animation with values which conform to the "complex" set definition, the transformation is safe because proof obligations have been discharged under the assumption that the values belonged to the "complex" set, and (i) either the values are only used (they are constants), and so properties are trivially maintained, or (ii) the values are

computed, but then at least one of the discharged proof-obligation was about the belonging of the computed value to the "complex" set. Though less direct, the justification for other heuristics also rely on the fact that they are applied to verified texts.

10.2.2 Transformation step

As soon as all proof obligations have been discharged, we start animating the specification. This animation process is often struck either due to some shortcomings of animators, such as unsupported features of the language, or due to some non-executable elements, such as non-constructive definitions, which are used to specify the behavior. This is the point where we introduce our proposed heuristics to VTA.

Whenever we discover any element in the specification which is non-executable, we inspect the problem and try to match the case with the list of heuristics. This inspection and matching practice includes checking if the same application condition holds as defined by the heuristic *pattern* 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 proof-obligation) or by a rigorous argument that application of heuristic would not change the behavior of the specification.

10.2.3 Animation step

Once transformations have been applied, the specification belongs to the animatable class. Animation would demonstrate the behavior of the specification. If the demonstrated behavior is as per expectations then we have the verified and 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 the actual requirements.

10.3 Animation: A reflection

Literature discusses validation as a heterogeneous process which is based on a variety of techniques to ensure that analysts have understood the requirements of stakeholders correctly and have not introduced any error while specifying them. During a typical validation process, requirements are checked for their consistency, unambiguity, and completeness. These qualities of requirements are established by customers or with the help of Software Requirement Specifications (SRS) documents. Requirement coverage matrices are often used to describe the degree to which the requirements of particular specification have been validated. Different testing oracles are also employed to prove or disprove whether some validation test has passed or failed. Definition and categorization of different outputs of system against which the specification can be compared also help this cause. Extensive test

plans, either individual test cases or collaborative test suites are also a part of the standard validation process. In short, the goal of all these techniques, along with animation, is to provide a feedback link to customers to give them a chance to check whether the requirements specified are correct or not.

During our experience, we have discovered that animation is a multi-disciplinary technique which can be used for a variety of activities during software development. Primarily, animation is used as a quality-assurance activity, i.e., to gain confidence in specifications. It can also be used as prototyping. The benefit over here is that we can execute the specification without translating it into code. 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 the 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.

10.4 Summary

In formal methods, a key idea to assess that an implementation is correct is to break its verification into smaller proofs associated with each refinement step. Likewise, we integrate animation into the stepwise development of specifications in order to break their validation into smaller assessments. Customers, like this, can be involved into the development right from the start and problems can be detected right on the spot.

In this chapter, we have introduced the innovative VTA framework which advocates the frequent use of animation into the rigorous software development cycle. Combined with verification and our proposed *low-cost* transformations, it helps reducing the overall cost and time of the validation process. In the end of the chapter, we have presented some reflections on the utility of the animation.

Chapter 11

Conclusion & future work

Event-B is designed around the idea that a piece of code can be "correct per construction." Due to its strong emphasize on the notion of correctness using proofs, its refinement-based idea of development and its effective tool support, it is a good candidate to specify safety-critical systems.

As a complex and reasonably well-known domain, land transportation is a good choice to test and assess specification methods and processes. We have conducted a thorough analysis of the use of Event-B for engineering domains using this example. We have shown what a domain model should be comprised of, how to refine it and how it should be verified and validated.

Using a language which is not conceived particularly for domain engineering was a challenging task. We stumbled upon some of Event-B's shortcomings, for example, lack of expression of temporal and ordering constraints. The point is that 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 [Schneider 2010] or event refinement diagrams [Butler 2009], can be made practical is still an open issue. Answers are beginning to appear. We just hope they can be used soon.

Apart from this, we have found Event-B an adequate language for domain engineering. Its general philosophy is well-suited to our purpose. The notions of events and non-determinism has allowed us easy modeling of independent vehicles without any assumption other than their common property: they move. The strong safety constraints we have considered are also easily modeled. Modeling of other properties, such as, collision avoidance, timing, etc. also did not pose great difficulties. All was done through standard refinement techniques.

We would appreciate to see tools evolving in the direction of richer visualization of the specifications. Our notes about observation levels, flat linear structures, parallel refinements, or composition of refinements 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.

Event-B provides us with an efficient set of theorem provers, however what we have discovered is that proofs alone are not sufficient to produce a "good" software: we need also to execute, i.e., to test, the software. Two main reasons justify this proposition. Some properties, notably temporal, are *virtually impossible* to state

with constructs such as states, invariants, or events which this language provides us. We need to ensure that the specification is an adequate model of the problem we want to solve. Software must be verified and validated.

Waiting to have an executable program to begin the validation leads to the same difficulties as proving a program against its specification: costly, very complex, soon unmanageable. The strategy which works with the verification of specifications built by a stepwise refinement process could also work for validation. We should be able to enrich the development process by adding validation activities to each refinement step. We have then proposed an animation-based rigorous validation framework for software specifications.

One limiting factor of the technique of animation is that not all specifications are animatable. Qualities sought after for a well-written specification, such as abstractness, non-determinism, non-constructive definitions, are contradictory with what is required for effective computation. Specification, thus, can be classified into provable, animatable, or both.

Good thing about animation is that in many cases we can "downgrade" a proven, non-animatable, specification into a "behaviorally equivalent" animatable, but non-provable, specification. We have then proposed several transformations to realize this idea. The validity of such a technique depends on the semantics of the transformations. We have developed ad-hoc semantics based on the behavior of the model. The preservation property we have expressed can be summed up as: "whatever we observe on the execution of the transformed specification would have been observed on the proven specification."

We are assured that applications of the heuristics are independent of each other, i.e., application of one heuristic does not contradict or depend upon another. In future, we intend to formally prove this along with the inclusion of more heuristics into the list as modeling and validation of new properties of the domain is expected. We also plan to check their compatibility with animators other than Brama, such as ProB.

Despite having transformation rules at our disposal, sometimes animators still fail to execute a specification. For such cases, we expect that translation of Event-B text into an executable programming language can be a reasonable fall-back strategy. This can also be seen as a proposed future work.

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). The boring parts of the transformation do not contain overly complex text manipulation and can easily be implemented in the form of a plugin for Rodin.

Bibliography

[Abrial 1996] J.-R. Abrial. The B book. Cambridge University Press, 1996. (Cited on pages 18 and 43.)

[Abrial 1998] Jean-Raymond Abrial and Louis Mussat. *Introducing Dynamic Constraints in B.* In B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method, pages 83–128, London, UK, 1998. Springer-Verlag. (Cited on page 45.)

[Abrial 2007] J.-R. Abrial and S. Hallerstede. *Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B.* Fundamenta Informaticae, vol. 77, no. 1-2, pages 1–28, 2007. (Cited on pages 37, 41 and 45.)

[Abrial 2010] Jean-Raymond Abrial. Modeling in event-b: System and software engineering. Cambridge University Press, 2010. (Cited on page 1.)

[Ahmad 2004] F. Ahmad and U. Aziz. *A survey of domain analysis techniques and domain reuse in Pakistan.* In Multitopic Conference, 2004. Proceedings of INMIC 2004. 8th International, pages 434 – 439, 2004. (Cited on page 11.)

[Alexander 2003] I. Alexander. *Misuse Cases: Use Cases with Hostile Intent.* IEEE Software, vol. 20, pages 58–66, 2003. (Cited on page 21.)

[Alexander 2004] I. Alexander. *Negative Scenarios and Misuse Cases.* Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle, pages 119–139, 2004. (Cited on page 21.)

[Aziz 2009] Benjamin Aziz, Alvaro E. Arenas, Juan Bicarregui, Christophe Ponsard and Philippe Massonet. *From Goal-Oriented Requirements to Event-B Specifications.* In First Nasa Formal Method Symposium, California, US, 2009. (Cited on page 44.)

[Baille 1999] Gérard Baille, Philippe Garnier, Hervé Mathieu and Pissard-Gibollet Roger. *Le cycab de l'INRIA Rhônes-Alpes.* Technical Report RT-0229, INRIA – Rhônes-Alpes, 1999. (Cited on page 48.)

[Ball 2004] Thomas Ball, Byron Cook, Vladimir Levin and Sriram K. Rajamani. *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft.* In Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, pages 1–20, 2004. (Cited on page 30.)

[Banach 1998] Richard Banach and M. Poppleton. *Retrenchment: An Engineering Variation on Refinement.* In Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method, pages 129–147, London, UK, 1998. Springer-Verlag. (Cited on page 45.)

[Beck 1989] K. Beck and W. Cunningham. *A laboratory for teaching object-oriented thinking*. OOPSLA' 89, Special Issue of ACM SIGPLAN Notices, vol. 24, pages 1–6, 1989. (Cited on page 21.)

[Bendisposto 2008] J. Bendisposto, M. Leuschel, O. Ligot and M. Samia. *La validation de modèles Event-B avec le plug-in ProB pour RODIN*. Technique et Science Informatiques, vol. 27, no. 8, pages 1065–1084, 2008. (Cited on page 44.)

[Bicarregui 2008] Juan Bicarregui, Alvaro Arenas, Benjamin Aziz, Philippe Massonet and Christophe Ponsard. *Towards Modelling Obligations in Event-B*. In Proceedings of the 1st international conference on Abstract State Machines, B and Z, ABZ '08, pages 181–194. Springer-Verlag, 2008. (Cited on page 46.)

[Bisztray 2008] Dénes Bisztray, Reiko Heckel and Hartmut Ehrig. *Verification of architectural refactorings by rule extraction*. In FASE'08/ETAPS'08: Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering, pages 347–361, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 44.)

[Bjesse 2005] Per Bjesse. *What is formal verification?* SIGDA Newsl., vol. 35, no. 24, 2005. (Cited on page 29.)

[Bjørner 2004] Dines Bjørner. *TRain: The Railway domain - A "Grand Challenge" for Computing Science & Transportation Engineering*. In Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, Toulouse, France, pages 607–612, 2004. (Cited on page 14.)

[Bjørner 2005] Dines Bjørner. *A Cloverleaf of Software Engineering*. In Software Engineering and Formal Methods, International Conference on, pages 75–85, Los Alamitos, CA, USA, 2005. IEEE Computer Society. (Cited on page 14.)

[Bjørner 2006] D. Bjørner. Software Engineering 3: Domains, Requirements, and Software Design (Texts in Theoretical Computer Science, an EATCS Series). Springer-Verlag, 2006. (Cited on page 14.)

[Bjørner 2007] D. Bjørner. *Development of Transportation Systems*. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA), 2007. (Cited on pages 1 and 14.)

[Bjørner 2008a] Dines Bjørner. *Domain Engineering*. In In BCS FACS Seminars, Lecture Notes in Computer Science, the BCS FAC Series (eds. Paul Boca and Jonathan Bowen, pages 1–42. Springer, 2008. (Cited on page 14.)

[Bjørner 2008b] Dines Bjørner. *From Domain to Requirements*. In Concurrency, Graphs and Models, pages 278–300. Springer, 2008. (Cited on page 14.)

[Bjørner 2009a] D. Bjørner. DOMAIN ENGINEERING: Technology Management, Research and Engineering. JAIST, 2009. (Cited on page 14.)

[Bjørner 2009b] Dines Bjørner. *Rôle of Domain Engineering in Software Development - Why Current Requirements Engineering Is Flawed !* In Ershov Memorial Conference, Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, pages 2–34, 2009. (Cited on pages 10 and 14.)

[Bjørner 2010] D. Bjørner. *Domain science and engineering from computer science to the sciences of informatics. Part I: Engineering.* Cybernetics and Systems Analysis, vol. 46, pages 609–623, 2010. (Cited on page 14.)

[Boehm 1978] Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. MacLeod and Michael J. Merritt. Characteristics of software quality. North-Holland Publishing Company, 1978. (Cited on page 27.)

[Booch 1998] G. Booch, J. Rumbaugh and I. Jacobson. The unified modeling language user guide. Addison-Wesley, 1998. (Cited on page 20.)

[Booth 1967] Taylor L. Booth. Sequential machines and automata theory. John Wiley & Sons, 1967. (Cited on page 18.)

[Bowen 1995a] Jonathan P. Bowen and Michael G. Hinchey. *Seven More Myths of Formal Methods.* IEEE Softw., vol. 12, no. 4, pages 34–41, 1995. (Cited on page 23.)

[Bowen 1995b] Jonathan P. Bowen and Michael G. Hinchey. *Ten Commandments of Formal Methods.* Computer, vol. 28, pages 56–63, 1995. (Cited on page 24.)

[Bowen 2006] Jonathan P. Bowen and Michael G. Hinchey. *Ten Commandments of Formal Methods ...Ten Years Later.* Computer, vol. 39, pages 40–48, 2006. (Cited on page 24.)

[Bozga 1998] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis and Sergio Yovine. *Kronos: A Model-Checking Tool for Real-Time Systems.* In CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification, pages 546–550, London, UK, 1998. Springer-Verlag. (Cited on page 30.)

[Brooks 1987] Frederick P. Brooks Jr. *No Silver Bullet Essence and Accidents of Software Engineering.* Computer, vol. 20, no. 4, pages 10–19, 1987. (Cited on page 25.)

[Bryans 2010] J. W. Bryans, J. S. Fitzgerald, A. Romanovsky and A. Roth. *Patterns for Modelling Time and Consistency in Business Information Systems.* In Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS '10, pages 105–114, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on page 45.)

[Bryant 1986] Randal E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Trans. Comput., vol. 35, no. 8, pages 677–691, 1986. (Cited on page 30.)

[Budde 1990] Reinhard Budde and Heinz Ziillighoven. *Prototyping Revisited*. In Proceedings of the IEEE International Conference on Computer Systems and Software Engineering (COMPEURO'90), Tel-Aviv, Israel, 1990. IEEE Computer Society. (Cited on page 32.)

[Butler 1996] Michael J. Butler. *Stepwise refinement of communicating systems*. Sci. Comput. Program., vol. 27, pages 139–173, September 1996. (Cited on page 41.)

[Butler 2002] Michael Butler. *A System-Based Approach to the Formal Development of Embedded Controllers for a Railway*. Design Automation for Embedded Systems, vol. 6, pages 355–366, 2002. (Cited on page 44.)

[Butler 2009] Michael Butler. *Decomposition Structures for Event-B*. In Proceedings of the 7th International Conference on Integrated Formal Methods, IFM '09, pages 20–38, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on pages 45 and 143.)

[Camilleri 1997] Albert John Camilleri. *A Hybrid Approach to Verifying Liveness in a Symmetric Multi-Processor*. In TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, pages 49–67, London, UK, 1997. Springer-Verlag. (Cited on page 24.)

[Cansell 2007] D. Cansell, D. Mery and J. Rehm. *Time Constraint Patterns for Event B Development*. In J. Julliand and O. Kouchnarenko, editeurs, 7th International Conference of B Users, volume 4355 of *LNCS*, pages 140–154. Springer Verlag, 2007. (Cited on pages 45, 52 and 74.)

[Chen 1976] Peter Pin-Shan Chen. *The entity-relationship model—toward a unified view of data*. ACM Trans. Database Syst., vol. 1, no. 1, pages 9–36, 1976. (Cited on page 18.)

[Christel 1992] Michael G. Christel and Kyo C. Kang. *Issues in Requirements Elicitation*. Rapport technique CMU/SEI-92-TR-012, Carnegie Mellon University, September 1992. (Cited on page 17.)

[Clarke 1982] Edmund M. Clarke and E. Allen Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In Logic of Programs, Workshop, pages 52–71, London, UK, 1982. Springer-Verlag. (Cited on page 30.)

[Clarke 1996] Edmund M. Clarke and Jeannette M. Wing. *Formal methods: state of the art and future directions*. ACM Comput. Surv., vol. 28, no. 4, pages 626–643, 1996. (Cited on pages 30 and 31.)

[Cle 2009] Clearsy. *User Manual of Atelier B, version 4.0*, 2009. (Cited on page 44.)

[Cleaveland 1993] Rance Cleaveland, Joachim Parrow and Bernhard Steffen. *The concurrency workbench: a semantics-based tool for the verification of concurrent systems*. ACM Trans. Program. Lang. Syst., vol. 15, no. 1, pages 36–72, 1993. (Cited on page 30.)

[Clemons 1985] Eric Clemons and Arnold Greenfield. *The Sage System Architecture: A System for the Rapid Development of Graphics Interfaces for Decision Support*. IEEE Computer Graphics and Applications, vol. 5, pages 38–50, 1985. (Cited on page 33.)

[Colin 2008a] S. Colin, A. Lanoix, O. Kouchnarenko and J. Souquières. *Towards Validating a Platoon of Cristal Vehicles using CSP‖B*. In J. Meseguer and G. Rosu, editeurs, 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008), numéro 5140 de LNCS, pages 139–144. Springer-Verlag, July 2008. (Cited on page 124.)

[Colin 2008b] S. Colin, A. Lanoix, O. Kouchnarenko and J. Souquières. *Using CSP‖B Components: Application to a Platoon of Vehicles*. In 13th International ERCIM Wokshop on Formal Methods for Industrial Critical Systems (FMICS 2008), LNCS. Springer-Verlag, September 2008. (Cited on page 124.)

[Cousot 1977] Patrick Cousot and Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, New York, NY, USA, 1977. ACM. (Cited on page 95.)

[Czarnecki 2000] Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. (Cited on pages 10 and 13.)

[Dardenne 1993] Anne Dardenne, Axel van Lamsweerde and Stephen Fickas. *Goal-directed requirements acquisition*. Science of Computer Programming, vol. 20, no. 1-2, pages 3–50, 1993. (Cited on page 20.)

[Daviet 1996] P. Daviet and M. Parent. *Longitudinal and Lateral Servoing of Vehicles in a Platoon*. In Proceeding of the IEEE Intelligent Vehicles Symposium, pages 41–46, 1996. (Cited on page 123.)

[Davis 1990] AM Davis. *System testing: Implications of requirements specifications*. Information and Software Technology, vol. 32, no. 6, pages 407 – 414, 1990. (Cited on page 14.)

[Dowek 1993] Gilles Dowek, Amy Felty, Hugo Herbelin, Gerard Huet, Catherine Parent, Christine Paulin Mohring, Benjamin Werner and Chetan Murthy.

The Coq proof assistant user's guide : version 5.8. Research Report RT-0154, INRIA, 1993. (Cited on page 31.)

[Erasmy 1995] François Erasmy and Emil Sekerinski. *Raise: A Rigorous approach using stepwise refinement.* In Claus Lewerentz and Thomas Lindner, editeurs, Formal Development of Reactive Systems, volume 891 of *Lecture Notes in Computer Science*, pages 277–293. Springer Berlin / Heidelberg, 1995. (Cited on page 43.)

[Essamé 2004] Didier Essamé. *Handling Safety Critical Requirements in System Engineering Using the B Formal Method.* In Maritta Heisel, Peter Liggesmeyer and Stefan Wittmann, editeurs, Computer Safety, Reliability, and Security, volume 3219 of *Lecture Notes in Computer Science*, pages 115–115. Springer Berlin / Heidelberg, 2004. (Cited on page 44.)

[Fagan 1976] M. E. Fagan. *Design and code inspections to reduce errors in program development.* IBM System Journal, vol. 15, no. 3, 1976. (Cited on page 33.)

[Ferr'e 1999] X. Ferr'e and S. Vegas. *An Evaluation of Domain Analysis Methods.* In In 4th CAiSE/IFIP8.1 International Workshop in Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD99, 1999. (Cited on page 11.)

[Finkelstein 1996] A. Finkelstein and I. Sommerville. *The Viewpoints FAQ.* Software Engineering Journal, vol. 11, pages 2–4, 1996. (Cited on page 20.)

[Futatsugi 1985] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud and José Meseguer. *Principles of OBJ2.* In POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 52–66, New York, NY, USA, 1985. ACM. (Cited on page 43.)

[Gemino 2004] Andrew Gemino. *Empirical comparisons of animation and narration in requirements validation.* Requirment Engineering, vol. 9, no. 3, pages 153–168, 2004. (Cited on page 31.)

[Gondal 2009] Ali Gondal, Michael Poppleton and Colin Snook. *Feature Composition - Towards product lines of Event-B models.* In 1st International Workshop on Model-Driven Product Line Engineering, pages 18–25, Twente, The Netherlands, 2009. (Cited on pages 45 and 79.)

[Gordon 1993] M. J. C. Gordon and T. F. Melham. Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press, 1993. (Cited on page 31.)

[Gørtz 1994] Jesper Gørtz. *Specifying safety and progress properties with RSL.* In Maurice Naftalin, Tim Denvir and Miquel Bertran, editeurs, FME '94: Industrial Benefit of Formal Methods, volume 873 of *Lecture Notes in Computer Science*, pages 567–581. Springer Berlin / Heidelberg, 1994. (Cited on page 43.)

[Groslambert 2006] Julien Groslambert. *Verification of LTL on B Event Systems*. In Jacques Julliand and Olga Kouchnarenko, editeurs, B 2007: Formal Specification and Development in B, volume 4355 of *Lecture Notes in Computer Science*, pages 109–124. Springer Berlin / Heidelberg, 2006. (Cited on page 45.)

[Group 1993a] The Raise Language Group. Raise method manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. (Cited on page 43.)

[Group 1993b] The RAISE Language Group. The raise specification language. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. (Cited on pages 14 and 43.)

[Hall 1990] Anthony Hall. *Seven Myths of Formal Methods*. IEEE Softw., vol. 7, no. 5, pages 11–19, 1990. (Cited on page 23.)

[Harel 1987] D. Harel. *Statecharts: A visual formalism for complex systems*. Science of Computer Programming, vol. 8, pages 231–274, 1987. (Cited on page 29.)

[Harrison 2003] John Harrison. *Formal Verification at Intel*. Logic in Computer Science, Symposium on, 2003. (Cited on page 24.)

[Harsu 2002] Maarit Harsu. *A survey on domain engineering*. Rapport technique, Tampere University of Technology, Tampere, Finland, 2002. (Cited on page 11.)

[Henzinger 2003] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar and Grégoire Sutre. *Software verification with BLAST*. In SPIN'03: Proceedings of the 10th international conference on Model checking software, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag. (Cited on page 30.)

[Hoare 1985] C. A. R. Hoare. Communicating sequential processes. Prentice Hall Int., 1985. (Cited on page 29.)

[Holbrook 1990] H. Holbrook. *A Scenario-based Methodology for Conducting Requirements Elicitation*. ACM Sigsoft Software Engineering Notes, vol. 1, pages 95–104, 1990. (Cited on page 21.)

[Holzmann 1997] Gerard J. Holzmann. *The Model Checker SPIN*. IEEE Trans. Software Engineering, vol. 23, no. 5, pages 279–295, 1997. (Cited on page 30.)

[Huth 2000] Michael R. A. Huth and Mark Ryan. Logic in computer science: modelling and reasoning about systems. Cambridge University Press, New York, NY, USA, 2000. (Cited on page 30.)

[IEEE 1990] Institute of Electrical IEEE and Electronics Engineers. *IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology*, 1990. (Cited on pages 14, 18 and 19.)

[IEEE 1998a] Institute of Electrical IEEE and Electronics Engineers. *IEEE standard for software reviews*, 1998. (Cited on pages 33 and 34.)

[IEEE 1998b] Institute of Electrical IEEE and Electronics Engineers. *IIEEE Recommended Practice for Software Requirements Specifications*, 1998. (Cited on page 18.)

[IEEE 2004] Institute of Electrical IEEE and Electronics Engineers. *IEEE Standard for Software Verification and Validation*, 2004. (Cited on page 19.)

[Jackson 1983] M. A Jackson. System development (prentice-hall international series in computer science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1983. (Cited on page 74.)

[Jackson 2001] M. Jackson. Problem frames: Analyzing and structuring software development problems. Addison-Wesley, 2001. (Cited on page 21.)

[Jarke 1998] Matthias Jarke, X. Tung Bui and John M. Carroll. *Scenario Management: An Interdisciplinary Approach*. Requirements Engineering, vol. 3, pages 155–173, 1998. (Cited on page 20.)

[Jones 1986] C B Jones. Systematic software development using vdm. Prentice Hall International (UK) Ltd., 1986. (Cited on page 18.)

[Joochim 2010] T. Joochim, C. Snook, M. Poppleton and A Gravell. *TIMING DIAGRAMS REQUIREMENTS MODELING USING EVENT-B FORMAL METHODS*. In IASTED International Conference on Software Engineering (SE2010), Innsbruck, Austria, 2010. (Cited on page 45.)

[Kang 1990] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Rapport technique, Carnegie-Mellon University Software Engineering Institute, 1990. (Cited on page 12.)

[Kaufmann 1996] Matt Kaufmann and J. Strother Moore. *ACL2: An Industrial Strength Version of Nqthm*. In Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96), 1996. (Cited on page 31.)

[Kotonya 1998] Gerald Kotonya and Ian Sommerville. Requirements engineering: Processes and techniques. Wiley, 1998. (Cited on page 33.)

[Lalioti 1993] V. Lalioti and P. Loucopoulos. *Visualisation for Validation*. In Fifth Conference an Advanced Information Systems Engineering, CAiSE, pages 143–164. Springer, 1993. (Cited on page 32.)

[Lamport 1977] L. Lamport. *Proving the Correctness of Multiprocess Programs*. IEEE Transactions on Software Engineering, vol. 3, no. 2, pages 125–143, 1977. (Cited on pages 43, 75 and 76.)

[Lamsweerde 2000] Axel van Lamsweerde. *Formal specification: a roadmap*. In ICSE '00: Proceedings of the Conference on The Future of Software Engineering, pages 147–159, New York, NY, USA, 2000. ACM. (Cited on page 28.)

[Lamsweerde 2001] A van Lamsweerde. *Goal-Oriented Requirements Engineering: A Guided Tour*. In 5th IEEE International Symposium on Requirements Engineering, 2001. (Cited on page 20.)

[Lanoix 2008] Arnaud Lanoix. *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. (Cited on pages 124 and 125.)

[Laplante 2009] Phillip A. Laplante. Requirements engineering for software and systems. Auerbach Publications, Boston, MA, USA, 2009. (Cited on page 15.)

[Leuschel 2003] M. Leuschel and M. Butler. *ProB: A Model Checker for B*. In Keijiro Araki, Stefania Gnesi and Dino Mandrioli, editeurs, FME 2003: Formal Methods, LNCS 2805, pages 855–874. Springer-Verlag, 2003. (Cited on pages 75 and 91.)

[Leuschel 2008] M. Leuschel and M. Butler. *ProB: An Automated Analysis Toolset for the B Method*. Journal Software Tools for Technology Transfer, vol. 10, no. 2, pages 185–203, 2008. (Cited on page 92.)

[Logrippo 1990] Luigi Logrippo, Tim Melanchuk and Robert J. Du Wors. *The algebraic specification language LOTOS: an industrial experience*. In Conference proceedings on Formal methods in software development, pages 59–66. ACM, 1990. (Cited on page 29.)

[Loucopoulos 1995] Pericles Loucopoulos and Vassilios Karakostas. System requirements engineering. McGraw-Hill, Inc., New York, NY, USA, 1995. (Cited on page 15.)

[Marc 1993] Edited Marc, John Hershberger (Eds.), Marc H. Brown and John Hershberger. *Animation of Geometric Algorithms: A Video Review*. DEC Systems Research Center, Research Report, vol. 87, 1993. (Cited on page 32.)

[Marca 1987] David A. Marca and Clement L. McGowan. Sadt: structured analysis and design technique. McGraw-Hill, Inc., New York, NY, USA, 1987. (Cited on page 20.)

[Mashkoor 2010a] Atif Mashkoor and Jean-Pierre Jacquot. *Domain Engineering with Event-B: Some Lessons We Learned*. In 18th International Requirements Engineering Conference (RE'10), Sydney, Australia, 2010. (Cited on page 45.)

[Mashkoor 2010b] Atif Mashkoor and Abderrahman Matoussi. *Towards Validation of Requirements Models*. In 2nd International Conference on Abstract State Machines (ASM), Alloy, B and Z (ABZ'10), Orford, Canada, 2010. (Cited on page 44.)

[Matoussi 2008] A. Matoussi, F. Gervais and R. Laleau. *A First Attempt to Express KAOS Refinement Patterns with Event B*. In Proc. of the Int. Conf. on ASM, B and Z (ABZ), Lecture Notes in Computer Science, pages 12–14. Springer-Verlag, 2008. (Cited on page 44.)

[McCall 1977] Jim A. McCall, Paul K. Richards and Gene F. Walters. *Factors in Software Quality (Volume-III)*. Technical report, GENERAL ELECTRIC CO SUNNYVALE CALIF, 1977. (Cited on page 18.)

[Milner 1982] R. Milner. A calculus of communicating systems. Springer-Verlag New York, Inc., 1982. (Cited on page 29.)

[Milner 1997] Robin Milner, Mads Tofte, Robert Harper and David Macqueen. The definition of standard ml - revised. The MIT Press, 1997. (Cited on page 44.)

[Mullery 1979] G. P. Mullery. *CORE - A method for controlled requirement specification*. In ICSE '79: Proceedings of the 4th international conference on Software engineering, pages 126–135. IEEE Press, 1979. (Cited on page 20.)

[Mylopoulos 1992] J. Mylopoulos, L. Chung and B. Nixon. *Representing and Using Nonfunctional Requirements: A Process-Oriented Approach*. IEEE Trans. Softw. Eng., vol. 18, no. 6, pages 483–497, 1992. (Cited on page 20.)

[Neighbors 1980] James Milne Neighbors. *Software construction using components*. PhD thesis, University of California, Irvine, USA, 1980. (Cited on page 11.)

[Neighbors 1989] J. M. Neighbors. *Draco: a method for engineering reusable software systems*. Software reusability: vol. 1, concepts and models, pages 295–319, 1989. (Cited on page 12.)

[Nipkow 2002] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. Isabelle/hol — a proof assistant for higher-order logic, volume 2283 of *LNCS*. Springer, 2002. (Cited on page 31.)

[Olsen 1994] Anders Olsen. Systems engineering using sdl-92. Elsevier Science Inc., 1994. (Cited on page 29.)

[Owre 1992] S. Owre, J. M. Rushby, and N. Shankar. *PVS: A Prototype Verification System*. In Deepak Kapur, editeur, 11th International Conference on Automated Deduction (CADE), volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. (Cited on page 44.)

[Papatsaras 2002] Antonis Papatsaras and Bill Stoddart. *Global and Communicating State Machine Models in Event Driven B: A Simple Railway Case Study.* In ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B, pages 458–476, London, UK, 2002. Springer-Verlag. (Cited on page 44.)

[Paulson 1996] Lawrence C. Paulson. Ml for the working programmer (2nd ed.). Cambridge University Press, New York, NY, USA, 1996. (Cited on page 43.)

[Paulson 2008] Lawrence C. Paulson. *The Isabelle Reference Manual*, 2008. (Cited on page 24.)

[Pnueli 1981] Amir Pnueli. *The Temporal Semantics of Concurrent Programs.* Theor. Comput. Sci., vol. 13, pages 45–60, 1981. (Cited on pages 29 and 30.)

[Ponsard 2007] C. Ponsard, P. Massonet, J.F. Molderez and G. Dallons. *Formal Requirements Modelling and Early Verification and Validation of Critical Systems.* In Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'07, 2007. (Cited on page 32.)

[Poppleton 2007] Michael R. Poppleton. *Towards feature-oriented specification and development with event-B.* In Proceedings of the 13th international working conference on Requirements engineering: foundation for software quality, REFSQ'07, pages 367–381, 2007. (Cited on page 45.)

[Poppleton 2008] Michael Poppleton. *The composition of Event-B models.* In 1st International Conference on ASM, B and Z (ABZ2008), London, UK, 2008. (Cited on page 79.)

[Pressman 2005] R. S. Pressman. Software engineering: A practitioner's approach. McGraw-Hill Inc, 2005. (Cited on pages 11 and 17.)

[rai 2002] Specification case studies in raise. Springer-Verlag, London, UK, 2002. (Cited on page 14.)

[Raja 2009] Uzair Akbar Raja. *Empirical Studies of Requirements Validation Techniques.* In 2nd International Conference on Computer, Control and Communication, Karachi, Pakistan, 2009. (Cited on page 31.)

[Roman 1989] G.-C. Roman and K. C. Cox. *A Declarative Approach to Visualizing Concurrent Computations.* Computer, vol. 22, no. 10, pages 25–36, 1989. (Cited on page 33.)

[Ross 1977] D. T. Ross and K. E. Schoman. *Structured Analysis for Requirements Definition.* IEEE Transactions on Software Engineering, vol. 3, no. 1, pages 6–15, 1977. (Cited on page 17.)

[Roy 1992] Valérie Roy and Robert De Simone. *Auto/Autograph.* Formal Methods in System Design, vol. 1, no. 2, pages 239–249, 1992. (Cited on page 30.)

[Scheuer 2008] Alexis Scheuer, Olivier Simonin and François Charpillet. *Safe Longitudinal Platoons of Vehicles without Communication*. Research Report RR-6741, INRIA, 2008. (Cited on page 123.)

[Schneider 2010] Steve Schneider, Helen Treharne and Heike Wehrheim. *A CSP Approach to Control in Event-B*. In Integrated Formal Methods - IFM 2010 Integrated Formal Methods - 8th International Conference, IFM 2010, volume 6396 of *LNCS*, pages 260–274, Nancy France, 2010. Springer Berlin / Heidelberg. (Cited on page 143.)

[Servat 2006] Thierry Servat. *BRAMA: A New Graphic Animation Tool for B Models*. In B 2007: Formal Specification and Development in B, pages 274–276. Springer-Verlag, 2006. (Cited on pages 5 and 87.)

[Silva 2010] Renato Silva, Carine Pascal, T.S. Hoang and Michael Butler. *Decomposition Tool for Event-B*. In Workshop on Tool Building in Formal Methods, Orford, Canada, 2010. (Cited on pages 45 and 79.)

[Simos 1995] Mark A. Simos. *Organization domain modeling (ODM): formalizing the core domain modeling life cycle*. SIGSOFT Software Engineering Notes, vol. 20, no. SI, pages 196–205, 1995. (Cited on page 13.)

[Simos 1996] Mark Simos, Dick Creps, Carol Klingler, Larry Levine and Dean Allemang. *Organization domain modeling (ODM) guidebook, version 2.0*. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defence Systems, June 1996. (Cited on page 13.)

[Simos 1997] M. Simos. *Organization Domain Modeling and OO Analysis and Design: Distinctions, Integration, New Directions*. In In STJA'97 Conference Proceedings, pages 166–175, 1997. (Cited on page 10.)

[Snook 2006] C. Snook and M. Butler. *UML-B: Formal modeling and design aided by UML*. ACM Transactions on Software Engineering and Methodology, vol. 15, no. 1, pages 92–122, 2006. (Cited on page 44.)

[Sommerville 1997] I. Sommerville and P. Sawyer. *Viewpoints: principles, problems and a practical approach to requirements engineering*. Annals of Software Engineering, vol. 3, pages 101–130, 1997. (Cited on page 20.)

[Sommerville 2006] Ian Sommerville. Software engineering (8th ed.). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2006. (Cited on pages 15 and 16.)

[Spivey 1989] J. M. Spivey. The z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. (Cited on page 18.)

[Srinivas 1991] Yellamraju Srinivas. *Algebraic specification for domains*. Domain Analysis: Acquisition of Reusable Information for Software Construction, pages 90–124, 1991. (Cited on page 13.)

[Sutcliffe 2003] Alistair Sutcliffe. *Scenario-Based Requirements Engineering*. Requirements Engineering, IEEE International Conference on, 2003. (Cited on page 20.)

[Thayer 1997] Richard H. Thayer, Sidney C. Bailin and M. Dorfman. Software requirements engineerings, 2nd edition. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997. (Cited on page 15.)

[van Lamsweerde 2009] Axel van Lamsweerde. Requirements engineering: From system goals to uml models to software specifications. Wiley, 2009. (Cited on page 44.)

[Van 2004] Hung Tran Van, Axel van Lamsweerde, Philippe Massonet and Christophe Ponsard. *Goal-Oriented Requirements Animation*. In RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International, pages 218–228, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 32.)

[Wartik 1992] Steven Wartik and Rub'en Prieto-Diaz. *CRITERIA FOR COMPARING REUSE-ORIENTED DOMAIN ANALYSIS APPROACHES*. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), vol. 2, pages 403–431, 1992. (Cited on page 11.)

[Yadav 2009] Divakar Yadav and Michael Butler. *Verification of Liveness Properties in Distributed Systems*. In Second International Conference on Contemporary Computing (IC3'09), pages 625–636. Noida, India, 2009. (Cited on page 76.)

[Yourdon 1978] Edward Yourdon. Structured walkthroughs. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978. (Cited on page 33.)

[Yourdon 1979] Edward Yourdon and Larry L. Constantine. Structured design: Fundamentals of a discipline of computer program and systems design. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st édition, 1979. (Cited on page 18.)

[Yousuf 2008] Farzana Yousuf, Zahid Zaman and Naveed Ikram. *Requirements Validation Techniques in GSD: A Survey*. In I2th IEEE International Multitopic Conference (INMIC'08), Karachi, Pakistan, 2008. (Cited on page 31.)

[Yu 1997] Eric S. K. Yu. *Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering*. In RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, Washington, DC, USA, 1997. IEEE Computer Society. (Cited on page 20.)

[Zave 1997a] P. Zave and M. Jackson. *Four dark corners for requirements engineering*. ACM Transactions on Software Engineering and Methodology, vol. 6, no. 1, pages 1–30, January 1997. (Cited on page 72.)

[Zave 1997b] Pamela Zave. *Classification of research efforts in requirements engineering.* ACM Computing Surveys, vol. 29, no. 4, pages 315–321, 1997. (Cited on page 15.)

[Zowghi 2005] Didar Zowghi and Chad Coulin. *Requirements Elicitation: A Survey of Techniques, Approaches, and Tools.* Engineering and Managing Software Requirements, pages 19–46, 2005. (Cited on page 17.)

Appendix A

Original Event-B specifications

A.1 The land transport domain model

An Event-B Specification of Net
Creation Date: 11 Jul 2011 @ 07:14:51 PM

CONTEXT Net

SETS

`Nets`
`Hubs`
`Connections`

CONSTANTS

`obsNetHubs`
`obsNetConnections`
`hubConnections`
`connectionOrigin`
`connectionDestination`

AXIOMS

axm9 : $Connections \neq \varnothing$
axm8 : $Nets \neq \varnothing$
axm6 : $Hubs \neq \varnothing$
axm3 : $finite(Connections)$
axm2 : $finite(Nets)$
axm1 : $finite(Hubs)$
typ1 : $obsNetHubs \in Hubs \leftrightarrow Nets$
axm4 : $finite(obsNetHubs)$
typ2 : $obsNetConnections \in Connections \rightarrow Nets$
axm5 : $finite(obsNetConnections)$
typ3 : $connectionOrigin \in Connections \rightarrow Hubs$
typ4 : $connectionDestination \in Connections \rightarrow Hubs$
typ5 : $hubConnections \in Hubs \rightarrow \mathbb{P}(Connections)$
prop1 : $dom(obsNetHubs) = Hubs \land ran(obsNetHubs) = Nets$
prop2 : $dom(obsNetConnections) = Connections \land ran(obsNetConnections) = Nets$
prop3 : $\forall c \cdot c \in Connections \Rightarrow connectionOrigin(c) \neq connectionDestination(c)$
prop4 : $\forall n \cdot n \in Nets \Rightarrow card(obsNetHubs^{-1}[\{n\}]) \geq 2 \land card(obsNetConnections^{-1}[\{n\}]) \geq 1$
prop5 : $\forall c \cdot c \in Connections \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionOrigin(c)\}] \land obsNetConnections[\{c\}] \subseteq obsNetHubs[\{connectionDestination(c)\}]$
prop6 : $\forall h, c \cdot h \in Hubs \land c \in hubConnections(h) \Rightarrow obsNetConnections[\{c\}] \subseteq obsNetHubs[\{h\}]$
prop7 : $\forall c \cdot c \in Connections \Rightarrow card(obsNetConnections[\{c\}]) = 1$
prop8 : $\forall h \cdot h \in Hubs \Rightarrow card(obsNetHubs[\{h\}]) \geq 1$

END

An Event-B Specification of Net1
Creation Date: 11 Jul 2011 @ 07:14:54 PM

CONTEXT Net1
EXTENDS Net
CONSTANTS

`junctions`
`stations`
`obsNetJunctions`
`isJunction`
`obsNetStations`

AXIOMS

typ1 : $junctions \subseteq Hubs$
typ2 : $stations \subseteq Hubs$
typ3 : $obsNetJunctions \in Nets \rightarrow \mathbb{P}(junctions)$
typ4 : $isJunction \in Hubs \rightarrow BOOL$
type5 : $obsNetStations \in Nets \rightarrow \mathbb{P}(stations)$
prop1 : $junctions \cap stations = \varnothing$
prop2 : $junctions \cup stations = Hubs$
prop3 : $\forall h \cdot h \in Hubs \wedge card(hubConnections(h)) = 1 \Rightarrow isJunction(h) = FALSE$
prop4 : $\forall h \cdot h \in stations \Rightarrow isJunction(h) = FALSE$
prop5 : $\forall h \cdot h \in junctions \Rightarrow isJunction(h) = TRUE$
prop6 : $\forall n \cdot n \in Nets \Rightarrow card(obsNetHubs^{-1}[\{n\}] \cap stations) \geq 2$

END

An Event-B Specification of Net2
Creation Date: 11 Jul 2011 @ 07:14:56 PM

CONTEXT Net2
EXTENDS Net1
CONSTANTS

`paths`
`routes`
`isRoute`
`seqPaths`

AXIOMS

`type1` : $paths \subseteq Connections$

`type2` : $seqPaths = \{seq | \exists n \cdot n \in \mathbb{N}_1 \land seq \in 1 \mathinner{..} n \rightarrowtail paths \land finite(seq) \land card(seq) = n\}$

`type3` : $isRoute \in seqPaths \rightarrow BOOL$

`def1` : $\forall r \cdot r \in seqPaths \land ((connectionOrigin(r(1)) \in stations \land connectionDestination(r(card(r))) \in stations \land (obsNetHubs[\{connectionOrigin(r(1))\}] \cap obsNetHubs[\{connectionDestination(r(card(r)))\}] = \varnothing) \land (\forall i \cdot i \in 2..card(r) \land connectionDestination(r(i-1)) = connectionOrigin(r(i))) \land connectionOrigin(r(1)) \neq connectionDestination(r(card(r))) \land (\forall i1, i2 \cdot i1 \in 1..card(r) \land i2 \in 1..card(r) \land i1 \neq i2 \Rightarrow connectionOrigin(r(i1)) \neq connectionOrigin(r(i2))) \land (\forall i1, i2 \cdot i1 \in 1..card(r) \land i2 \in 1..card(r) \land i1 \neq i2 \Rightarrow connectionDestination(r(i1)) \neq connectionDestination(r(i2)))) \Leftrightarrow isRoute(r) = TRUE)$

`prop1` : $routes = \{sp | sp \in seqPaths \land isRoute(sp) = TRUE\}$

`axm1` : $\forall c \cdot c \in Connections \Rightarrow (connectionDestination(c) \in stations \land connectionOrigin(c) \in stations \Rightarrow (\exists r \cdot r \in routes \land connectionOrigin(c) = connectionOrigin(r(1)) \land connectionDestination(c) = connectionDestination(r(card(r)))))$

END

An Event-B Specification of Net4
Creation Date: 11 Jul 2011 @ 07:14:58 PM

CONTEXT Net4
EXTENDS Net2
SETS
`States`
CONSTANTS
`hubCapacity`
`entering`
`leaving`
`onHub`
`crossed`
`initial`
AXIOMS
`type1` : $hubCapacity \in Hubs \rightarrow \mathbb{N}_1$
`type2` : $States = \{entering, leaving, onHub, crossed, initial\}$
`prop1` : $\forall j \cdot j \in junctions \Rightarrow hubCapacity(j) = 1$
`prop2` : $\forall s \cdot s \in stations \Rightarrow (\exists n \cdot n \in 1 \mathinner{..} 100 \land hubCapacity(s) = n)$
`prop3` : $entering \neq leaving$
`prop4` : $entering \neq onHub$
`prop5` : $entering \neq crossed$
`prop6` : $leaving \neq onHub$
`prop7` : $leaving \neq crossed$
`prop8` : $onHub \neq crossed$
`prop9` : $initial \neq entering$
`prop10` : $initial \neq onHub$
`prop11` : $initial \neq leaving$
`prop12` : $initial \neq crossed$
END

An Event-B Specification of Net5
Creation Date: 11 Jul 2011 @ 07:14:59 PM

CONTEXT Net5
EXTENDS Net4
CONSTANTS

`hubCrossingTime`

`pathTraversingTime` cette variable n'est plus ncessaire

`deltaTime` La constante de temps pour les calculs de distance avec la vitesse

AXIOMS

`type1` : $hubCrossingTime \in Hubs \rightarrow \mathbb{N}$

`type2` : $pathTraversingTime \in paths \rightarrow \mathbb{N}$

`type3` : $deltaTime \in \mathbb{N}$

`prop3` : $deltaTime > 0$

END

An Event-B Specification of Net6
Creation Date: 11 Jul 2011 @ 07:15:02 PM

CONTEXT Net6
EXTENDS Net5
CONSTANTS

`pathLen`

AXIOMS

`axm3 :` $pathLen \in paths \rightarrow \mathbb{N}_1$

END

An Event-B Specification of Location
Creation Date: 11 Jul 2011 @ 07:14:49 PM

CONTEXT Location
EXTENDS Net
SETS
GlobalLocations
CONSTANTS
hubLocations
connectionLocations
obsHubLocations
obsConnectionLocations
AXIOMS
type1 : $hubLocations \subseteq GlobalLocations$
type2 : $connectionLocations \subseteq GlobalLocations$
type3 : $obsHubLocations \in Hubs \rightarrow \mathbb{P}(hubLocations)$
type4 : $obsConnectionLocations \in Connections \rightarrow \mathbb{P}(connectionLocations)$
prop1 : $hubLocations \neq connectionLocations$
prop2 : $hubLocations \cup connectionLocations = GlobalLocations$
axm1 : $GlobalLocations \neq \varnothing$
axm2 : $\forall h1, h2 \cdot h1 \in Hubs \wedge h2 \in Hubs \wedge h1 \neq h2 \Rightarrow obsHubLocations(h1) \cap obsHubLocations(h2) = \varnothing$
END

An Event-B Specification of StartState
Creation Date: 11 Jul 2011 @ 07:15:07 PM

CONTEXT StartState
EXTENDS Location
CONSTANTS
`startVehicleLocation`
AXIOMS
`prop1` : $startVehicleLocation \in Vehicles \rightarrow GlobalLocations$
END

An Event-B Specification of StartState1
Creation Date: 11 Jul 2011 @ 07:15:09 PM

CONTEXT StartState1
EXTENDS StartState
AXIOMS

prop1 : $\forall v \cdot v \in Vehicles \Rightarrow (\exists s \cdot s \in stations \land startVehicleLocation(v) \in obsHubLocations(s))$

END

An Event-B Specification of StartState3
Creation Date: 11 Jul 2011 @ 07:15:11 PM

CONTEXT StartState3
EXTENDS StartState1
CONSTANTS

`startHubsToCross`

`startConnectionsToTraverse`

AXIOMS

`type1` : $startConnectionsToTraverse \subseteq Vehicles \times paths$

`type2` : $startHubsToCross \subseteq Vehicles \times Hubs$

`axm1` : $startConnectionsToTraverse = \varnothing$

`axm2` : $startHubsToCross = \varnothing$

END

An Event-B Specification of StartState4
Creation Date: 11 Jul 2011 @ 07:15:14 PM

CONTEXT StartState4
EXTENDS StartState3
AXIOMS

prop1 : $\forall h \cdot h \in Hubs \Rightarrow hubCapacity(h) \geq card(\{v \cdot v \in Vehicles \land startVehicleLocation(v) \in obsHubLocations(h) | v\})$

END

An Event-B Specification of StartState5
Creation Date: 11 Jul 2011 @ 07:15:15 PM

CONTEXT StartState5
EXTENDS StartState4
CONSTANTS

`startVehicleTime`

`startTravelTime`

AXIOMS

`type1` : $startVehicleTime \in Vehicles \rightarrow \mathbb{N}$

`type2` : $startTravelTime \in Vehicles \rightarrow \mathbb{N}$

END

An Event-B Specification of Vehicle
Creation Date: 11 Jul 2011 @ 07:15:17 PM

CONTEXT Vehicle

SETS

`Vehicles`

CONSTANTS

`criticalDistance`

`maxSpeed`

`minSpeed`

AXIOMS

`axm1` : $Vehicles \neq \varnothing$

`axm2` : $criticalDistance \in \mathbb{N}_1$

`axm3` : $criticalDistance > 1$

`axm4` : $maxSpeed \in \mathbb{N}_1$

`axm5` : $minSpeed \in \mathbb{N}_1$

`axm6` : $finite(Vehicles)$

END

An Event-B Specification of Movement0
Creation Date: 11 Jul 2011 @ 07:15:19 PM

MACHINE Movement0
SEES StartState
VARIABLES
`location`
INVARIANTS
`inv1 :` $location \in Vehicles \rightarrow GlobalLocations$
EVENTS
Initialisation
begin
`act1 :` $location := startVehicleLocation$
end
Event $travel \mathrel{\widehat{=}}$
any
$vehicle$
$newLocation$
where
`grd1 :` $vehicle \in Vehicles$
`grd2 :` $newLocation \in GlobalLocations$
`grd3 :` $newLocation \neq location(vehicle)$
then
`act1 :` $location(vehicle) := newLocation$
end
END

An Event-B Specification of Movement1
Creation Date: 11 Jul 2011 @ 07:15:21 PM

MACHINE Movement1
REFINES Movement0
SEES StartState1
VARIABLES
`location`
EVENTS
Initialisation
begin
`act1 :` $location := startVehicleLocation$
end
Event $travel \mathrel{\widehat{=}}$
refines $travel$
any
$vehicle$
$newLocation$
$origin$
$destination$
where
`grd1 :` $vehicle \in Vehicles$
`grd2 :` $origin \in stations$
`grd3 :` $destination \in stations$
`grd4 :` $origin \neq destination$
`grd5 :` $location(vehicle) \in obsHubLocations(origin)$
`grd6 :` $newLocation \in obsHubLocations(destination)$
`grd7 :` $newLocation \neq location(vehicle)$
then
`act1 :` $location(vehicle) := newLocation$
end
END

An Event-B Specification of Movement2
Creation Date: 11 Jul 2011 @ 07:15:23 PM

MACHINE Movement2
REFINES Movement1
SEES Net2, StartState1
VARIABLES
`location`
EVENTS
Initialisation
begin
act1 : $location := startVehicleLocation$
end
Event *travel* $\widehat{=}$
refines *travel*
any
$vehicle$
$newLocation$
r
$origin$
$destination$
where
grd1 : $r \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $origin \in stations$
grd4 : $destination \in stations$
grd5 : $origin \neq destination$
grd6 : $origin = connectionOrigin(r(1))$
grd7 : $destination = connectionDestination(r(card(r)))$
grd8 : $location(vehicle) \in obsHubLocations(origin)$
grd9 : $newLocation \in obsHubLocations(destination)$
grd10 : $newLocation \neq location(vehicle)$
then
act1 : $location(vehicle) := newLocation$
end
END

An Event-B Specification of Movement3
Creation Date: 11 Jul 2011 @ 07:15:25 PM

MACHINE Movement3
REFINES Movement2
SEES StartState3
VARIABLES
`position`
`connectionsToTraverse`
`hubsToCross`
`location`
INVARIANTS
`inv1` : $connectionsToTraverse \subseteq Vehicles \times paths$
`inv2` : $hubsToCross \subseteq Vehicles \times Hubs$
`inv3` : $position \in Vehicles \rightarrow GlobalLocations$
EVENTS
Initialisation
begin
`act1` : $position := startVehicleLocation$
`act2` : $connectionsToTraverse := startConnectionsToTraverse$
`act3` : $hubsToCross := startHubsToCross$
`act4` : $location := startVehicleLocation$
end
Event *travel* $\widehat{=}$
refines *travel*
any
vehicle
newLocation
r
origin
destination
where
`grd1` : $r \in routes$
`grd2` : $vehicle \in Vehicles$
`grd3` : $origin \in stations$
`grd4` : $destination \in stations$
`grd5` : $origin \neq destination$
`grd6` : $origin = connectionOrigin(r(1))$
`grd7` : $destination = connectionDestination(r(card(r)))$
`grd8` : $location(vehicle) \in obsHubLocations(origin)$
`grd9` : $newLocation \in obsHubLocations(destination)$
`grd10` : $newLocation \neq location(vehicle)$
`grd11` : $position(vehicle) = newLocation$
then
`act1` : $location(vehicle) := newLocation$
end
Event *traversePath* $\widehat{=}$
Status convergent
any
vehicle
r
p
newPosition
where

grd1 : $vehicle \in Vehicles$
grd2 : $r \in routes$
grd3 : $p \in ran(r)$
grd4 : $position(vehicle) \in obsHubLocations(connectionOrigin(p))$
grd5 : $newPosition \in obsHubLocations(connectionDestination(p))$
grd6 : $newPosition \neq position(vehicle)$
grd7 : $vehicle \mapsto p \in connectionsToTraverse$
grd8 : $vehicle \mapsto connectionOrigin(p) \notin hubsToCross$

then

act1 : $position(vehicle) := newPosition$
act2 : $connectionsToTraverse := connectionsToTraverse \setminus \{vehicle \mapsto p\}$

end

Event *crossHub* $\widehat{=}$

Status convergent

any

vehicle
hub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $vehicle \mapsto hub \in hubsToCross$

then

act2 : $hubsToCross := hubsToCross \setminus \{vehicle \mapsto hub\}$

end

Event *startTravel* $\widehat{=}$

any

vehicle
r

where

grd1 : $vehicle \in Vehicles$
grd2 : $r \in routes$
grd3 : $position(vehicle) \in obsHubLocations(connectionOrigin(r(1)))$
grd5 : $vehicle \notin dom(connectionsToTraverse)$
grd4 : $vehicle \notin dom(hubsToCross)$

then

act1 : $hubsToCross := hubsToCross \cup \{i \cdot i \in 1..card(r) | vehicle \mapsto connectionOrigin(r(i))\}$
act2 : $connectionsToTraverse := connectionsToTraverse \cup \{p \cdot p \in ran(r) | vehicle \mapsto p\}$

end

VARIANT

$\texttt{card}(\texttt{hubsToCross}) + \texttt{card}(\texttt{connectionsToTraverse})$

END

An Event-B Specification of Movement4
Creation Date: 11 Jul 2011 @ 07:15:27 PM

MACHINE Movement4
REFINES Movement3
SEES StartState4
VARIABLES

`position`
`connectionsToTraverse`
`hubsToCross`
`location`
`hubLoad`
`vehicleState`

INVARIANTS

inv1 : $hubLoad \in Hubs \to \mathbb{N}$
inv2 : $\forall h \cdot h \in Hubs \Rightarrow hubLoad(h) \leq hubCapacity(h)$
inv3 : $vehicleState \in Vehicles \times Hubs \to States$
inv4 : $\forall v \cdot card(\{h \cdot vehicleState(v \mapsto h) = onHub \lor vehicleState(v \mapsto h) = leaving | h\}) \leq 1$
inv5 : $\forall v, h \cdot vehicleState(v \mapsto h) = onHub \Rightarrow position(v) \in obsHubLocations(h)$
inv6 : $\forall v, h \cdot vehicleState(v \mapsto h) = leaving \Rightarrow position(v) \in obsHubLocations(h)$

EVENTS
Initialisation
begin

act1 : $position := startVehicleLocation$
act2 : $connectionsToTraverse := startConnectionsToTraverse$
act3 : $hubsToCross := startHubsToCross$
act4 : $location := startVehicleLocation$
act5 : $hubLoad := \{h \cdot h \in Hubs | h \mapsto 0\}$
act6 : $vehicleState := \{v, h \cdot v \in Vehicles \land h \in Hubs \land startVehicleLocation(v) \notin obsHubLocations(h) | (v \mapsto h) \mapsto initial\} \cup \{v, h \cdot v \in Vehicles \land h \in Hubs \land startVehicleLocation(v) \in obsHubLocations(h) | (v \mapsto h) \mapsto onHub\}$

end

Event $travel \mathrel{\widehat{=}}$
refines *travel*
any

vehicle
newLocation
r
origin
destination

where

grd1 : $r \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $origin \in stations$
grd4 : $destination \in stations$
grd5 : $origin \neq destination$
grd6 : $origin = connectionOrigin(r(1))$
grd7 : $destination = connectionDestination(r(card(r)))$
grd8 : $location(vehicle) \in obsHubLocations(origin)$
grd9 : $newLocation \in obsHubLocations(destination)$
grd10 : $newLocation \neq location(vehicle)$
grd11 : $position(vehicle) = newLocation$

grd12 : $vehicleState(vehicle \mapsto destination) = onHub$

then

act1 : $location(vehicle) := newLocation$

end

Event *traversePath* $\widehat{=}$

Status convergent

refines *traversePath*

any

vehicle

r

p

newPosition

where

grd1 : $vehicle \in Vehicles$

grd2 : $r \in routes$

grd3 : $p \in paths \land p \in ran(r)$

grd4 : $position(vehicle) \in obsHubLocations(connectionOrigin(p))$

grd5 : $newPosition \in obsHubLocations(connectionDestination(p))$

grd6 : $newPosition \neq position(vehicle)$

grd7 : $vehicle \mapsto p \in connectionsToTraverse$

grd8 : $vehicle \mapsto connectionOrigin(p) \notin hubsToCross$

grd9 : $vehicleState(vehicle \mapsto connectionOrigin(p)) = crossed$

then

act1 : $position(vehicle) := newPosition$

act2 : $connectionsToTraverse := connectionsToTraverse \setminus \{vehicle \mapsto p\}$

act3 : $vehicleState(vehicle \mapsto connectionDestination(p)) := entering$

end

Event *crossHub* $\widehat{=}$

Status convergent

refines *crossHub*

any

vehicle

hub

where

grd1 : $vehicle \in Vehicles$

grd2 : $hub \in Hubs$

grd3 : $position(vehicle) \in obsHubLocations(hub)$

grd4 : $vehicle \mapsto hub \in hubsToCross$

grd5 : $vehicleState(vehicle \mapsto hub) = onHub$

then

act1 : $hubsToCross := hubsToCross \setminus \{vehicle \mapsto hub\}$

act2 : $vehicleState(vehicle \mapsto hub) := leaving$

end

Event *startTravel* $\widehat{=}$

refines *startTravel*

any

vehicle

r

where

grd1 : $vehicle \in Vehicles$

grd2 : $r \in routes$

grd3 : $position(vehicle) \in obsHubLocations(connectionOrigin(r(1)))$

grd4 : $vehicleState(vehicle \mapsto connectionOrigin(r(1))) = onHub$

grd5 : $vehicle \notin dom(connectionsToTraverse)$

grd6 : $vehicle \notin dom(hubsToCross)$

then

act1 : $connectionsToTraverse := connectionsToTraverse \cup \{p \cdot p \in ran(r) | vehicle \mapsto p\}$
act2 : $hubsToCross := hubsToCross \cup \{i \cdot i \in 1..card(r) | vehicle \mapsto connectionOrigin(r(i))\}$
act3 : $vehicleState := (vehicleState \lhd \{i \cdot i \in 2..card(r) | vehicle \mapsto connectionOrigin(r(i)) \mapsto initial\}) \lhd \{vehicle \mapsto connectionDestination(r(card(r))) \mapsto initial\}$

end

Event *enterHub* $\widehat{=}$

any
vehicle
hub

where
grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $hubLoad(hub) < hubCapacity(hub)$
grd5 : $vehicleState(vehicle \mapsto hub) = entering$

then
act1 : $position(vehicle) := position(vehicle)$
act2 : $hubLoad(hub) := hubLoad(hub) + 1$
act3 : $vehicleState(vehicle \mapsto hub) := onHub$

end

Event *leaveHub* $\widehat{=}$

any
vehicle
hub
r

where
grd1 : $r \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $hub \in Hubs$
grd4 : $position(vehicle) \in obsHubLocations(hub)$
grd5 : $vehicle \mapsto hub \notin hubsToCross$
grd6 : $vehicleState(vehicle \mapsto hub) = leaving$
grd7 : $hubLoad(hub) \geq 1$

then
act2 : $hubLoad(hub) := hubLoad(hub) - 1$
act3 : $vehicleState(vehicle \mapsto hub) := crossed$

end

Event *wait* $\widehat{=}$

any
vehicle
hub

where
grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $hubLoad(hub) \geq hubCapacity(hub)$
grd5 : $vehicleState(vehicle \mapsto hub) = entering$

then
act1 : $position(vehicle) := position(vehicle)$
act2 : $vehicleState(vehicle \mapsto hub) := entering$

end

VARIANT
$\texttt{card}(\texttt{hubsToCross}) + \texttt{card}(\texttt{connectionsToTraverse})$

END

An Event-B Specification of Movement5
Creation Date: 11 Jul 2011 @ 07:15:29 PM

MACHINE Movement5
REFINES Movement4
SEES StartState5
VARIABLES

`position`
`connectionsToTraverse`
`hubsToCross`
`location`
`hubLoad`
`vehicleState`
`travelTime`
`time`
`startTime`

INVARIANTS

inv1 : $travelTime \in Vehicles \rightarrow \mathbb{N}$
inv2 : $time \in \mathbb{N}$
inv3 : $startTime \in Vehicles \rightarrow \mathbb{N}$

EVENTS

Initialisation

begin

act1 : $position := startVehicleLocation$
act2 : $connectionsToTraverse := startConnectionsToTraverse$
act3 : $hubsToCross := startHubsToCross$
act4 : $location := startVehicleLocation$
act5 : $hubLoad := \{h \cdot h \in Hubs | h \mapsto 0\}$
act6 : $vehicleState := \{v, h \cdot v \in Vehicles \wedge h \in Hubs \wedge startVehicleLocation(v) \notin obsHubLocations(h) | (v \mapsto h) \mapsto initial\} \cup \{v, h \cdot v \in Vehicles \wedge h \in Hubs \wedge startVehicleLocation(v) \in obsHubLocations(h) | (v \mapsto h) \mapsto onHub\}$
act7 : $startTime := startVehicleTime$
act8 : $travelTime := startTravelTime$
act9 : $time := 0$

end

Event $travel \mathrel{\widehat{=}}$
refines *travel*

any

vehicle
newLocation
r
origin
destination

where

grd1 : $r \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $origin \in stations$
grd4 : $destination \in stations$
grd5 : $origin \neq destination$
grd6 : $origin = connectionOrigin(r(1))$
grd7 : $destination = connectionDestination(r(card(r)))$
grd8 : $location(vehicle) \in obsHubLocations(origin)$

grd9 : $newLocation \in obsHubLocations(destination)$
grd10 : $newLocation \neq location(vehicle)$
grd11 : $position(vehicle) = newLocation$
grd12 : $vehicleState(vehicle \mapsto destination) = onHub$
grd13 : $time \geq startTime(vehicle)$

then

act1 : $location(vehicle) := newLocation$
act2 : $travelTime(vehicle) := time - startTime(vehicle)$

end

Event *traversePath* $\widehat{=}$
Status convergent
refines *traversePath*

any

vehicle
r
p
newPosition

where

grd1 : $vehicle \in Vehicles$
grd2 : $r \in routes$
grd3 : $p \in paths \land p \in ran(r)$
grd4 : $position(vehicle) \in obsHubLocations(connectionOrigin(p))$
grd5 : $newPosition \in obsHubLocations(connectionDestination(p))$
grd6 : $newPosition \neq position(vehicle)$
grd7 : $vehicle \mapsto p \in connectionsToTraverse$
grd8 : $vehicle \mapsto connectionOrigin(p) \notin hubsToCross$
grd9 : $vehicleState(vehicle \mapsto connectionOrigin(p)) = crossed$

then

act1 : $position(vehicle) := newPosition$
act2 : $connectionsToTraverse := connectionsToTraverse \setminus \{vehicle \mapsto p\}$
act3 : $vehicleState(vehicle \mapsto connectionDestination(p)) := entering$

end

Event *crossHub* $\widehat{=}$
Status convergent
refines *crossHub*

any

vehicle
hub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $vehicle \mapsto hub \in hubsToCross$
grd5 : $vehicleState(vehicle \mapsto hub) = onHub$

then

act1 : $hubsToCross := hubsToCross \setminus \{vehicle \mapsto hub\}$
act2 : $vehicleState(vehicle \mapsto hub) := leaving$

end

Event *startTravel* $\widehat{=}$
refines *startTravel*

any

vehicle
r

where

grd1 : $vehicle \in Vehicles$
grd2 : $r \in routes$

grd3 : $position(vehicle) \in obsHubLocations(connectionOrigin(r(1)))$
grd4 : $vehicleState(vehicle \mapsto connectionOrigin(r(1))) = onHub$
grd5 : $vehicle \notin dom(connectionsToTraverse)$
grd6 : $vehicle \notin dom(hubsToCross)$

then

act1 : $connectionsToTraverse := connectionsToTraverse \cup \{p \cdot p \in ran(r) | vehicle \mapsto p\}$
act2 : $hubsToCross := hubsToCross \cup \{i \cdot i \in 1..card(r) | vehicle \mapsto connectionOrigin(r(i))\}$
act3 : $vehicleState := (vehicleState \lhd\!\!\!\!- \{i \cdot i \in 2..card(r) | vehicle \mapsto connectionOrigin(r(i)) \mapsto initial\}) \lhd\!\!\!\!-$
$\{vehicle \mapsto connectionDestination(r(card(r))) \mapsto initial\}$
act4 : $startTime(vehicle) := time$

end

Event *enterHub* $\widehat{=}$
Status convergent
refines *enterHub*

any

vehicle
hub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $hubLoad(hub) < hubCapacity(hub)$
grd5 : $vehicleState(vehicle \mapsto hub) = entering$

then

act1 : $position(vehicle) := position(vehicle)$
act2 : $hubLoad(hub) := hubLoad(hub) + 1$
act3 : $vehicleState(vehicle \mapsto hub) := onHub$

end

Event *leaveHub* $\widehat{=}$
Status convergent
refines *leaveHub*

any

vehicle
hub
r

where

grd1 : $r \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $hub \in Hubs$
grd4 : $position(vehicle) \in obsHubLocations(hub)$
grd5 : $vehicle \mapsto hub \notin hubsToCross$
grd6 : $vehicleState(vehicle \mapsto hub) = leaving$
grd7 : $hubLoad(hub) \geq 1$

then

act2 : $hubLoad(hub) := hubLoad(hub) - 1$
act3 : $vehicleState(vehicle \mapsto hub) := crossed$

end

Event *wait* $\widehat{=}$
refines *wait*

any

vehicle
hub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$

grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $hubLoad(hub) \geq hubCapacity(hub)$
grd5 : $vehicleState(vehicle \mapsto hub) = entering$

then

act1 : $position(vehicle) := position(vehicle)$
act2 : $vehicleState(vehicle \mapsto hub) := entering$

end

Event $tickTac \mathrel{\widehat{=}}$

any

t

where

grd1 : $t \in \mathbb{N}$
grd2 : $t > time$

then

act1 : $time := t$

end

END

An Event-B Specification of Movement6
Creation Date: 11 Jul 2011 @ 07:15:30 PM

MACHINE Movement6
REFINES Movement5
SEES Net6, StartState5
VARIABLES

`position`
`connectionsToTraverse`
`hubsToCross`
`location`
`hubLoad`
`vehicleState`
`travelTime`
`time`
`startTime`
`vehiclePath`
`vehiclePosition`

INVARIANTS

`inv1` : $vehiclePath \in Vehicles \pfun paths$
`inv2` : $vehiclePosition \in Vehicles \pfun \mathbb{N}_1$
`inv3` : $dom(vehiclePath) = dom(vehiclePosition)$
`inv4` : $\forall v1, v2 \cdot v1 \in Vehicles \land v2 \in Vehicles \land v1 \neq v2 \land v1 \in dom(vehiclePosition) \land v2 \in dom(vehiclePosition) \land vehiclePath(v1) = vehiclePath(v2) \Rightarrow vehiclePosition(v1) \neq vehiclePosition(v2)$
`inv5` : $\forall v, p \cdot v \in Vehicles \land p \in paths \land v \in dom(vehiclePath) \land p = vehiclePath(v) \Rightarrow vehiclePosition(v) \in 1 \upto pathLen(p)$
`inv6` : $\forall v, h \cdot v \in Vehicles \land h \in Hubs \Rightarrow ((vehicleState(v \mapsto h) = onHub \lor vehicleState(v \mapsto h) = leaving) \Rightarrow v \notin dom(vehiclePath))$

EVENTS

Initialisation

begin

`act1` : $position := startVehicleLocation$
`act2` : $connectionsToTraverse := startConnectionsToTraverse$
`act3` : $hubsToCross := startHubsToCross$
`act4` : $location := startVehicleLocation$
`act5` : $hubLoad := \{h \cdot h \in Hubs | h \mapsto 0\}$
`act6` : $vehicleState := \{v, h \cdot v \in Vehicles \land h \in Hubs \land startVehicleLocation(v) \notin obsHubLocations(h) | (v \mapsto h) \mapsto initial\} \cup \{v, h \cdot v \in Vehicles \land h \in Hubs \land startVehicleLocation(v) \in obsHubLocations(h) | (v \mapsto h) \mapsto onHub\}$
`act7` : $startTime := startVehicleTime$
`act8` : $travelTime := startTravelTime$
`act9` : $time := 0$
`act12` : $vehiclePath := \varnothing$
`act13` : $vehiclePosition := \varnothing$

end

Event $travel \mathrel{\widehat{=}}$
refines $travel$

any

$vehicle$
$newLocation$
r

origin
destination

where

grd1 : $r \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $origin \in stations$
grd4 : $destination \in stations$
grd5 : $origin \neq destination$
grd6 : $origin = connectionOrigin(r(1))$
grd7 : $destination = connectionDestination(r(card(r)))$
grd8 : $location(vehicle) \in obsHubLocations(origin)$
grd9 : $newLocation \in obsHubLocations(destination)$
grd10 : $newLocation \neq location(vehicle)$
grd11 : $position(vehicle) = newLocation$
grd12 : $vehicleState(vehicle \mapsto destination) = onHub$
grd13 : $time \geq startTime(vehicle)$

then

act1 : $location(vehicle) := newLocation$
act2 : $travelTime(vehicle) := time - startTime(vehicle)$

end

Event *traversePath* $\widehat{=}$
Status convergent
refines *traversePath*

any

vehicle
r
p
newPosition

where

grd1 : $vehicle \in Vehicles$
grd2 : $r \in routes$
grd3 : $p \in paths \land p \in ran(r)$
grd4 : $position(vehicle) \in obsHubLocations(connectionOrigin(p))$
grd5 : $newPosition \in obsHubLocations(connectionDestination(p))$
grd6 : $vehicle \mapsto p \in connectionsToTraverse$
grd7 : $vehicle \mapsto connectionOrigin(p) \notin hubsToCross$
grd8 : $vehicle \in dom(vehiclePath) \land vehiclePath(vehicle) = p$
grd9 : $vehicleState(vehicle \mapsto connectionOrigin(p)) = crossed$
grd10 : $vehiclePosition(vehicle) = pathLen(p)$

then

act1 : $position(vehicle) := newPosition$
act2 : $connectionsToTraverse := connectionsToTraverse \setminus \{vehicle \mapsto p\}$
act3 : $vehicleState(vehicle \mapsto connectionDestination(p)) := entering$

end

Event *crossHub* $\widehat{=}$
Status convergent
refines *crossHub*

any

vehicle
hub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $vehicle \mapsto hub \in hubsToCross$
grd5 : $vehicleState(vehicle \mapsto hub) = onHub$

then

act2 : $hubsToCross := hubsToCross \setminus \{vehicle \mapsto hub\}$

act1 : $vehicleState(vehicle \mapsto hub) := leaving$

end

Event *startTravel* $\widehat{=}$

refines *startTravel*

any

$vehicle$

r

where

grd1 : $vehicle \in Vehicles$

grd2 : $r \in routes$

grd3 : $position(vehicle) \in obsHubLocations(connectionOrigin(r(1)))$

grd4 : $vehicleState(vehicle \mapsto connectionOrigin(r(1))) = onHub$

grd5 : $vehicle \notin dom(connectionsToTraverse)$

grd6 : $vehicle \notin dom(hubsToCross)$

then

act1 : $connectionsToTraverse := connectionsToTraverse \cup \{p \cdot p \in ran(r) | vehicle \mapsto p\}$

act2 : $hubsToCross := hubsToCross \cup \{i \cdot i \in 1..card(r) | vehicle \mapsto connectionOrigin(r(i))\}$

act3 : $vehicleState := (vehicleState \lhd \{i \cdot i \in 2..card(r) | vehicle \mapsto connectionOrigin(r(i)) \mapsto initial\}) \lhd \{vehicle \mapsto connectionDestination(r(card(r))) \mapsto initial\}$

act4 : $startTime(vehicle) := time$

end

Event *enterHub* $\widehat{=}$

refines *enterHub*

any

$vehicle$

hub

where

grd1 : $vehicle \in Vehicles$

grd2 : $hub \in Hubs$

grd3 : $position(vehicle) \in obsHubLocations(hub)$

grd4 : $hubLoad(hub) < hubCapacity(hub)$

grd5 : $vehicleState(vehicle \mapsto hub) = entering$

then

act1 : $position(vehicle) := position(vehicle)$

act2 : $hubLoad(hub) := hubLoad(hub) + 1$

act3 : $vehicleState(vehicle \mapsto hub) := onHub$

act5 : $vehiclePath := \{vehicle\} \lhd vehiclePath$

act6 : $vehiclePosition := \{vehicle\} \lhd vehiclePosition$

end

Event *leaveHub* $\widehat{=}$

refines *leaveHub*

any

$vehicle$

hub

r

p

$vehiclesOnPath$

where

grd1 : $r \in routes$

grd2 : $vehicle \in Vehicles$

grd3 : $hub \in Hubs$

grd4 : $position(vehicle) \in obsHubLocations(hub)$

grd5 : $vehicle \mapsto hub \notin hubsToCross$

grd6 : $vehicleState(vehicle \mapsto hub) = leaving$
grd7 : $hubLoad(hub) \geq 1$
grd8 : $p \in paths \land p \in ran(r)$
grd9 : $hub = connectionOrigin(p)$
grd10 : $vehicle \mapsto p \in connectionsToTraverse$
grd11 : $vehiclesOnPath \subseteq Vehicles$
grd12 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land vehiclePath(v) = p | v\}$
grd13 : $\forall v \cdot v \in vehiclesOnPath \Rightarrow vehiclePosition(v) > criticalDistance$

then

act1 : $vehicleState(vehicle \mapsto hub) := crossed$
act2 : $hubLoad(hub) := hubLoad(hub) - 1$
act3 : $vehiclePath(vehicle) := p$
act4 : $vehiclePosition :| (\exists pos \cdot pos \in 1 \mathinner{..} pathLen(p) \land vehiclePosition' = vehiclePosition \Leftarrow \{vehicle \mapsto pos\} \land (\forall v \cdot v \in vehiclesOnPath \land v \neq vehicle \Rightarrow vehiclePosition'(v) - vehiclePosition'(vehicle) \geq criticalDistance))$

end

Event *wait* $\widehat{=}$

refines *wait*

any

vehicle
hub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $hubLoad(hub) \geq hubCapacity(hub)$
grd5 : $vehicleState(vehicle \mapsto hub) = entering$

then

act1 : $position(vehicle) := position(vehicle)$
act2 : $vehicleState(vehicle \mapsto hub) := entering$

end

Event *moveOnPath* $\widehat{=}$

any

vehicle
path
vehiclesOnPath
move

where

grd1 : $vehicle \in Vehicles$
grd2 : $path \in paths$
grd3 : $vehicle \in dom(vehiclePath) \land vehiclePath(vehicle) = path$
grd4 : $vehiclePosition(vehicle) < pathLen(path)$
grd5 : $vehiclesOnPath \subseteq Vehicles$
grd6 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land vehiclePath(v) = path | v\}$
grd7 : $\forall v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) \Rightarrow vehiclePosition(v) - vehiclePosition(vehicle) > criticalDistance$
grd8 : $\{v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) | v\} \neq \varnothing \Rightarrow move \in 1 \mathinner{..} ((min(\{v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) | vehiclePosition(vehicle)\}) - criticalDistance)$
grd9 : $\{v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) | v\} = \varnothing \Rightarrow move \in 1 \mathinner{..} (pathLen(path) - vehiclePosition(vehicle))$

then

act1 : $vehiclePosition(vehicle) := vehiclePosition(vehicle) + move$

end

Event *waitToEnterOnPath* $\widehat{=}$

any
vehicle
path
route
vehiclesOnPath
where
grd1 : $route \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $path \in paths \land path \in ran(route)$
grd4 : $position(vehicle) \in obsHubLocations(connectionOrigin(path))$
grd5 : $vehicle \notin dom(vehiclePath)$
grd6 : $vehicle \mapsto path \in connectionsToTraverse$
grd7 : $vehicle \mapsto connectionOrigin(path) \notin hubsToCross$
grd8 : $vehicleState(vehicle \mapsto connectionOrigin(path)) = leaving$
grd9 : $vehiclesOnPath \subseteq Vehicles$
grd10 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePosition) \land v \in dom(vehiclePath) \land vehiclePath(v) = path | v\}$
grd11 : $\neg(\forall v \cdot v \in vehiclesOnPath \Rightarrow vehiclePosition(v) > criticalDistance)$
then
skip
end

Event *waitToMoveOnPath* $\widehat{=}$
any
vehicle
path
vehiclesOnPath
where
grd1 : $vehicle \in Vehicles$
grd2 : $path \in paths$
grd3 : $vehicle \in dom(vehiclePath) \land vehiclePath(vehicle) = path$
grd4 : $vehiclesOnPath \subseteq Vehicles$
grd5 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land v \in dom(vehiclePosition) \land vehiclePath(v) = path | v\}$
grd6 : $\exists v \cdot v \in vehiclesOnPath \land v \neq vehicle \land vehiclePosition(v) > vehiclePosition(vehicle) \land vehiclePosition(v) - vehiclePosition(vehicle) \leq criticalDistance$
then
skip
end

Event *ticTac* $\widehat{=}$
refines *tickTac*
any
t
where
grd1 : $t \in \mathbb{N}$
grd2 : $t > time$
then
act1 : $time := t$
end

END

An Event-B Specification of Movement7
Creation Date: 11 Jul 2011 @ 07:15:32 PM

MACHINE Movement7
REFINES Movement6
SEES Net7, StartState5
VARIABLES

`position`
`connectionsToTraverse`
`hubsToCross`
`location`
`hubLoad`
`vehicleState`
`travelTime`
`time`
`startTime`
`vehiclePath`
`vehiclePosition`
`activationTime`
`blockedVehicles`

INVARIANTS

inv1 : $activationTime \in Vehicles \pfun \mathbb{N}$
inv2 : $activationTime \neq \varnothing \Rightarrow time \leq min(ran(activationTime))$
inv3 : $dom(vehiclePath) \subseteq dom(activationTime) \cup blockedVehicles$
inv4 : $blockedVehicles \subseteq Vehicles$

EVENTS
Initialisation
begin

act1 : $position := startVehicleLocation$
act2 : $connectionsToTraverse := startConnectionsToTraverse$
act3 : $hubsToCross := startHubsToCross$
act4 : $location := startVehicleLocation$
act5 : $hubLoad := \{h \cdot h \in Hubs | h \mapsto 0\}$
act6 : $vehicleState := \{v, h \cdot v \in Vehicles \land h \in Hubs \land startVehicleLocation(v) \notin obsHubLocations(h) | (v \mapsto h) \mapsto initial\} \cup \{v, h \cdot v \in Vehicles \land h \in Hubs \land startVehicleLocation(v) \in obsHubLocations(h) | (v \mapsto h) \mapsto onHub\}$
act7 : $startTime := startVehicleTime$
act8 : $travelTime := startTravelTime$
act9 : $time := 0$
act10 : $vehiclePath := \varnothing$
act13 : $vehiclePosition := \varnothing$
act14 : $activationTime := \varnothing$
act15 : $blockedVehicles := \varnothing$

end

Event $travel \mathrel{\widehat{=}}$
refines *travel*
any

vehicle
newLocation
r
origin
destination

where

grd1 : $r \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $origin \in stations$
grd4 : $destination \in stations$
grd5 : $origin \neq destination$
grd6 : $origin = connectionOrigin(r(1))$
grd7 : $destination = connectionDestination(r(card(r)))$
grd8 : $location(vehicle) \in obsHubLocations(origin)$
grd9 : $newLocation \in obsHubLocations(destination)$
grd10 : $newLocation \neq location(vehicle)$
grd11 : $position(vehicle) = newLocation$
grd12 : $vehicleState(vehicle \mapsto destination) = onHub$
grd13 : $time \geq startTime(vehicle)$
grd14 : $vehicle \in dom(activationTime)$
grd15 : $time = activationTime(vehicle)$

then

act1 : $location(vehicle) := newLocation$
act2 : $travelTime(vehicle) := time - startTime(vehicle)$
act3 : $activationTime := \{vehicle\} \lhd activationTime$

end

Event *traversePath* $\widehat{=}$
extends *traversePath*

any

vehicle
r
p
newPosition

where

grd1 : vehicle $\in$ Vehicles
grd2 : r $\in$ routes
grd3 : p $\in$ paths $\wedge$ p $\in$ ran(r)
grd4 : position(vehicle) $\in$ obsHubLocations(connectionOrigin(p))
grd5 : newPosition $\in$ obsHubLocations(connectionDestination(p))
grd6 : vehicle $\mapsto$ p $\in$ connectionsToTraverse
grd7 : vehicle $\mapsto$ connectionOrigin(p) $\notin$ hubsToCross
grd8 : vehicle $\in$ dom(vehiclePath) $\wedge$ vehiclePath(vehicle) = p
grd9 : vehicleState(vehicle $\mapsto$ connectionOrigin(p)) = crossed
grd10 : vehiclePosition(vehicle) = pathLen(p)

then

act1 : position(vehicle) := newPosition
act2 : connectionsToTraverse := connectionsToTraverse $\setminus$ {vehicle $\mapsto$ p}
act3 : vehicleState(vehicle $\mapsto$ connectionDestination(p)) := entering

end

Event *crossHub* $\widehat{=}$
extends *crossHub*

any

vehicle
hub

where

grd1 : vehicle $\in$ Vehicles
grd2 : hub $\in$ Hubs
grd3 : position(vehicle) $\in$ obsHubLocations(hub)
grd4 : vehicle $\mapsto$ hub $\in$ hubsToCross
grd5 : vehicleState(vehicle $\mapsto$ hub) = onHub

then

act2 : hubsToCross := hubsToCross $\setminus$ {vehicle $\mapsto$ hub}

act1 : $\texttt{vehicleState}(\texttt{vehicle} \mapsto \texttt{hub}) := \texttt{leaving}$

end

Event *startTravel* $\widehat{=}$

refines *startTravel*

any

vehicle

r

where

grd1 : $vehicle \in Vehicles$

grd2 : $r \in routes$

grd3 : $position(vehicle) \in obsHubLocations(connectionOrigin(r(1)))$

grd4 : $vehicleState(vehicle \mapsto connectionOrigin(r(1))) = onHub$

grd5 : $vehicle \notin dom(connectionsToTraverse)$

grd6 : $vehicle \notin dom(hubsToCross)$

grd7 : $vehicle \notin dom(activationTime)$

then

act1 : $connectionsToTraverse := connectionsToTraverse \cup \{p \cdot p \in ran(r) | vehicle \mapsto p\}$

act2 : $hubsToCross := hubsToCross \cup \{i \cdot i \in 1..card(r) | vehicle \mapsto connectionOrigin(r(i))\}$

act3 : $vehicleState := (vehicleState \lhd\!\!\!\!- \{i \cdot i \in 2..card(r) | vehicle \mapsto connectionOrigin(r(i)) \mapsto initial\}) \lhd\!\!\!\!- \{vehicle \mapsto connectionDestination(r(card(r))) \mapsto initial\}$

act4 : $startTime(vehicle) := time$

act5 : $activationTime := activationTime \cup \{vehicle \mapsto time\}$

end

Event *enterHub* $\widehat{=}$

refines *enterHub*

any

vehicle

hub

where

grd1 : $vehicle \in Vehicles$

grd2 : $hub \in Hubs$

grd3 : $position(vehicle) \in obsHubLocations(hub)$

grd4 : $hubLoad(hub) < hubCapacity(hub)$

grd5 : $vehicleState(vehicle \mapsto hub) = entering$

grd7 : $\top$

grd8 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $position(vehicle) := position(vehicle)$

act2 : $hubLoad(hub) := hubLoad(hub) + 1$

act3 : $vehicleState(vehicle \mapsto hub) := onHub$

act5 : $vehiclePath := \{vehicle\} \lhd\!\!\!\!- vehiclePath$

act6 : $vehiclePosition := \{vehicle\} \lhd\!\!\!\!- vehiclePosition$

act7 : $activationTime := activationTime \lhd\!\!\!\!- \{vehicle \mapsto time + hubCrossingTime(hub)\}$

end

Event *leaveHub* $\widehat{=}$

refines *leaveHub*

any

vehicle

hub

r

p

vehiclesOnPath

where

grd1 : $r \in routes$

grd2 : $vehicle \in Vehicles$

grd3 : $hub \in Hubs$
grd4 : $position(vehicle) \in obsHubLocations(hub)$
grd5 : $vehicle \mapsto hub \notin hubsToCross$
grd6 : $vehicleState(vehicle \mapsto hub) = leaving$
grd7 : $hubLoad(hub) \geq 1$
grd8 : $p \in paths \land p \in ran(r)$
grd9 : $hub = connectionOrigin(p)$
grd10 : $vehicle \mapsto p \in connectionsToTraverse$
grd11 : $vehiclesOnPath \subseteq Vehicles$
grd12 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land v \in dom(vehiclePosition) \land vehiclePath(v) = p|v\}$
grd13 : $\forall v \cdot v \in vehiclesOnPath \Rightarrow vehiclePosition(v) > criticalDistance$
grd14 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $vehicleState(vehicle \mapsto hub) := crossed$
act2 : $hubLoad(hub) := hubLoad(hub) - 1$
act3 : $vehiclePath(vehicle) := p$
act4 : $vehiclePosition : |(\exists pos \cdot pos \in 1 \mathinner{..} pathLen(p) \land vehiclePosition' = vehiclePosition \Leftarrow \{vehicle \mapsto pos\} \land (\forall v \cdot v \in vehiclesOnPath \land v \neq vehicle \Rightarrow vehiclePosition'(v) - vehiclePosition'(vehicle) \geq criticalDistance))$
act6 : $activationTime := activationTime \Leftarrow \{vehicle \mapsto time + deltaTime\}$

end

Event *wait* $\widehat{=}$

refines *wait*

any

vehicle
hub
vehiclesOnHub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $hubLoad(hub) \geq hubCapacity(hub)$
grd5 : $vehicleState(vehicle \mapsto hub) = entering$
grd6 : $vehiclesOnHub = \{v | v \in Vehicles \land position(v) \in obsHubLocations(hub) \land (vehicleState(v \mapsto hub) = leaving \lor vehicleState(v \mapsto hub) = onHub)\} \lhd activationTime$
grd7 : $vehiclesOnHub \neq \varnothing$
grd8 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $position(vehicle) := position(vehicle)$
act2 : $vehicleState(vehicle \mapsto hub) := entering$
act3 : $activationTime := activationTime \Leftarrow \{vehicle \mapsto min(ran(vehiclesOnHub))\}$

end

Event *lockOut* $\widehat{=}$

refines *wait*

any

vehicle
hub
vehiclesOnHub

where

grd1 : $vehicle \in Vehicles$
grd2 : $hub \in Hubs$
grd3 : $position(vehicle) \in obsHubLocations(hub)$
grd4 : $hubLoad(hub) \geq hubCapacity(hub)$
grd5 : $vehicleState(vehicle \mapsto hub) = entering$

grd6 : $\top$
grd7 : $vehiclesOnHub = \{v | v \in Vehicles \land position(v) \in obsHubLocations(hub) \land (vehicleState(v \mapsto hub) = leaving \lor vehicleState(v \mapsto hub) = onHub)\} \lhd activationTime$
grd8 : $vehiclesOnHub = \varnothing$
grd9 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $position(vehicle) := position(vehicle)$
act2 : $vehicleState(vehicle \mapsto hub) := entering$
act3 : $activationTime := \{vehicle\} \lhd activationTime$
act4 : $blockedVehicles := blockedVehicles \cup \{vehicle\}$

end

Event $moveOnPath \mathrel{\widehat{=}}$
refines $moveOnPath$

any

$vehicle$
$path$
$vehiclesOnPath$
$move$

where

grd1 : $vehicle \in Vehicles$
grd2 : $path \in paths$
grd3 : $vehicle \in dom(vehiclePath) \land vehiclePath(vehicle) = path$
grd4 : $vehiclePosition(vehicle) < pathLen(path)$
grd5 : $vehiclesOnPath \subseteq Vehicles$
grd6 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePosition) \land v \in dom(vehiclePath) \land vehiclePath(v) = path | v\}$
grd7 : $\forall v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) \Rightarrow vehiclePosition(v) - vehiclePosition(vehicle) > criticalDistance$
grd8 : $\{v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) | v\} \neq \varnothing \Rightarrow move \in 1 .. ((min(\{v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) | vehiclePosition(vehicle)\}) - criticalDistance)$
grd9 : $\{v \cdot v \in vehiclesOnPath \land vehiclePosition(v) > vehiclePosition(vehicle) | v\} = \varnothing \Rightarrow move \in 1 .. (pathLen(path) - vehiclePosition(vehicle))$
grd10 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $vehiclePosition(vehicle) := vehiclePosition(vehicle) + move$
act3 : $activationTime := activationTime \lhd\!\!\!- \{vehicle \mapsto time + deltaTime\}$

end

Event $waitToEnterOnPath \mathrel{\widehat{=}}$
refines $waitToEnterOnPath$

any

$vehicle$
$path$
$route$
$vehiclesOnPath$

where

grd1 : $route \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $path \in paths \land path \in ran(route)$
grd4 : $position(vehicle) \in obsHubLocations(connectionOrigin(path))$
grd5 : $vehicle \notin dom(vehiclePath)$
grd6 : $\top$
grd7 : $vehicle \mapsto path \in connectionsToTraverse$
grd8 : $vehicle \mapsto connectionOrigin(path) \notin hubsToCross$
grd9 : $vehicleState(vehicle \mapsto connectionOrigin(path)) = leaving$
grd10 : $\top$

grd11 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land vehiclePath(v) = path|v\}$
grd12 : $\neg(\forall v \cdot v \in vehiclesOnPath \Rightarrow vehiclePosition(v) > criticalDistance)$
grd14 : $vehiclesOnPath \lhd activationTime \neq \varnothing$
grd13 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $activationTime := activationTime \lhd\!\!+ \{vehicle \mapsto min(activationTime[vehiclesOnPath])\}$

end

Event *lockIn* $\widehat{=}$

refines *waitToEnterOnPath*

any

vehicle
path
route
vehiclesOnPath

where

grd1 : $route \in routes$
grd2 : $vehicle \in Vehicles$
grd3 : $path \in paths \land path \in ran(route)$
grd4 : $position(vehicle) \in obsHubLocations(connectionOrigin(path))$
grd5 : $vehicle \notin dom(vehiclePath)$
grd6 : $\top$
grd7 : $vehicle \mapsto path \in connectionsToTraverse$
grd8 : $vehicle \mapsto connectionOrigin(path) \notin hubsToCross$
grd9 : $vehicleState(vehicle \mapsto connectionOrigin(path)) = leaving$
grd10 : $\top$
grd11 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land vehiclePath(v) = path|v\}$
grd12 : $\neg(\forall v \cdot v \in vehiclesOnPath \Rightarrow vehiclePosition(v) > criticalDistance)$
grd14 : $vehiclesOnPath \lhd activationTime = \varnothing$
grd13 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $blockedVehicles := blockedVehicles \cup \{vehicle\}$
act2 : $activationTime := \{vehicle\} \lhd\!\!- activationTime$

end

Event *waitToMoveOnPath* $\widehat{=}$

refines *waitToMoveOnPath*

any

vehicle
path
vehiclesOnPath

where

grd1 : $vehicle \in Vehicles$
grd2 : $path \in paths$
grd3 : $vehicle \in dom(vehiclePath) \land vehiclePath(vehicle) = path$
grd4 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land v \in dom(vehiclePosition) \land vehiclePath(v) = path|v\}$
grd5 : $\exists v \cdot v \in vehiclesOnPath \land v \neq vehicle \land vehiclePosition(v) > vehiclePosition(vehicle) \land vehiclePosition(v) - vehiclePosition(vehicle) \leq criticalDistance$
grd6 : $vehiclesOnPath \lhd activationTime \neq \varnothing$
grd7 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$

then

act1 : $activationTime := activationTime \lhd\!\!+ \{vehicle \mapsto time + min(activationTime[vehiclesOnPath])\}$

end

Event *lockOnPath* $\widehat{=}$

refines *waitToMoveOnPath*

any
vehicle
path
vehiclesOnPath
where
grd1 : $vehicle \in Vehicles$
grd2 : $path \in paths$
grd3 : $vehicle \in dom(vehiclePath) \land vehiclePath(vehicle) = path$
grd4 : $\top$
grd5 : $vehiclesOnPath = \{v \cdot v \in Vehicles \land v \in dom(vehiclePath) \land v \in dom(vehiclePosition) \land vehiclePath(v) = path | v\}$
grd6 : $\exists v \cdot v \in vehiclesOnPath \land v \neq vehicle \land vehiclePosition(v) > vehiclePosition(vehicle) \land vehiclePosition(v) - vehiclePosition(vehicle) \leq criticalDistance$
grd11 : $vehiclesOnPath \lhd activationTime = \varnothing$
grd10 : $vehicle \in dom(activationTime) \land time = activationTime(vehicle)$
then
act1 : $blockedVehicles := blockedVehicles \cup \{vehicle\}$
act2 : $activationTime := \{vehicle\} \lhd\!\!\!- activationTime$
end

Event $ticTac \mathrel{\widehat{=}}$
refines $ticTac$
any
t
where
grd1 : $activationTime \neq \varnothing$
grd2 : $t = min(ran(activationTime))$
grd3 : $t > time$
then
act1 : $time := t$
end

END

A.2 The platooning system

An Event-B Specification of context
Creation Date: 10 Aug 2011 @ 02:04:44 PM

CONTEXT context

CONSTANTS

`VEHICLES`

`CRITICAL_DISTANCE`

`initial_xpos`

AXIOMS

`axm1` : $VEHICLES \in \mathbb{N}_1$

`axm2` : $VEHICLES \geq 2$

`axm7` : $CRITICAL\_DISTANCE \in \mathbb{N}_1$

`axm3` : $initial\_xpos \in 1 \mathinner{..} VEHICLES \rightarrow \mathbb{N}$

`axm4` : $\forall v \cdot (v \in 1 \mathinner{..} VEHICLES \Rightarrow initial\_xpos(v) = (VEHICLES - v) * (CRITICAL\_DISTANCE + 1))$

END

An Event-B Specification of context_1
Creation Date: 10 Aug 2011 @ 02:04:49 PM

CONTEXT context_1
EXTENDS context
END

An Event-B Specification of context_2
Creation Date: 10 Aug 2011 @ 02:04:51 PM

CONTEXT context_2
EXTENDS context_1
CONSTANTS

MAX_SPEED
MIN_ACCEL
MAX_ACCEL
initial_speed
new_speed
new_xpos
new_xpos_max
new_xpos_min

AXIOMS

axm1 : $MAX\_SPEED \in \mathbb{N}_1$
axm3 : $MAX\_ACCEL \in \mathbb{N}_1$
axm4 : $MIN\_ACCEL \in \mathbb{Z}$
axm5 : $MIN\_ACCEL < 0$
axm61 : $initial\_speed \in 1 \mathinner{..} VEHICLES \to 0 \mathinner{..} MAX\_SPEED$
axm62 : $\forall vehi0 \cdot (vehi0 \in 1 \mathinner{..} VEHICLES \Rightarrow (\exists speed0 \cdot (speed0 \in 0 \mathinner{..} MAX\_SPEED \land initial\_speed(vehi0) = speed0)))$
axm2 : $new\_speed \in (0 \mathinner{..} MAX\_SPEED \times MIN\_ACCEL \mathinner{..} MAX\_ACCEL) \to \mathbb{Z}$
axm22 : $\forall speed1, accel1 \cdot$
$($
$speed1 \in 0 \mathinner{..} MAX\_SPEED \land accel1 \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
$\Rightarrow$
$new\_speed(speed1 \mapsto accel1) = speed1 + accel1$
$)$
axm71 : $new\_xpos \in (\mathbb{N} \times 0 \mathinner{..} MAX\_SPEED \times MIN\_ACCEL \mathinner{..} MAX\_ACCEL) \to \mathbb{N}$
axm72 : $\forall xpos0, speed0, accel0 \cdot$
$($
$(xpos0 \in \mathbb{N} \land speed0 \in 0 \mathinner{..} MAX\_SPEED \land accel0 \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL)$
$\Rightarrow$
$(new\_xpos(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 + speed0 + (accel0 / 2))$
$)$
axm81 : $new\_xpos\_max \in \mathbb{N} \times 0 \mathinner{..} MAX\_SPEED \times MIN\_ACCEL \mathinner{..} MAX\_ACCEL \to \mathbb{N}$
axm82 : $\forall xpos0, speed0, accel0 \cdot ($
$xpos0 \in \mathbb{N} \land speed0 \in 0 \mathinner{..} MAX\_SPEED \land accel0 \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
$\Rightarrow$
$($
$(accel0 = 0 \Rightarrow new\_xpos\_max(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 + MAX\_SPEED)$
$\land$
$(accel0 \neq 0 \Rightarrow new\_xpos\_max(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 + MAX\_SPEED - (((MAX\_SPEED - speed0) * (MAX\_SPEED - speed0)) / (2 * accel0)))$
$)$
$)$
axm91 : $new\_xpos\_min \in \mathbb{N} \times 0 \mathinner{..} MAX\_SPEED \times MIN\_ACCEL \mathinner{..} MAX\_ACCEL \to \mathbb{N}$
axm92 : $\forall xpos0, speed0, accel0 \cdot ($
$xpos0 \in \mathbb{N} \land speed0 \in 0 \mathinner{..} MAX\_SPEED \land accel0 \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
$\Rightarrow$
$($

$$(accel0 = 0 \Rightarrow new\_xpos\_min(xpos0 \mapsto speed0 \mapsto accel0) = xpos0)$$
$$\wedge$$
$$(accel0 \neq 0 \Rightarrow new\_xpos\_min(xpos0 \mapsto speed0 \mapsto accel0) = xpos0 - ((speed0 * speed0)/(2 * accel0)))$$
$$)$$
$$)$$

END

An Event-B Specification of context_3
Creation Date: 10 Aug 2011 @ 02:04:53 PM

CONTEXT context_3
EXTENDS context_2
CONSTANTS

`initial_accel`

AXIOMS

`axm61` : $initial\_accel \in 1 \mathinner{..} VEHICLES \to MIN\_ACCEL \mathinner{..} MAX\_ACCEL$

`axm62` : $\forall vehi0 \cdot (vehi0 \in 1 \mathinner{..} VEHICLES \Rightarrow (\exists accel0 \cdot (accel0 \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL \land initial\_accel(vehi0) = accel0)))$

END

An Event-B Specification of context_4_v2
Creation Date: 10 Aug 2011 @ 02:04:56 PM

CONTEXT context_4_v2
EXTENDS context_3
CONSTANTS

IDEAL_SPEED

ideal_distance

new_accel

AXIOMS

axm1 : $IDEAL\_SPEED \in 0 \mathinner{..} MAX\_SPEED$

axm2 : $IDEAL\_SPEED < MAX\_SPEED$

axm6 : $ideal\_distance \in 0 \mathinner{..} MAX\_SPEED \to \mathbb{N}$

axm7 : $\forall speed0 \cdot (speed0 \in 0 \mathinner{..} MAX\_SPEED \Rightarrow ideal\_distance(speed0) = CRITICAL\_DISTANCE + speed0)$

axm31 : $new\_accel \in (\mathbb{Z} \times 0 \mathinner{..} MAX\_SPEED \times 0 \mathinner{..} MAX\_SPEED) \to \mathbb{Z}$

axm32 : $\forall p\_dist1, p\_speed1, p\_pre\_speed1 \cdot$
$($
$p\_dist1 \in \mathbb{Z} \land p\_speed1 \in 0 \mathinner{..} MAX\_SPEED \land p\_pre\_speed1 \in 0 \mathinner{..} MAX\_SPEED$
$\Rightarrow$
$new\_accel(p\_dist1 \mapsto p\_speed1 \mapsto p\_pre\_speed1) = p\_dist1 - ideal\_distance(p\_speed1) + p\_pre\_speed1 - p\_speed1$
$)$

thm1 : $\forall speed \cdot (speed \in 0 \mathinner{..} MAX\_SPEED \Rightarrow ideal\_distance(speed) \geq CRITICAL\_DISTANCE)$

END

An Event-B Specification of platoon
Creation Date: 11 Jul 2011 @ 07:40:42 PM

MACHINE platoon
SEES context
VARIABLES
xpos0
INVARIANTS
inv1 : $xpos0 \in 1 \mathinner{..} VEHICLES \rightarrow \mathbb{N}$
inv3 : $\forall v \cdot (v \in 2 \mathinner{..} VEHICLES \Rightarrow ((xpos0(v-1) - xpos0(v)) > CRITICAL\_DISTANCE))$
EVENTS
Initialisation
begin
act1 : $xpos0 := initial\_xpos$
end
Event $all\_moves \mathrel{\widehat{=}}$
any
$magic\_xpos$
where
grd1 : $magic\_xpos \in 1 \mathinner{..} VEHICLES \rightarrow \mathbb{N}$
grd2 : $\forall v \cdot (v \in 2 \mathinner{..} VEHICLES \Rightarrow ((magic\_xpos(v-1) - magic\_xpos(v)) > CRITICAL\_DISTANCE))$
then
act1 : $xpos0 := magic\_xpos$
end
END

An Event-B Specification of platoon_1
Creation Date: 11 Jul 2011 @ 07:40:46 PM

MACHINE platoon_1
REFINES platoon
SEES context
VARIABLES
xpos0
vehicle
xpos
INVARIANTS
inv2 : $xpos \in 1 \mathinner{..} VEHICLES \rightarrow \mathbb{N}$
inv3 : $vehicle \in 1 \mathinner{..} VEHICLES + 1$
inv1 : $\forall v \cdot (v \in 2 \mathinner{..} vehicle - 1 \Rightarrow (xpos(v - 1) - xpos(v)) > CRITICAL\_DISTANCE)$
EVENTS
Initialisation
begin
act1 : $xpos0 := initial\_xpos$
act4 : $vehicle := 1$
act2 : $xpos := initial\_xpos$
end
Event *move1* $\widehat{=}$
Status convergent
any
magic_xpos_vehicle
where
grd1 : $vehicle = 1$
grd2 : $magic\_xpos\_vehicle \in \mathbb{N}$
grd4 : $magic\_xpos\_vehicle \geq xpos(vehicle)$
then
act1 : $xpos(vehicle) := magic\_xpos\_vehicle$
act3 : $vehicle := vehicle + 1$
end
Event *move* $\widehat{=}$
Status convergent
any
magic_xpos_vehicle
where
grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $magic\_xpos\_vehicle \in \mathbb{N}$
grd4 : $magic\_xpos\_vehicle \geq xpos(vehicle)$
grd5 : $xpos(vehicle - 1) - magic\_xpos\_vehicle > CRITICAL\_DISTANCE$
then
act1 : $xpos(vehicle) := magic\_xpos\_vehicle$
act2 : $vehicle := vehicle + 1$
end
Event *all_moves* $\widehat{=}$
refines *all_moves*
when
grd1 : $vehicle = VEHICLES + 1$
with
magic_xpos : magic_xpos = xpos
then
act2 : $xpos0 := xpos$

act1 : $vehicle := 1$

end

VARIANT

$(\texttt{VEHICLES} + 1) - \texttt{vehicle}$

END

An Event-B Specification of platoon_2
Creation Date: 11 Jul 2011 @ 07:40:48 PM

MACHINE platoon_2
REFINES platoon_1
SEES context_2
VARIABLES
xpos0
vehicle
xpos
speed
INVARIANTS
inv1 : $speed \in 1 \mathinner{..} VEHICLES \to 0 \mathinner{..} MAX\_SPEED$
EVENTS
Initialisation
begin
act1 : $xpos0 := initial\_xpos$
act3 : $xpos := initial\_xpos$
act4 : $vehicle := 1$
act2 : $speed := initial\_speed$
end
Event *move1_normal* $\widehat{=}$
refines *move1*
any
magic_accel
nspeed
nxpos
where
grd1 : $vehicle = 1$
grd2 : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$
grd4 : $nspeed \in 0 \mathinner{..} MAX\_SPEED$
grd6 : $nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$
with
magic_xpos_vehicle : magic_xpos_vehicle = nxpos
then
act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := nspeed$
end
Event *move1_max* $\widehat{=}$
refines *move1*
any
magic_accel
nspeed
nxpos
where
grd1 : $vehicle = 1$
grd2 : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$
grd4 : $nspeed > MAX\_SPEED$
grd3 : $nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$
with
magic_xpos_vehicle : magic_xpos_vehicle = nxpos

then
act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := MAX\_SPEED$
end

Event *move1_reduce* $\widehat{=}$
refines *move1*
any
$magic\_accel$
$nspeed$
$nxpos$
where
grd1 : $vehicle = 1$
grd2 : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$
grd3 : $nspeed < 0$
grd4 : $nxpos = new\_xpos\_min(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$
with
magic_xpos_vehicle : magic_xpos_vehicle = nxpos
then
act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := 0$
end

Event *move_normal* $\widehat{=}$
refines *move*
any
$magic\_accel$
$nspeed$
$nxpos$
where
grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
grd3 : $nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$
grd4 : $nspeed \in 0 \mathinner{..} MAX\_SPEED$
grd7 : $nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$
grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$
with
magic_xpos_vehicle : magic_xpos_vehicle = nxpos
then
act2 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act3 : $speed(vehicle) := nspeed$
end

Event *move_max* $\widehat{=}$
refines *move*
any
$magic\_accel$
$nspeed$
$nxpos$
where
grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$
grd4 : $nspeed > MAX\_SPEED$
grd3 : $nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$

grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

with

magic_xpos_vehicle : magic_xpos_vehicle = nxpos

then

act2 : $vehicle := vehicle + 1$

act1 : $xpos(vehicle) := nxpos$

act3 : $speed(vehicle) := MAX\_SPEED$

end

Event *move_reduce* $\widehat{=}$

refines *move*

any

magic_accel

nspeed

nxpos

where

grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$

grd2 : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$

grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto magic\_accel)$

grd4 : $nspeed < 0$

grd3 : $nxpos = new\_xpos\_min(xpos(vehicle) \mapsto speed(vehicle) \mapsto magic\_accel)$

grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

with

magic_xpos_vehicle : magic_xpos_vehicle = nxpos

then

act2 : $vehicle := vehicle + 1$

act1 : $xpos(vehicle) := nxpos$

act3 : $speed(vehicle) := 0$

end

Event *all_moves* $\widehat{=}$

refines *all_moves*

when

grd1 : $vehicle = VEHICLES + 1$

then

act2 : $xpos0 := xpos$

act1 : $vehicle := 1$

end

END

An Event-B Specification of platoon_3_0
Creation Date: 11 Jul 2011 @ 07:40:50 PM

MACHINE platoon_3_0
REFINES platoon_2
SEES context_3
VARIABLES

`xpos0`
`vehicle`
`xpos`
`speed`
`d_vehicle`
`accel`

INVARIANTS

`inv1` : $d\_vehicle \in 1 \mathinner{..} VEHICLES + 1$
`inv2` : $accel \in 1 \mathinner{..} VEHICLES \to MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
`inv7` : $(d\_vehicle = VEHICLES + 1) \lor (\forall v \cdot (v \in 2 \mathinner{..} d\_vehicle - 1 \Rightarrow (\exists f1, f2 \cdot (f1 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \land f2 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_m$

$(f1(xpos(v-1) \mapsto speed(v-1) \mapsto accel(v-1)) - f2(xpos(v) \mapsto speed(v) \mapsto accel(v)) > CRITICAL\_DISTANCE))))$

EVENTS

Initialisation

begin

`act1` : $xpos0 := initial\_xpos$
`act3` : $xpos := initial\_xpos$
`act4` : $vehicle := 1$
`act2` : $speed := initial\_speed$
`act5` : $d\_vehicle := 1$
`act6` : $accel := initial\_accel$

end

Event $decide1 \mathrel{\widehat{=}}$
Status convergent

any

$magic\_accel$

where

`grd2` : $vehicle = 1$
`grd1` : $d\_vehicle = 1$
`grd3` : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$

then

`act1` : $d\_vehicle := d\_vehicle + 1$
`act2` : $accel(d\_vehicle) := magic\_accel$

end

Event $decide \mathrel{\widehat{=}}$
Status convergent

any
$magic\_accel$
where
grd1 : $vehicle = 1$
grd2 : $d\_vehicle \in 2 \mathinner{..} VEHICLES$
grd3 : $magic\_accel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
grd4 : $\exists g1, g2 \cdot$
$(g1 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \land g2 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_$

$(g1(xpos(d\_vehicle - 1) \mapsto speed(d\_vehicle - 1) \mapsto accel(d\_vehicle - 1))$
$-g2(xpos(d\_vehicle) \mapsto speed(d\_vehicle) \mapsto magic\_accel) > CRITICAL\_DISTANCE)$
)
then
act1 : $d\_vehicle := d\_vehicle + 1$
act2 : $accel(d\_vehicle) := magic\_accel$
end
Event $move1\_normal \;\widehat{=}$
refines $move1\_normal$
any
$nspeed$
$nxpos$
where
grd1 : $vehicle = 1$
grd2 : $d\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed \in 0 \mathinner{..} MAX\_SPEED$
grd6 : $nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
with
magic_accel : `magic_accel` $=$ `accel(vehicle)`
then
act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := nspeed$
end
Event $move1\_max \;\widehat{=}$
refines $move1\_max$
any
$nspeed$
$nxpos$
where
grd1 : $vehicle = 1$
grd2 : $d\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed > MAX\_SPEED$
grd3 : $nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
with
magic_accel : `magic_accel` $=$ `accel(vehicle)`
then
act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := MAX\_SPEED$
end
Event $move1\_reduce \;\widehat{=}$
refines $move1\_reduce$
any
$nspeed$

nxpos

where

grd1 : $vehicle = 1$
grd2 : $d\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd3 : $nspeed < 0$
grd4 : $nxpos = new\_xpos\_min(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$

with

magic_accel : $\texttt{magic\_accel} = \texttt{accel(vehicle)}$

then

act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := 0$

end

Event *move_normal* $\widehat{=}$

refines *move_normal*

any

nspeed
nxpos

where

grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $d\_vehicle = VEHICLES + 1$
grd3 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed \in 0 \mathinner{..} MAX\_SPEED$
grd7 : $nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

with

magic_accel : $\texttt{magic\_accel} = \texttt{accel(vehicle)}$

then

act2 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act3 : $speed(vehicle) := nspeed$

end

Event *move_max* $\widehat{=}$

refines *move_max*

any

nspeed
nxpos

where

grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $d\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed > MAX\_SPEED$
grd3 : $nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

with

magic_accel : $\texttt{magic\_accel} = \texttt{accel(vehicle)}$

then

act2 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act3 : $speed(vehicle) := MAX\_SPEED$

end

Event *move_reduce* $\widehat{=}$

refines *move_reduce*

any

nspeed

$nxpos$

where

grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$

grd2 : $d\_vehicle = VEHICLES + 1$

grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$

grd4 : $nspeed < 0$

grd3 : $nxpos = new\_xpos\_min(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$

grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

with

magic_accel : magic_accel = accel(vehicle)

then

act2 : $vehicle := vehicle + 1$

act1 : $xpos(vehicle) := nxpos$

act3 : $speed(vehicle) := 0$

end

Event $all\_moves \mathrel{\widehat{=}}$

refines all_moves

when

grd1 : $vehicle = VEHICLES + 1$

grd2 : $d\_vehicle = VEHICLES + 1$

then

act2 : $xpos0 := xpos$

act1 : $vehicle := 1$

act3 : $d\_vehicle := 1$

end

VARIANT

(VEHICLES + 1) − d_vehicle

END

An Event-B Specification of platoon_4_0
Creation Date: 11 Jul 2011 @ 07:40:52 PM

MACHINE platoon_4_0
REFINES platoon_3_0
SEES context_4_v2
VARIABLES

- xpos0
- vehicle
- xpos
- speed
- d_vehicle
- accel
- p_vehicle
- p_speed
- p_pre_speed
- p_dist

INVARIANTS

- inv4 : $p\_vehicle \in 1 \mathinner{..} VEHICLES + 1$
- inv1 : $p\_speed \in 1 \mathinner{..} VEHICLES \rightarrow 0 \mathinner{..} MAX\_SPEED$
- inv5 : $(p\_vehicle = VEHICLES + 1) \lor (\forall v \cdot (v \in 1 \mathinner{..} p\_vehicle - 1 \Rightarrow p\_speed(v) = speed(v)))$
- inv2 : $p\_dist \in 2 \mathinner{..} VEHICLES \rightarrow \mathbb{Z}$
- inv7 : $(p\_vehicle = VEHICLES + 1) \lor (\forall v \cdot (v \in 2 \mathinner{..} p\_vehicle - 1 \Rightarrow p\_dist(v) = xpos(v - 1) - xpos(v)))$
- inv3 : $p\_pre\_speed \in 2 \mathinner{..} VEHICLES \rightarrow 0 \mathinner{..} MAX\_SPEED$
- inv6 : $(p\_vehicle = VEHICLES + 1) \lor (\forall v \cdot (v \in 2 \mathinner{..} p\_vehicle - 1 \Rightarrow p\_pre\_speed(v) = speed(v - 1)))$

EVENTS
Initialisation
begin

- act1 : $xpos0 := initial\_xpos$
- act3 : $xpos := initial\_xpos$
- act4 : $vehicle := 1$
- act2 : $speed := initial\_speed$
- act5 : $d\_vehicle := 1$
- act6 : $accel := initial\_accel$
- act10 : $p\_vehicle := 1$
- act7 : $p\_speed := initial\_speed$
- act8 : $p\_pre\_speed := \{1\} \lhd initial\_speed$
- act9 : $p\_dist := \{1\} \lhd initial\_xpos$

end
Event *perceive1* $\widehat{=}$
Status convergent
when

- grd1 : $vehicle = 1$
- grd2 : $d\_vehicle = 1$
- grd3 : $p\_vehicle = 1$

then

- act2 : $p\_speed(p\_vehicle) := speed(p\_vehicle)$

act1 : $p\_vehicle := p\_vehicle + 1$

end

Event *perceive* $\widehat{=}$

Status convergent

when

grd1 : $vehicle = 1$

grd2 : $d\_vehicle = 1$

grd3 : $p\_vehicle \in 2 \mathinner{..} VEHICLES$

then

act2 : $p\_speed(p\_vehicle) := speed(p\_vehicle)$

act3 : $p\_dist(p\_vehicle) := xpos(p\_vehicle - 1) - xpos(p\_vehicle)$

act4 : $p\_pre\_speed(p\_vehicle) := speed(p\_vehicle - 1)$

act1 : $p\_vehicle := p\_vehicle + 1$

end

Event *decide1_normal* $\widehat{=}$

refines *decide1*

any

naccel

where

grd1 : $vehicle = 1$

grd2 : $d\_vehicle = 1$

grd4 : $p\_vehicle = VEHICLES + 1$

grd3 : $naccel = IDEAL\_SPEED - p\_speed(d\_vehicle)$

grd5 : $naccel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$

with

magic_accel : $\texttt{magic\_accel} = \texttt{naccel}$

then

act1 : $d\_vehicle := d\_vehicle + 1$

act2 : $accel(d\_vehicle) := naccel$

end

Event *decide1_max* $\widehat{=}$

refines *decide1*

any

naccel

where

grd5 : $vehicle = 1$

grd1 : $d\_vehicle = 1$

grd2 : $p\_vehicle = VEHICLES + 1$

grd3 : $naccel = IDEAL\_SPEED - p\_speed(d\_vehicle)$

grd4 : $naccel > MAX\_ACCEL$

with

magic_accel : $\texttt{magic\_accel} = \texttt{MAX\_ACCEL}$

then

act1 : $d\_vehicle := d\_vehicle + 1$

act2 : $accel(d\_vehicle) := MAX\_ACCEL$

end

Event *decide1_min* $\widehat{=}$

refines *decide1*

any

naccel

where

grd5 : $vehicle = 1$

grd4 : $d\_vehicle = 1$

grd3 : $p\_vehicle = VEHICLES + 1$

grd2 : $naccel = IDEAL\_SPEED - p\_speed(d\_vehicle)$

grd1 : $naccel < MIN\_ACCEL$

with
magic_accel : magic_accel = MIN_ACCEL
then
act2 : $d\_vehicle := d\_vehicle + 1$
act1 : $accel(d\_vehicle) := MIN\_ACCEL$
end

Event *decide_normal* $\widehat{=}$
refines *decide*
any
naccel
where
grd1 : $vehicle = 1$
grd2 : $d\_vehicle \in 2 \mathinner{..} VEHICLES$
grd4 : $p\_vehicle = VEHICLES + 1$
grd5 : $naccel = new\_accel(p\_dist(d\_vehicle) \mapsto p\_speed(d\_vehicle) \mapsto p\_pre\_speed(d\_vehicle))$
grd6 : $naccel \in MIN\_ACCEL \mathinner{..} MAX\_ACCEL$
grd3 : $\exists g1, g2 \cdot$
$(g1 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \wedge g2 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_$

$(g1(xpos(d\_vehicle - 1) \mapsto speed(d\_vehicle - 1) \mapsto accel(d\_vehicle - 1))$
$-g2(xpos(d\_vehicle) \mapsto speed(d\_vehicle) \mapsto naccel) > CRITICAL\_DISTANCE)$
$)$
with
magic_accel : magic_accel = naccel
then
act1 : $d\_vehicle := d\_vehicle + 1$
act2 : $accel(d\_vehicle) := naccel$
end

Event *decide_max* $\widehat{=}$
refines *decide*
any
naccel
where
grd3 : $vehicle = 1$
grd2 : $d\_vehicle \in 2 \mathinner{..} VEHICLES$
grd1 : $p\_vehicle = VEHICLES + 1$
grd5 : $naccel = new\_accel(p\_dist(d\_vehicle) \mapsto p\_speed(d\_vehicle) \mapsto p\_pre\_speed(d\_vehicle))$
grd6 : $naccel > MAX\_ACCEL$
grd4 : $\exists g1, g2 \cdot$
$(g1 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \wedge g2 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_$

$(g1(xpos(d\_vehicle - 1) \mapsto speed(d\_vehicle - 1) \mapsto accel(d\_vehicle - 1))$
$-g2(xpos(d\_vehicle) \mapsto speed(d\_vehicle) \mapsto MAX\_ACCEL) > CRITICAL\_DISTANCE)$
$)$
with
magic_accel : magic_accel = MAX_ACCEL
then
act2 : $d\_vehicle := d\_vehicle + 1$
act1 : $accel(d\_vehicle) := MAX\_ACCEL$
end

Event *decide_min* $\widehat{=}$
refines *decide*
any
naccel
where
grd3 : $vehicle = 1$

grd2 : $d\_vehicle \in 2 \mathinner{..} VEHICLES$
grd1 : $p\_vehicle = VEHICLES + 1$
grd5 : $naccel = new\_accel(p\_dist(d\_vehicle) \mapsto p\_speed(d\_vehicle) \mapsto p\_pre\_speed(d\_vehicle))$
grd6 : $naccel < MIN\_ACCEL$
grd4 : $\exists g1, g2 \cdot$
$(g1 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_min\} \land g2 \in \{new\_xpos, new\_xpos\_max, new\_xpos\_$
$(g1(xpos(d\_vehicle - 1) \mapsto speed(d\_vehicle - 1) \mapsto accel(d\_vehicle - 1))$
$-g2(xpos(d\_vehicle) \mapsto speed(d\_vehicle) \mapsto MIN\_ACCEL) > CRITICAL\_DISTANCE)$
)

with
magic_accel : magic_accel = MIN_ACCEL
then
act2 : $d\_vehicle := d\_vehicle + 1$
act1 : $accel(d\_vehicle) := MIN\_ACCEL$
end

Event *move1_normal* $\widehat{=}$
refines *move1_normal*
any
nspeed
nxpos
where
grd1 : $vehicle = 1$
grd2 : $d\_vehicle = VEHICLES + 1$
grd7 : $p\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed \in 0 \mathinner{..} MAX\_SPEED$
grd6 : $nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
then
act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := nspeed$
end

Event *move1_max* $\widehat{=}$
refines *move1_max*
any
nspeed
nxpos
where
grd1 : $vehicle = 1$
grd2 : $d\_vehicle = VEHICLES + 1$
grd7 : $p\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed > MAX\_SPEED$
grd3 : $nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
then
act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := MAX\_SPEED$
end

Event *move1_reduce* $\widehat{=}$
refines *move1_reduce*
any
nspeed
nxpos
where

grd1 : $vehicle = 1$
grd2 : $d\_vehicle = VEHICLES + 1$
grd7 : $p\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd3 : $nspeed < 0$
grd4 : $nxpos = new\_xpos\_min(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$

then

act3 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act2 : $speed(vehicle) := 0$

end

Event *move_normal* $\widehat{=}$
refines *move_normal*

any

nspeed
nxpos

where

grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $d\_vehicle = VEHICLES + 1$
grd7 : $p\_vehicle = VEHICLES + 1$
grd3 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed \in 0 \mathinner{..} MAX\_SPEED$
grd5 : $nxpos = new\_xpos(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

then

act2 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act3 : $speed(vehicle) := nspeed$

end

Event *move_max* $\widehat{=}$
refines *move_max*

any

nspeed
nxpos

where

grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $d\_vehicle = VEHICLES + 1$
grd7 : $p\_vehicle = VEHICLES + 1$
grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed > MAX\_SPEED$
grd3 : $nxpos = new\_xpos\_max(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

then

act2 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act3 : $speed(vehicle) := MAX\_SPEED$

end

Event *move_reduce* $\widehat{=}$
refines *move_reduce*

any

nspeed
nxpos

where

grd1 : $vehicle \in 2 \mathinner{..} VEHICLES$
grd2 : $d\_vehicle = VEHICLES + 1$
grd7 : $p\_vehicle = VEHICLES + 1$

grd5 : $nspeed = new\_speed(speed(vehicle) \mapsto accel(vehicle))$
grd4 : $nspeed < 0$
grd3 : $nxpos = new\_xpos\_min(xpos(vehicle) \mapsto speed(vehicle) \mapsto accel(vehicle))$
grd6 : $xpos(vehicle - 1) - nxpos > CRITICAL\_DISTANCE$

then

act2 : $vehicle := vehicle + 1$
act1 : $xpos(vehicle) := nxpos$
act3 : $speed(vehicle) := 0$

end

Event $all\_moves \mathrel{\widehat{=}}$

refines all_moves

when

grd1 : $vehicle = VEHICLES + 1$
grd2 : $d\_vehicle = VEHICLES + 1$
grd3 : $p\_vehicle = VEHICLES + 1$

then

act2 : $xpos0 := xpos$
act1 : $vehicle := 1$
act3 : $d\_vehicle := 1$
act4 : $p\_vehicle := 1$

end

VARIANT

$(\texttt{VEHICLES} + 1) - \texttt{p\_vehicle}$

END