Rapid Creation of Interactive Formal Prototypes for Validating Safety-Critical Systems

Inaugural-Dissertation

zur
Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Lukas Ladenberger

aus Heydebreck-Cosel

Düsseldorf, November 2016

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent:	Prof. Dr. Michael Leuschel Heinrich-Heine-Universität Düsseldorf
Koreferent:	Prof. Dr. Peter Gorm Larsen Aarhus University

Tag der mündlichen Prüfung: 07.02.2017

“One Picture Is Worth Ten Thousand Words.”

Origin Unknown

Abstract

The application of formal methods to the development of interactive safety-critical systems usually involves a multidisciplinary team with different roles and expertise (e.g. formal engineers, user interface designers and domain experts). For instance, it is important for the formal engineer to get feedback from the domain expert and the user interface engineer to further develop the formal specification. On the other hand, the domain expert needs to check whether his expectations of the application domain are met in the formal specification and the user interface engineer needs to prevent conflicts between the system's functionality and the user interface design. In general, these tasks deal with the question "Are we building the right system?" and are typically performed with of *validation* techniques.

Animation is a popular validation technique for state-based formal methods such as classical-B and Event-B. The purpose of animation is to inspect the desired behavior of a formal specification by *executing* it. It can also be used to walk-through and analyze scenarios within a multidisciplinary team. However, a formal specification typically becomes complex and large which can make the analysis of a specific aspect of the system using animation difficult and error prone even for formal engineers. Furthermore, while formal engineers have the necessary expertise (e.g. the mathematical notation often used in formal methods) for performing animation techniques, other roles such as user interface engineers or domain experts may not be well versed in formal methods. Consequently, animation techniques may become inaccessible to non-formal method experts.

In this thesis, we present a novel graphical environment called *BMotionWeb* that provides features for the lightweight validation of interactive safety-critical systems by creating *interactive formal prototypes*. An interactive formal prototype complements the use of animation with *interactive data visualization*, a technique to support human understanding by viewing and interacting with pictures or diagrams rather than by examining a substantial amount of data (e.g. numerical or textural data). As the famous saying states, "one picture is worth ten thousand words". We present a reference implementation for BMotionWeb based on the ProB animation engine which enables the rapid creation of interactive formal prototypes for the state-based formal methods classical-B and Event-B and present an extension for the event-based formal method CSPM. In order to demonstrate the use of BMotionWeb, we present various case studies, including interactive systems, industrial case studies, and case studies for teaching formal methods.

Zusammenfassung

Der Einsatz von formalen Methoden für die Entwicklung von interaktiven sicherheitskritischen Systemen involviert typischerweise verschiedene Rollen und Fachkenntnisse, wie zum Beispiel Formale-Methoden-Ingenieure, Benutzeroberflächen-Entwickler und Domänen-Experten. So ist zum einen der Formale-Methoden-Ingenieur auf Feedback vom Domänen-Experten und dem Benutzeroberflächen-Entwickler angewiesen, anderseits ist es die Aufgabe vom Domänen-Experten zu prüfen, ob die formale Spezifikation die Erwartungen bezüglich der Domäne erfüllt. Weiterhin muss der Benutzeroberflächen-Entwickler verhindern, dass es keine Fehlanpassungen zwischen der Systemfunktionalität und dem Design der Benutzeroberfläche gibt. Im Allgemeinen beschäftigen sich diese Aufgaben mit der Frage "Bauen wir das richtige System?" und werden normalerweise mit Hilfe von Techniken zur *Validierung* ausgeführt.

Eine der bekannten Techniken für die Validierung von zustandsbasierten formalen Methoden wie classical-B und Event-B ist *Animation*. Der Zweck von Animation ist es eine formale Spezifikation *auszuführen*, um so die Inspektion vom Systemverhalten zu ermöglichen. Jedoch werden formale Spezifikationen typischerweise komplex und umfangreich, so dass der Einsatz von Animation für die Analyse von sicherheitskritischen Systemen schwer und fehleranfällig werden kann - sogar für Formale-Methoden-Ingenieure. Zudem haben Domänen-Experten oder Benutzeroberflächen-Entwickler meist unzureichende Kenntnisse im Einsatz von formalen Methoden (zum Beispiel über die mathematische Notation, welche bei formalen Methoden oft eingesetzt wird), was dazu führen kann, dass Animation nicht verwendet werden kann.

In dieser Arbeit präsentieren wir BMotionWeb, ein neues grafisches Tool, das ermöglicht *interaktive formale Prototypen* für die leichtgewichtige Validierung von interaktiven sicherheitskritischen Systemen zu erstellen. Ein interaktiver formaler Prototyp kombiniert Animation mit *interaktiver Datenvisualisierung*, einer Technik, die benutzt wird, um Menschen bei der Analyse von Daten (z.B. numerische or textuelle Daten) zu unterstützen. Dabei folgt interaktive Datenvisualisierung ganz dem Motto "Ein Bild sagt mehr als tausend Worte" und repräsentiert die Daten als Bilder oder Diagramme, welche dann als Grundlage für die Analyse dienen. Wir präsentieren eine Referenzimplementierung von BMotionWeb basierend auf dem ProB Animator, welche das Erstellen von interaktiven formalen Prototypen für die zustandsbasierten formalen Methoden classical-B und Event-B und der eventbasierten formalen Methode CSPM ermöglicht. Um den Nutzen von BMotionWeb zu demonstrieren, präsentieren wir verschiedene Fallbeispiele einschließlich interaktive Systeme, industrielle Fallbeispiele und Fallbeispiele für das Unterrichten von formalen Methoden.

Preface

The foundation for this work was laid in my bachelor and master theses at the research department for software techniques and programming languages (STUPS) at the university of Düsseldorf. The topic of both theses was related to formal methods, one of the main research areas of STUPS. It concerned the development of a tool for creating visualizations of formal specifications. A first prototype of the tool was developed in my bachelor thesis and then improved for industrial use in my master thesis. The tool is called BMotionStudio and is still in use in the industry and for teaching formal methods. This, of course, makes me proud, since the results of bachelor and master theses are unfortunately often discarded or never used any more. At that time, I had no idea that my journey at the university and especially the work on BMotionStudio would continue.

While I was working on my master thesis in year 2010, a spin-off company called Formal Mind was founded by Michael Jastram, who was at that time a Ph.D. student at STUPS, and by my supervisor Michael Leuschel. The goal of Formal Mind was to bring the formal method tools and techniques developed at STUPS into the industry. Since Formal Mind was looking for employees, they asked me to be a member of their team right after I finished my master thesis. I accepted the proposition without further ado since it was very interesting: I would be able to combine the work in the Formal Mind company with a Ph.D. position at the university of Düsseldorf. Thus, I had the chance to be part of building up a young company, to gain experiences in research and to complete my Ph.D.

When beginning my Ph.D., the first challenge was to find a research topic which was not easy as expected. However, after some time, I decided to extend the work from my master thesis: the further development of BMotionStudio. Beside research and writing on my thesis, I also had the chance to participate at two EU projects: I joined the final phase of the Deploy project (from 2011 to 2012) and participated in the full Advance project (from 2012 to 2014). Furthermore, I had the possibility to attend different conferences. For instance, I attended the ICFEM in Paris, the SEFM in Vienna, and the IFM in Pisa. I am very proud of my five years of experiences in the STUPS team and very thankful for Michael Leuschel for giving me the opportunities that I had.

I need also to thank many others who supported me during this time. First of all, I am thankful for my steering committee Michael Leuschel and Peter Gorm Larsen, and to Michael Jastram my mentor at the beginning of my Ph.D. time. They gave me this chance and saw the potential of the BMotionStudio tool. I am also thankful to all other colleagues of the STUPS team who gave me useful feedback about my research and played a lot of fun games of foosball with me. In alphabetical order: Carl Friedrich Bolz, Joy Clark, Ivalyo Dobrikov, Nadine Elbeshausen, Fabian Fritz, Stefan Hallerstede, Dominik Hansen, Philip Hoefges, Sebastian Krings, Philipp Koerner, Daniel Plagge,

David Schneider, Ingo Weigelt, Harald Wiegard, and John Witulski. Furthermore, I want to say thanks to the persons who have contributed to this work and have used the BMotionStudio tool and gave me feedback and bug reports. Special thanks goes to my co authors of my research papers: Michael Leuschel, Ivalyo Dobrikov and Jens Bendisposto.

Finally, I warmly thank my family. Firstly, I want to thank my wife Annette Ladenberger who has always supported me and token care of our little daughter Lena, especially during the busy phases of my work. She also supported me in a difficult phase during my studies helped me back on track. Of course, I want to thank my parents who supported me all the time, especially during my studies and at the beginning of my Ph.D. They did everything possible for my academic career. Furthermore, I thank my parents-in-law and my parents for taking care of our daughter on the days when my wife and I were busy with our jobs.

In summary, I can say that the time doing my Ph.D. at the university was very exciting. You will find the results of this time in this work. I hope you enjoy reading it.

Contents

“One Picture Is Worth Ten Thousand Words.”
Origin Unknown

1 Introduction and Motivation

1.1. Research Context

When entering an aircraft, a train, or a car, hardly anyone thinks about that she or he is entering a so called *safety-critical system*. A safety-critical system is a system in which a failure could lead to noticeable economic damage, significant property damage or even human loss. They are part of our daily life: according to the German “Statistischem Bundesamt”[1] 181 million people traveled by plane, and 2613 million people traveled by train in Germany in 2013. Nowadays, safety-critical systems become more and more dependent on computers which are responsible for their *correct* operation. For instance, cars are equipped with an increasing number of electronics and software, e.g. cruise control systems, lane departure warning systems, and even auto-pilot systems.

Classical approaches for the development of (safety-critical) systems involve humans activities. For instance, software engineers write source code, user interface designers develop the interface between the operating end user and the software, and domain experts share their domain knowledge with other persons involved in the project. Unfortunately, humans can make mistakes, and mistakes can lead to software failures. An example of a famous software failure is the fail start of the first test flight of the Ariane 5 rocket on 4 June 1996: 37 seconds after launch the rocket self-destructed because of a malfunction in the software [Dow97]. The result was a great economic loss and property damage. Another example is the collision between a Boeing and a Tupolev aircraft with 71 victims over the Bodensee in southern Germany on 1 July 2002 [Flu04]. Different factors led to this tragic accident including human errors and misunderstanding. The decisive factor, however, was a conflicting order from the TCAS (Traffic Alert and Collision Avoidance System) and from the air traffic control. According to the german

[1]https://www.destatis.de.

accident report [Flu04], the integration of the TCAS was insufficiently formalized: the different aviation regulations were not standardized and were incomplete and partially contradictory.

As a consequence of such incidents and with an increasing number of safety-critical systems which accompany us in our daily life, there is a great demand for engineering methods that help to ensure reliability and robustness of the system and that uncover failures before rolling the system out. One approach to overcome this challenge is to apply *formal methods*.

Formal methods in a nutshell. Formal methods allow the specification and analysis of (safety-critical) systems based on mathematical techniques with the main goal of ensuring *reliability* and *robustness* in the system.

A wide range of formal methods for different applications exist [Alm+11; BN98; CW96]. For instance, formal methods tailored to hardware systems, software systems, distributed systems or real-time systems and can also be used for the analysis of user interfaces. Current research also deals with the application of formal methods to cyber-physical systems [BG11; Adv; DES]. In this thesis, we are mainly concerend with the state-based formal methods classical-B [Sch01; Abr96] and its successor Event-B [Abr10]. While classical-B aims at specifying and analyzing software systems, Event-B is typically used in the field of reactive systems.

When applying formal methods, a *formal specification* of the system is developed. A formal specification contains an (abstract) model of the system that provides a rigorous description of the system. Based on the formal specification, *verification* and *validation* techniques can be used to verify the (safety-critical) properties of the system and to validate the desired behaviour of the system. The overall goal of verification is to check if the specification fulfils the requirements and to ensure the correctness of the system. In other words: verification helps to answer the question “Are we building the system right?”. Common formal verification techniques are formal proof and model-checking [CGP99]. Although verification can help to guarantee the correctness of the system, you may end up with a system that is correct but that doesn’t behave as expected. Especially non-functional requirements are hard to express and to verify in a formal way. Here is where validation comes into play. Validation deals with the question “Are we building the right system?”. *Animation* [Bic+97] is a popular validation technique in the field of state-based formal methods such as classical-B and Event-B. The basic idea of animation is to analyze a formal specification by *executing* it. It explores the reachable states of the specification by evaluating transitions and exposes the information encoded in the states (e.g. concrete values of state variables) to the formal engineer. Thus, the formal engineer can inspect the desired behavior and analyze specific states in the system at any stage of the development process. This can also promote the collaborative work with other roles and levels of expertise usually involved in developing safety-critical systems (e.g. user interface designers, domain experts and end users).

As an example, using animation a formal engineer can walk through scenarios with a domain expert and check whether the domain expert's expectations of the application domain are met in the formal specification. This work is concerned with such animation techniques.

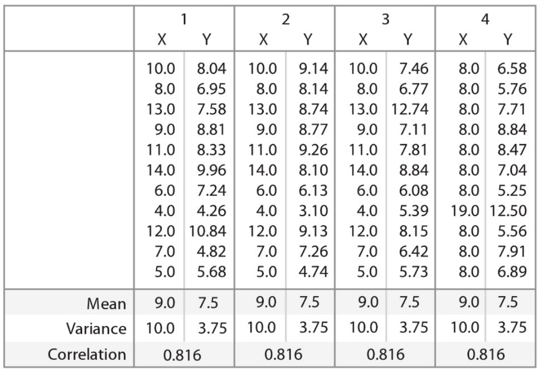
1.2. Research Problem and Question

A formal specification typically becomes complex and large in the process of developing a real-life, industrial system. The amount of details, such as the number of state variables and transitions of the formal specification, may increase during the implementation of the system. This can make the analysis of a specific aspect or state using animation techniques difficult and error prone, even for formal engineers. Another challenge that can arise while applying animation techniques is that they require a certain level of knowledge about the mathematical notation to understand the meaning of a specific state. While formal engineers provide the necessary expertise in formal methods, other roles such as domain experts may only have partial or even no knowledge of formal methods. As a consequence, animation techniques may become inaccessible to non-formal method experts and inadequate for collaborative work.

Data visualization [TG83; Mun14], on the other hand, is known as a tool to communicate some data (e.g. numerical or textual data) by creating visual representations of the data. As the well known proverb "one picture is worth ten thousand words"[2] says, the objective of data visualization is to support human understanding of the data by viewing some pictures or diagrams rather than by examining a substantial amount of the data. For this, data visualization can take advantage of the *pre-attentive* processing skills of humans [War12]: it is easier for a human to identify specific aspects, characteristics, or properties of the data when they are different in size, orientation, or color. An effective visual representation also depends not only on the data to be visualized but also on the target audience that is going to use the visualization. For instance, plot diagrams or bar charts can provide a global overview and may help statistical analysts make comparisons of some numerical data. An alternative (visual) view of the data being analyzed can also be of particular use if the original data set is not readable by analysts, e.g. because of a foreign presentation of the data or the language used in the data. Consider the example from [Ans73] shown in Fig. 1.1. The figure visualizes the data from the Anscombe's Quartett [Ans73], a famous example for demonstrating the power of data visualization. The left side illustrates the data as a series of four small datasets each with eleven (x,y) points and with almost identical statistical properties (mean, variance and correlation, and linear regression lines). The right side of Fig. 1.1 shows the visual representation of the four datasets as scatter plot diagrams. These alternative (visual) views are good examples to demonstrate how they can support the analysis and interpretation of some data: although the datasets have almost identical statistical properties, we can clearly

[2]The proper origin of the proverb is unknown.

see that the structures of the four datasets are quite different although this is not obvious when just looking at the table. For instance, we can better understand the determination of the regression lines (e.g. due to outliers).



	1		2		3		4	
	X	Y	X	Y	X	Y	X	Y
	10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
	8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
	13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
	9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
	11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
	14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
	6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
	4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
	12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
	7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
	5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89
Mean	9.0	7.5	9.0	7.5	9.0	7.5	9.0	7.5
Variance	10.0	3.75	10.0	3.75	10.0	3.75	10.0	3.75
Correlation	0.816		0.816		0.816		0.816	

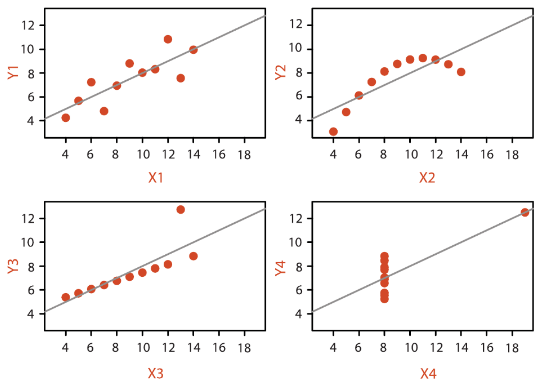


Figure 1.1.: Anscombe's Quartet [Ans73] raw data (left side) and visual representation as scatter plot diagrams taken from [Mun14] (right side)

Another important aspect for effective data visualization is *interactivity*. An *interactive system* is a computer system that is concerned with the interaction between humans and the computer. Dix et al. (2003) define the purpose of an interactive system as follows: "Traditionally, the purpose of an interactive system is to aid a user in accomplishing *goals* from some application *domain*" [Dix+03]. Examples for interactive systems are command line interfaces, operating systems (e.g. Windows and OS X) and graphical user interfaces (GUI). *Interactivity* is the defining feature of an interactive system and is also closely related to data visualization, where users can interact with the visualization, e.g. changing parameters and seeing the effect [Dix+03].

Based on these principles we define the following research question for this thesis:

"How can animation benefit from interactive data visualization to support the validation process of formal specifications and to make animation techniques more accessible to non-formal method experts?"

1.3. Thesis Goal and Contribution

To answer the question posed in Section 1.2, this thesis aims at achieving the following goals: (1) identify appropriate interactive visual representations for animated formal specifications and (2) develop an approach (method and tool) for linking the identified interactive visual representations to animated formal specifications.

In respect to these goals, the contributions of this thesis are outlined below:

State-of-the-art of animation-based visualization approaches. This contribution aims at assessing the *state-of-the-art* of visualization approaches in the field of formal methods. The result of the research is grouped into different classes of formal methods: state-based formal methods, process algebras and other formal methods for specifying real time systems and distributed systems, as well as variants of the $\lambda-calculus$. We present the advantages and limitations of the visualization approaches and analyze them with respect to the research problem stated in Section 1.2.

Interactive formal prototypes of state-based formal specifications. The key contribution of this thesis is an approach for the rapid creation of *interactive formal prototypes*. An interactive formal prototype combines animation with an interactive domain specific visualization[3] to support the validation of safety-critical systems. We present an appropriate method and implementation for supporting *state-based* formal methods.

Interactive formal prototypes of CSPM formal specifications. The second contribution of this thesis extends the approach for creating interactive formal prototypes for state-based formal specifications to *event-based* formal specifications. In this context, we present a method, implementation and case-studies based on CSPM, a formal language mainly used for specifying concurrent and distributed systems.

Combining interactive formal prototypes with other visualization techniques. Based on the assessment of the state-of-the-art research, this thesis is also concerned with identifying existing visualization techniques that could be combined with interactive formal prototypes. Two visualization techniques and their link to interactive formal prototypes are presented: *projection diagrams* and *trace diagrams*.

Application of interactive formal prototypes for validating safety-critical systems. This contribution consists of various example applications of interactive formal prototypes for supporting the validation process of safety-critical systems. We present case studies for the lightweight validation of interactive systems, for validating industrial applications, and for teaching formal methods.

1.4. Thesis Organization

This thesis is organized as follows: the current chapter (Chapter 1) aims at presenting the reader with the research context, the motivation and the thesis goals and contributions. The organization of the rest of this thesis is based on its contributions.

Chapter 2 discusses the state-of-the-art in visualization techniques in the field of formal methods.

Chapter 3 presents the first part of the key contribution of this thesis: a method for creating interactive formal prototypes of state-based formal specifications. In this context, we describe the concept of an interactive formal prototype in more detail.

[3]We give a more detailed definition of the term *domain specific visualization* in Section 3.2.2.

Chapter 4 presents the tool that implements the method presented in Chapter 3 called BMotionWeb. It describes the goals for BMotionWeb, gives an overview of the design and architecture of the tool, and describes its features.

Chapter 5 deals with other visualization techniques that could be linked with interactive formal prototypes to support the validation process of formal specifications. Two visualization techniques and their links are presented in this chapter: *projection diagrams* and *trace diagrams*.

Chapter 6 presents an extension of BMotionWeb to support the creation of interactive formal prototypes for the CSPM specification language. It describes the challenges and explains the difference to the state-based visualization approach presented in Chapter 3.

Chapter 7 describes various example applications of BMotionWeb, including industrial case studies, interactive systems, and the use of BMotionWeb for teaching formal methods.

Finally, Chapter 8 draws the conclusions and discusses future work.

1.5. Thesis Publications

Parts of this thesis have been published in the following peer-reviewed articles, conference proceedings and journals.

Visualising Event-B Models with B-Motion Studio [LBL09]

This paper presents the beginnings of the approach presented in this thesis: BMotion-Studio as a tool for creating domain specific visualizations of Event-B specifications. The tool comes as a plug-in for the Rodin platform. Moreover, a case study of a Event-B simple lift system is demonstrated in the paper. My contribution consists of the described approach, the entire implementation work, and the creation of the case study.

An Approach for Creating Domain Specific Visualisations of CSP Models [LDL14]

While the work presented in [LBL09] focuses on creating domain specific visualizations for the state-based formal method Event-B, this paper aims at presenting an approach towards creating domain specific visualizations for event-based formal methods. The paper describes a method, an implementation, and two case studies based on the CSPM language. My contribution was the method, the implementation, and the case studies in which the method was developed with Ivaylo Dobrikov.

Mastering the Visualization of Larger State Spaces with Projection Diagrams [LL15]

This paper presents an approach for creating so called *projection diagrams* with the goal to considerably reduce the size of large state spaces. Moreover, an extension of the approach is demonstrated to combine a projection diagram with a domain specific visualization. An appropriate algorithm and implementation is presented and the application of the approach is demonstrated based on several case studies. My contribution consists of: (1) developing the algorithm for creating a projection diagram further (the first design of the basic algorithm was developed by Michael Leuschel); (2) formalizing the algorithm and (3) combining the projection diagram with a domain specific visualization.

BMotionWeb: A Tool for Rapid Creation of Formal Prototypes [LL16]

BMotionWeb is the successor of BMotionStudio presented in [LBL09]. The paper describes the architecture and features of the tool and demonstrates its use for validating interactive systems. Two case studies are presented: the graphical user interface of a simple phonebook application and a cruise control system device. My contribution consists of the entire development of BMotionWeb and the elaboration of the two case studies.

Validation of the ABZ landing gear system using ProB [Han+14; Lad+15]

As part of the ABZ'14 conference, a case study of a landing gear system was proposed. This paper presents the formalization of the case study in Event-B, its validation using the ProB toolkit, and the development of a domain specific visualization. The paper has been extended in the journal version [Lad+15]. The journal version extends the Event-B specification of the landing gear system considerably and describes in more detail an extended version of the domain specific visualization. My contribution consists of developing the domain specific visualization.

Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation [Hoa+16]

The hemodialysis machine was the second case study proposed by the ABZ'16 conference. This paper presents the development of an Event-B specification using the iUML-B tool, the validation of the requirements, and the design of the hemodialysis machine. Moreover, a domain specific visualization has been created. Finally, it demonstrates the use of Co-Simulation techniques to validate the dynamic behaviour of the hemodialysis machine. My contribution consists of developing the domain specific visualization.

2 State-of-the-Art: Visualizing Formal Specifications

Visualization is a common approach to support the analysis and development of formal specifications. It is often used to create a graphical representation of some (abstract) data or piece of information within a formal specification. This chapter aims at assessing the *state-of-the-art* of visualization approaches with respect to the research problem stated in Section 1.2 and according to a set of criteria identified in Section 2.1. Since this thesis is concerned with animation techniques, we mainly focus on state-of-the-art visualization approaches which are based on animation.

The result of the state-of-the-art assessment is grouped into different classes of formal methods. The first section (Section 2.2) deals with state-based formal methods. Section 2.3 takes a closer look at process algebras. Finally, Section 2.4 considers other related formal methods that also provide visualization approaches. In each section, we first give a brief introduction of the characteristics of the respective formal method class and introduce a running example that is used to support the assessment where possible.[1] Subsequently, we discuss and compare the visualization approaches.

2.1. Set of Criteria

We introduce a set of criteria (in respect to the research problem stated in Section 1.2) to give a precise analysis of the studied visualization approaches:

- *Visual representation:* A main goal of this thesis is to identify appropriate visual representations for animated formal specifications. Thus, we take special account of the visual representations that are used and their motivations for the studied approaches.
- *Interactivity:* In Chapter 1, we have already stated that interactivity is an important aspect of data visualization. As a consequence, we also take special account

[1]Some of the considered tools and visualization techniques are obsolete and unavailable. As a consequence, the running example is only used for the assessment of tools and visualization techniques which are available for use.

of the interactive features of the studied approaches.

- *Additionally required user knowledge:* Non-formal method experts typically are not versed in formal methods (e.g. they have only little or even no knowledge about the mathematical notation of a formal method). In the course of studying the visualization approaches, we analyze the knowledge a user must have in order to *use* and *understand* a visualization. This also covers input needed from the user to produce a visualization (e.g. mathematical formulas).
- *Additionally required developer knowledge:* Most visualizations are generated automatically based on the given data. However, custom visualizations, where the mapping between the data and the visual representation are created manually, typically require some additional engineering skills (e.g. programming skills). Thus, we also consider the skills needed to *develop* a visualization.

2.2. State-Based Formal Methods

This section deals with visualization approaches for state-based formal methods. In state-based formal methods the system behaviour is typically described by *states* and *transitions*. A state is a particular configuration of variables, whereas transitions link two states and describe how the system evolves. Some states are marked as *initial* and the set of states and transitions reachable from the initial state is also called the *state space* of the specification. A prominent example for a state-based formal method is the B-method. There are two specification languages associated with the B-method: classical-B [Sch01; Abr96] and its successor Event-B [Abr10]. While classical-B aims at specifying and analysing software systems, Event-B is typically applied in the field of reactive systems.[2] Other state-based formal methods that are considered in this section are Z [SA92], TLA$^+$ [Lam02], ASM [BS12] and VDM [Jon86]. Most visualization approaches for state-based formal methods have the common goal of visualizing the information that constitutes the states of the system, e.g. values of variables and transitions. This covers graphical visualization approaches (see Section 2.2.2), graph visualization approaches like state space visualization (see Section 2.2.3), graphical user interface frontends of animation tools (see Section 2.2.4), and other visualization approaches (see Section 2.2.5).

2.2.1. Running Example: Simple Lift System

In this section, we use the example of a simple lift system to support the assessment of the different visualizations techniques. The lift system allows the movement of a single lift cabin between a finite number of floors and the opening and closing of the lift cabin

[2]Since this thesis is mainly concerned with the B-method, we will give an extended description of the B-method later in Section 4.2.1.

Table 2.1.: Overview of graphical visualization approaches for state-based formal methods

Approach / Tool	Supported Formalisms	Visualization Technology	Interactivity	Required Knowledge (Developer)
Brama	classical-B, Event-B	Flash	yes	Flash, ActionScript
AnimB	Event-B	Flash / Web	unknown	Flash, ActionScript / HTML, JavaScript
ProB (Flash)	classical-B	Flash	no	Flash, Java
BMotionStudio	Event-B	Java (Draw2D)	yes	-
Overture	VDM	Java	yes	Java
JeB	Event-B	Web	no	HTML5, JavaScript
WebASM	ASM	Web	unknown	HTML, JavaScript
ProB (Tcl/Tk)	classical-B, Z	Tcl/Tk	no	-
Lively Wal-Through (ViennaTalk)	VDM-SL	SmallTalk	yes	LiveTalk

door. The user can request the lift on a specific floor by pressing a request button that is installed on each floor.

Since this section deals with several state-based formal methods, we have created a formal specification of the simple lift system for each formalism that is considered in this section. This covers formal specifications written in classical-B, Event-B and VDM-SL. The formal specifications can be found in Appendix A.1.

2.2.2. Graphical Visualization

Graphical visualization is a common visualization approach in the field of state-based formal methods. The basic idea of creating a graphical visualization for state-based formal methods is to map each state in a formal specification to a proper graphical representation. However, the implementation and the usability vary greatly from one approach to another. This has implications for the required knowledge, time and resources for creating the graphical visualization. In this context, we take special account of the techniques provided by the studied approaches to map a state and its graphical representation (we define this as *gluing code*).

Table 2.1 gives a quick overview of the studied approaches and their supported formalisms, the visualization technologies that are used, and supported tools. Moreover, the table shows the assessment according to the set of criteria defined in Section 2.1. The following subsections are categorized based on the visualization technologies used by the studied approaches. In summary, we discuss approaches based on Flash, Java GUI-toolkits, web-technologies, and other technologies like Tcl/Tk and SmallTalk. In each subsection, we first give a brief introduction of the considered approach and illustrate it with a graphical visualization of the simple lift system (see Section 2.2.1) created with the respective approach, followed by an analysis.

Flash-based approaches

The tools Brama [Ser06], ProB [BL07] and AnimB [Mét] provide support for creating graphical visualizations using Flash[3] as the underlying graphical engine. Brama was designed by ClearSy and comes as a plug-in for the Rodin platform [Abr+10]. It is primarily an animator with support for the B-method (classical-B and Event-B), however it also provides a Flash extension for creating graphical visualizations. In Brama the developer creates the graphical representation of the formal specification using widgets which are provided by Flash (e.g. labels and images), and the gluing code is realized with the built-in scripting language *ActionScript*. Brama was used successfully for several industrial strength applications like the DOF1 system [Engc; Engb] and COPP system [Enga] for opening and closing the platform doors on line 1 and line 13 of the Paris subway.

Another tool with support for Flash-based graphical visualizations is AnimB [Mét]. Like Brama, AnimB is primarily an animator that comes as a plug-in for the Rodin platform.

The ProB tool also proposed a Flash-based approach for creating graphical visualizations [BL07] of classical-B specifications which comes as a plug-in for the Eclipse version of ProB [Ben06b].[4] The basic idea of the approach is similar to Brama: the developer creates the graphical representation using the build-it widgets in Flash and writes code for the gluing code. However, instead of writing ActionScript, it requires the user to write Java.

Although, the use of Flash provides a powerful tool for creating rich interactive graphical visualizations, it involves some disadvantages for the developer: since a Flash based tool is a self-contained tool, the developer of a visualization requires skills for using it. Furthermore, the developer requires additional programming skills for writing the gluing code, e.g. ActionScript for Brama and AnimB, and Java for [BL07]. Thus, creating a graphical visualization using the presented Flash-based approaches may become time consuming and error prone. For instance, with Brama the developer needs approximately one week to perfect the visualization of a formal specification created over two months [Ser06]. The authors of the approach presented in [BL07] state that it took them two days to setup the actual graphical representation based on Flash and less than one hour to write the gluing code (when the developer is already familiar with the Java programming language) for the visualization presented in [BL07]. Moreover, with the rise of HTML5, Flash has become extinct [QSu; Vau01; Pau15].

[3]http://www.adobe.com/devnet/flash.html.

[4]The tool is no longer maintained and available.

Java-based approaches

Most of the animation tools listed in Table 2.1 are implemented in Java or at least contain a part of the tool that is written in Java. These include ProB, BMotionStudio, Brama and Overture. Some of them also support creating graphical visualizations using Java based GUI-toolkits.

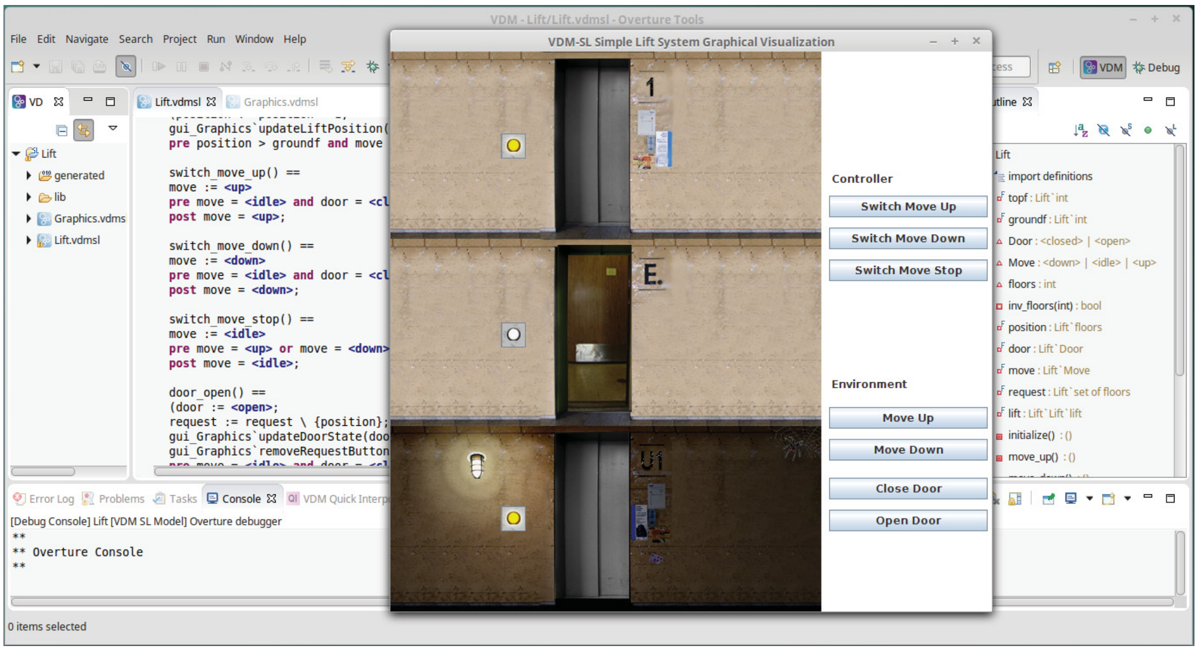
In [NLL12], the authors present an extension for the Overture tool [Spe04a; Spe04b] to combine VDM [Jon86] specifications with executable code. The authors argue that the extension is on the one hand useful to combine VDM specifications with external subsystems which may not be worthwhile to be formalized (e.g. external libraries) and can also be used as a technique to create graphical visualizations for VDM specifications. A graphical visualization is created using Java. Thus, the developer can make use of Java GUI-toolkits like AWT, Swing, or SWT to create the graphical representation of a state. For creating the gluing code, the approach provides two Java interfaces which need to be implemented by the developer: the *Remote Control* interface allows the developer to control the VDM animator (e.g. executing an operation) and the *External Java Library* interface is responsible for updating the state of the graphical representation.

Figure 2.1 demonstrates the visualization of the VDM-SL simple lift system running within the Overture tool. The visualization is made of different widgets (e.g. images and buttons) provided by the Java Swing toolkit [ELW98]. The gluing code is also written in Java and utilizes the Model-View-Controller (MVC) pattern. In order to provide interactive features in the visualization, the buttons (located on the right side) and the images that represent the floor request buttons (located on the left side) are wired with VDM operations.

Although the approach provides a generic tool for creating rich interactive graphical visualizations of VDM specifications, it comes at the price of usability since it requires significant knowledge in Java to create the graphical representation and the gluing code. Even creating visualizations for rather simple specifications may become a time consuming and error prone task. It took us approximately one full working day to create the visualization shown in Fig. 2.1. In particular, a lot of time has been invested to create the skeleton of the Java application which is based on the MVC design pattern. Indeed, the approach would benefit from a feature for automatically generating the skeleton of the Java application that is needed to create a visualization.

Another disadvantage is that in the worst case, a modification of the original specification is needed. For example, the VDL-SL simple lift system specification contains special expressions to call the *External Java Library* interface to update the state of the visualization as illustrated in Fig. 2.2. The *move_down* operation contains a special expression (see line 4) that is responsible for updating the position of the lift cabin in the visualization.

The second Java based tool is called BMotionStudio [LBL09] and supports the creation of graphical visualizations of Event-B specifications. BMotionStudio is integrated into



```
gui_Graphics`updateLiftPosition(
pre position > groundf and move

switch_move_up() ==
move := <up>
pre move = <idle> and door = <cl
post move = <up>;

switch_move_down() ==
move := <down>
pre move = <idle> and door = <cl
post move = <down>;

switch_move_stop() ==
move := <idle>
pre move = <up> or move = <down>
post move = <idle>;

door_open() ==
(door := <open>;
request := request \ {position};
gui_Graphics`updateDoorState(doo
gui_Graphics`removeRequestButton
```

```
[Debug Console] Lift [VDM SL Model] Overture debugger
**
** Overture Console
**
```

```
Lift
import definitions
topf : Lift`int
groundf : Lift`int
Door : <closed> | <open>
Move : <down> | <idle> | <up>
floors : int
inv_floors(int) : bool
position : Lift`floors
door : Lift`Door
move : Lift`Move
request : Lift`set of floors
lift : Lift`Lift`lift
initialize() : ()
move_up() : ()
```

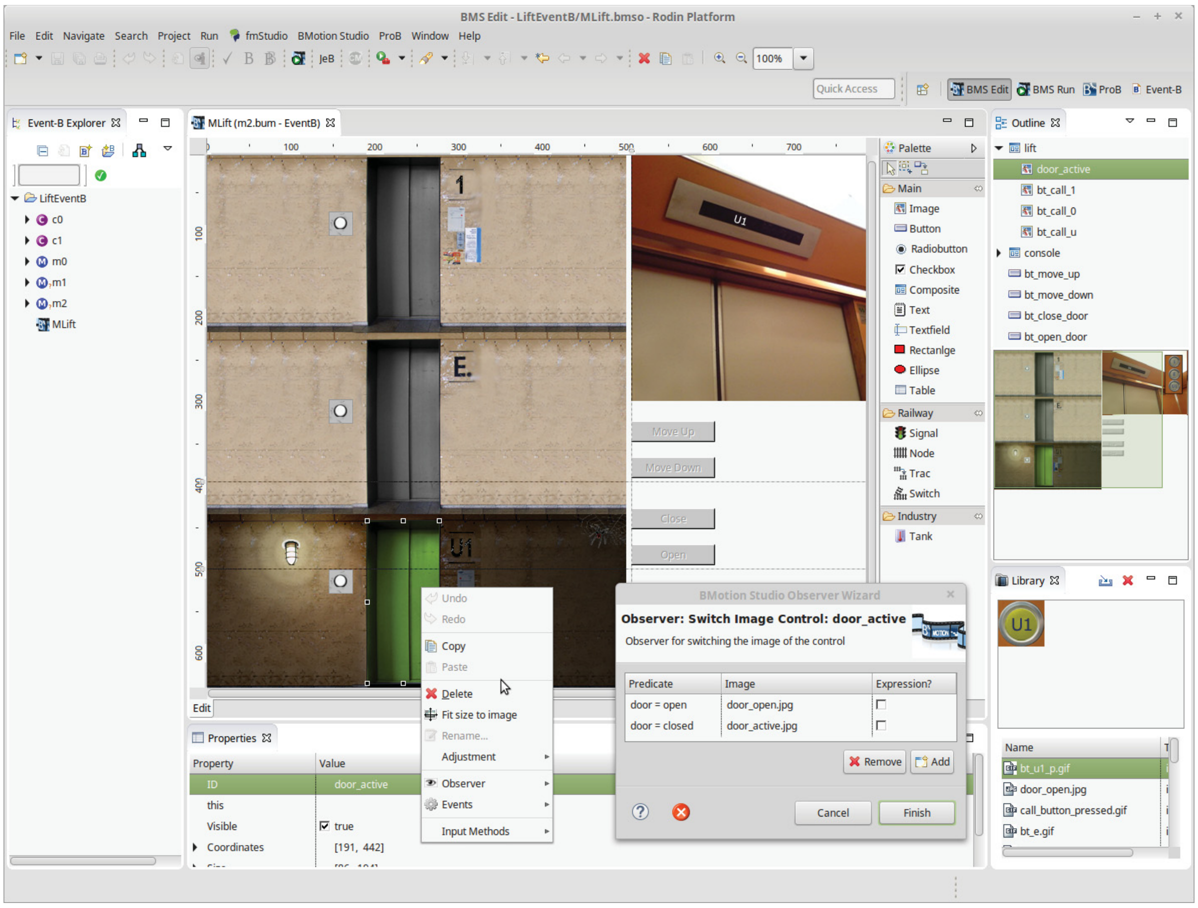
Figure 2.1.: Graphical visualization of the VDM-SL simple lift system created with the approach proposed in [NLL12]

```
move_down: () ==> ()
move_down() ==
(position := position - 1;
gui_Graphics‘updateLiftPosition(position))
pre position > groundf and move = <down>;
```

Figure 2.2.: *move_down* operation of the VDM-SL simple lift system specification

the Rodin platform [Abr+10] and uses the ProB [LB08] animator to execute an Event-B specification. The main motivation of BMotionStudio is to free the developer from writing programming code that creates the graphical representation and that establishes gluing code. Instead, the developer can use the built-in visual editor for this purpose. The visual editor provides several predefined graphical elements, e.g. shapes, labels, images and buttons, as well as well-known features from modern graphical editors, like drag-and-drop, copy/paste or undo/redo. In order to create a link between a graphical element and the state, the developer needs to set-up *observers*. An observer binds a list of formulas (e.g. predicates and expressions) to a graphical element and allows the tool to compute a visualization for any given state of the animated formal specification by changing the properties of the graphical element (e.g. the color or position) according to the evaluation of the formulas in the respective state. Moreover, the visualizations that are produced are interactive. For instance, the developer can wire an Event-B event to a graphical element which is executed when the user clicks on the graphical element.

Figure 2.3 shows the graphical visualization of the Event-B simple lift system created with BMotionStudio. The visualization consists of different graphical elements like images, buttons and containers representing the different parts of the simple lift system and sets up observers in order to create the gluing code. The visualization contains thirteen observers which each describe very simple formulas. For instance, the observer belonging to the graphical element that represents the door will evaluate the predicates $door = open$ and $door = closed$, and depending of their values ($TRUE$ or $FALSE$) the representation of the graphical element will be updated to display a different image (see Fig. 2.3). In addition, some of the graphical elements wire Event-B events. As an example, the graphical element that represents the request button on the first floor (the upper floor) wires the Event-B event *send_request* with the predicate $f = 1$.



BMotion Studio Observer Wizard

Observer: Switch Image Control: door_active

Observer for switching the image of the control

Predicate	Image	Expression?
door = open	door_open.jpg	☐
door = closed	door_active.jpg	☐

Properties

Property	Value
ID	door_active
this	
Visible	☑ true
Coordinates	[191, 442]

Name
bt_u1_p.gif
door_open.jpg
call_button_pressed.gif
bt_e.gif

Figure 2.3.: Creating the graphical visualization of the Event-B simple lift system with BMotionStudio

While BMotionStudio provides a convenient and fast approach to create simple interactive graphical visualizations for Event-B specifications, it is it difficult to use and apply it when creating complex visualizations. In particular, the visual editor is limited in terms of design flexibility and is inefficient when creating dynamic visualizations with numerous or repeated elements. For instance, creating a railroad track layout with a number of tracks, signals and switches or a communication network with several nodes can be very cumbersome. Another limitation of BMotionStudio is the limited reuse of existing graphical elements (e.g. provided by other Java based GUI-toolkits). Although the tool provides an extension mechanism to contribute new graphical element types, it is hard and cumbersome to use it, especially because the user needs Java programming skills and knowledge about the plug-in mechanism in Eclipse.

Web-based approaches

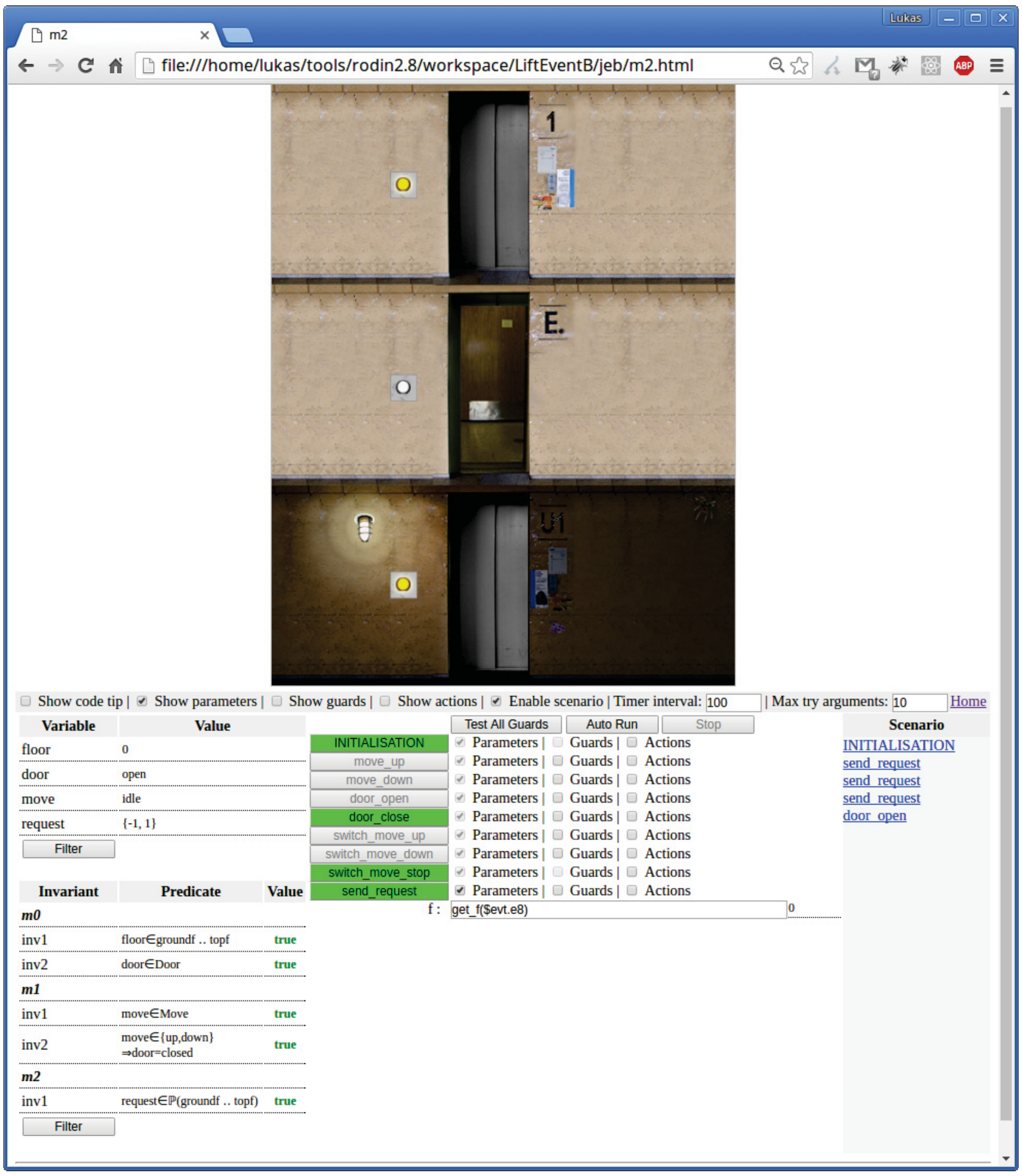
The use of web-technologies for developing formal method tools has become increasingly popular with the rise of HTML5 [W3C14] and an increasing number of tool developers bring more and more formal specification languages into the web.

The first web-based tool we are considering is called JeB [Yan13]. JeB is a framework for Event-B that is capable of producing a standalone web-based animation engine for an existing Event-B specification. The tool translates a given Event-B specification to JavaScript and generates a JavaScript-based animation engine and an HTML-based user-interface for animating the Event-B specification within a browser. The user-interface of the generated animation also provides an HTML5 canvas for creating a graphical visualization of the the Event-B specification based on HTML5 and JavaScript. In particular, two JavaScript functions needs to be implemented for this purpose: *jeb.animator.init* which initializes the visualization and *jeb.animator.draw* which draws the visualization according to the current state of the animation [Yan13].

The result of creating a graphical visualization for the Event-B simple lift system using JeB is demonstrated in Fig. 2.4. The corresponding HTML5 / JavaScript code can be found in Appendix A.1.2. The upper half of the figure shows the graphical visualization and the lower half of the figure shows the generated user-interface of the animation. It contains a list of variables with corresponding state values, a list of invariants, multiple buttons (one button for each event of the specification), and a list of events that have been executed so far. The user can interact with the animation, e.g. by clicking on an event button. This causes a state change in the animation, and the user interface and the visualization will be updated according to the new state.

Since the visualization is part of the generated standalone animation, it is possible to deploy it on a web-server. This makes the visualization (and animation) accessible from other devices such as tablets and mobile phones and allows sharing the visualization with other stakeholders (e.g. during an online project meeting). However, JeB also makes high demands on the developer of a visualization as JeB requires significant knowledge in HTML5 and JavaScript. Moreover, the approach lacks interactive features (e.g. executing an Event-B event via the visualization).

The third tool in this category, called WebASM [ZGS14] has its origin in the ASM (Abstract State Machine) [BS12] community. WebASM is a web-based tool that brings CoreASM [FGG07], an animation engine for executing ASM specifications into the web.



file:///home/lukas/tools/rodin2.8/workspace/LiftEventB/jeb/m2.html

☐ Show code tip | ✓ Show parameters | ☐ Show guards | ☐ Show actions | ✓ Enable scenario | Timer interval: 100 | Max try arguments: 10 Home

Variable	Value
floor	0
door	open
move	idle
request	{-1, 1}

Event			
INITIALISATION	✓ Parameters	☐ Guards	☐ Actions
move_up	✓ Parameters	☐ Guards	☐ Actions
move_down	✓ Parameters	☐ Guards	☐ Actions
door_open	✓ Parameters	☐ Guards	☐ Actions
door_close	✓ Parameters	☐ Guards	☐ Actions
switch_move_up	✓ Parameters	☐ Guards	☐ Actions
switch_move_down	✓ Parameters	☐ Guards	☐ Actions
switch_move_stop	✓ Parameters	☐ Guards	☐ Actions
send_request	✓ Parameters	☐ Guards	☐ Actions

f : get_f($evt.e8) 0

Scenario
INITIALISATION
send_request
send_request
send_request
door_open

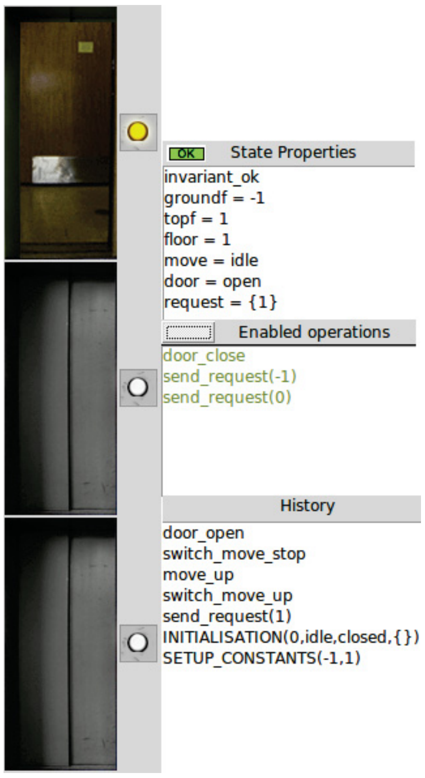
Invariant	Predicate	Value
m0		
inv1	floor∈groundf .. topf	true
inv2	door∈Door	true
m1		
inv1	move∈Move	true
inv2	move∈{up,down} ⇒door=closed	true
m2		
inv1	request∈ℙ(groundf .. topf)	true

Figure 2.4.: Graphical visualization of the Event-B simple lift system created with JeB and running in a web-browser

The tool provides a Java applet that embeds the CoreASM engine and a JavaScript API for communicating with the engine (e.g. for loading an ASM specification and accessing the state information of an ASM specification). Making the executed ASM specification available via JavaScript enables the development of web-based graphical visualizations of ASM specifications. Similar to JeB, WebASM also requires the knowledge of web-technologies like HTML and JavaScript in order to create the graphical representation and the gluing code.

Other approaches

In Section 2.2.2, we have analyzed the ProB approach for creating Flash-based graphical visualizations [BL07]. In addition to Flash, ProB also supports the use of Tcl/Tk [Leu+08] for creating graphical visualizations of classical-B specifications. Tcl is a scripting language that includes the Tk toolkit for creating graphical user interfaces. A major difference between both approaches lies in the creation of the gluing code since the Tcl/Tk based approach does not require additional skills for creating the graphical elements and for writing the gluing code. Instead, it requires an *animation function* written in classical-B to link a state and its graphical representation. As an example, Fig. 2.5 demonstrates the graphical visualization of the classical-B simple lift system using the animation function shown in Fig. 2.6. The visualization is built-on a two-dimensional grid where each cell in the grid can contain a bitmap image. The user can define the animation function and the images for the visualization in the `DEFINITIONS` block of the classical-B specification. An image definition has the form `ANIMATION_IMG`x `==` `"`*filename*`"`, where x is a number and *filename* the path to a bitmap image file. The animation function defines which image should be displayed in which cell. For this, the animation function must be of type `INTEGER * INTEGER +-> INTEGER`, i.e. a mapping from rows and columns to image numbers.


OK State Properties
invariant_ok
groundf = -1
topf = 1
floor = 1
move = idle
door = open
request = {1}

Enabled operations
door_close
send_request(-1)
send_request(0)

History
door_open
switch_move_stop
move_up
switch_move_up
send_request(1)
INITIALISATION(0,idle,closed,{})
SETUP_CONSTANTS(-1,1)

Figure 2.5.: Graphical visualization of the classical-B simple lift system in ProB Tcl/Tk

An interesting aspect of the ProB Tcl/Tk based approach is certainly that the developer

```
DEFINITIONS
 Rconv == (topf-r+groundf);
 ANIMATION_FUNCTION == ({ r,c,i | r : groundf..topf &
                         ((c=0 & i=0) or (c=1 & i=3)) } <+
                          ({ r,c,i | r : groundf..topf & Rconv : request &
                             c=1 & i=4 } \/
                           { r,c,i | r : groundf..topf & Rconv=floor & c=0 &
                             ((door = open & i=1) or (door = closed & i=2)) } ) );
   ANIMATION_IMG0 == "LiftEmpty.gif";
   ANIMATION_IMG1 == "LiftOpen.gif";
   ANIMATION_IMG2 == "LiftClosed.gif";
   ANIMATION_IMG3 == "CallButtonOff.gif";
   ANIMATION_IMG4 == "CallButtonOn.gif"
```

Figure 2.6.: Animation function for visualizing the classical-B simple lift system in ProB Tcl/Tk

of a graphical visualization only needs one notation, namely classical-B. No additional programming skills are needed in order to create the gluing code. However, the visualizations that are produced are rather simple and restricted, e.g. the approach lacks interactive components like buttons that can execute classical-B operations and predefined elements like shapes. Instead, the developer needs to create bitmap images (e.g. JPEG or GIF image files) using external tools which can be a time consuming task. Also, writing the required animation function can still be a considerable challenge (depending on the formal engineer's knowledge of classical-B).

The authors in [Oda+15] present a graphical visualization approach called "Lively Walk-Through". Lively Walk-Though combines VDM animation with a UI sketch and widgets to provide lightweight validation features for VDM-SL specifications. The approach is implemented in Smalltalk [GR83] and integrated into ViennaTalk, a Smalltalk library to work with VDM-SL specifications.[5] It comes with an integrated environment including a visual editor for creating UI widgets like fields, images, and buttons. For creating the gluing code, the approach introduces its own language called "LiveTalk". LiveTalk allows the user to wire interactive actions (e.g. executing an operation or evaluating an expression) to UI widgets. As an example, Fig. 2.7 shows the LiveTalk script for wiring the *send_request*(1) operation to the UI widget that represents the request button located on the upper floor of the simple lift system where *SendRequest1* is the id of the UI widget. All LiveTalk scripts for the VDM-SL simple lift system visualization can be found in Appendix A.1.3.

[5]https://github.com/tomooda/ViennaTalk-doc.

```
SendRequest1‘clicked
  send_request(1)
```

Figure 2.7.: Livetalk snippet for wiring an operation execution with an UI widget.

Table 2.2.: Overview of graph visualization approaches for state-based formal methods

Approach / Tool	**Supported Formalisms**	**Visualization Technology**	**Interactivity**
State Space (ProB)	classical-B, Event-B	Tcl/Tk	no
		D3	yes
Signature Merge (ProB)	classical-B, Event-B	Tcl/Tk	no
		D3	yes
DFA-abstraction (ProB)	classical-B, Event-B	Tcl/Tk	no
Under Approximation (GénéSyst)	classical-B, Event-B	unknown	unknown
Formula Visualization (ProB)	classical-B, Event-B	Tcl/Tk	no
		D3	yes

2.2.3. Graph Visualization

Graph visualization is a popular technique that combines data visualization and geometric graph theory [HMM00]. A graph visualization is typically constructed with nodes and edges, where the nodes are represented by shapes (e.g. rectangles, boxes or circles) and the edges by line segments or curves. They can be used to visualize different concepts such as algorithms or processes. In this section, we discuss graph visualization approaches for state-based formal methods. Table 2.2 gives a quick overview of the studied approaches.

State Space Visualization

For state-based formal methods, a state space can be constructed and validated automatically via model checking [CGP99]. In this process, the validity of temporal properties will be checked, but the state space itself is "invisible" to the user. However, often it is important for the developer or a stakeholder to inspect the state space (or parts of it) manually. This can be achieved interactively with animation or by visualizing the state space of a specification. The latter can be especially useful to get an overview of the system and to identify structural similarities, symmetries, and unanticipated properties within the system [Pre08].

The ProB tool provides support for visualizing the state space for B specifications (classical-B and Event-B), where the state space is constructed automatically by the built-in model-checker or interactively with animation. Two variants of the state space visualization approach exists. The fist variant is a built-in feature of the ProB Tcl/Tk version [LB08] and makes use of Graphviz[6] for producing the visualization. The second variant makes use of D3 [BOH11] and is described in [Cla13]. In both visualization approaches, the states are represented by shapes like ellipses, rectangles or triangles, where each node shows the value of the system variables for that state. Transitions are represented by directed arrows, where each transition is labeled with the name of the classical-B operation or Event-B event which triggered the transition. The difference between the approaches is that the D3-based variant produces interactive visualizations with features like zooming and panning within the visualization.

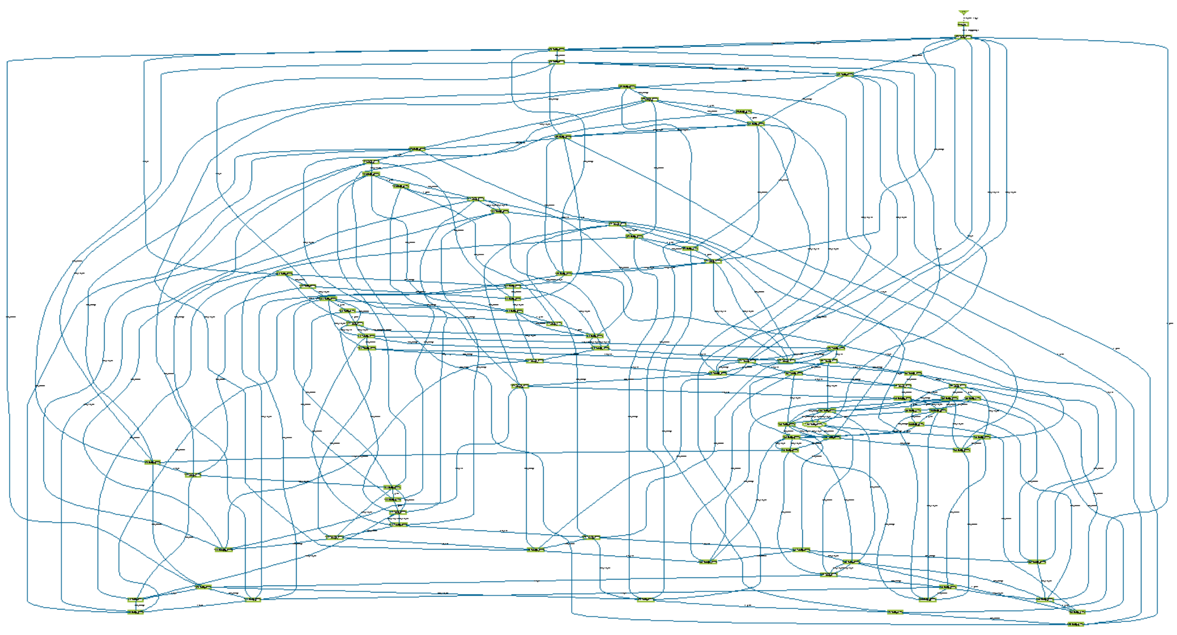


Figure 2.8.: ProB state space visualization of the classical-B simple lift system (full state space)

The number of states and transitions typically becomes very large ("state space explosion" problem [Val98; Pel08]), especially in industrial projects, and human inspection of the state space visualization may thus become a difficult task. For instance, the full state space visualization of the simple lift system covers 86 states and 242 transitions and is shown in Fig. 2.8 (the reader is not expected to be able to read the visualization, just to get a general impression of the problem). Although the visualization is produced for a relatively simple specification, it is still hard for humans to grasp. In the next section we study state space abstraction approaches that are capable of reducing the complexity of state space visualizations.

[6]http://www.graphviz.org/.

Abstraction Techniques

The authors in [LT05] present two state space abstraction approaches: the *signature merge approach* and the *DFA-abstraction algorithm*. Both approaches have been implemented in the ProB tool. The basic idea of the signature merge approach is to merge all states with the same enabled events to a common *signature*. The approach can be tuned by deselecting events from the signature. Although the signature merge visualization that is produced may not be equivalent to the original state space (as far as the sequences of the events are concerned), the approach can result in reducing the size of the state space while still preserving beneficial information. Figure 2.9 shows the signature merge visualization of the classical-B simple lift system. In contrast to the full state space visualization (see Fig. 2.8), the reduction is considerable (the signature merge graph includes only 8 nodes and 14 edges) and the visualization can be analyzed by a human. For instance, one can see that after executing the *switch_move_up* or *switch_move_down* operation, the *switch_move_stop* operation is always enabled in the next state (this is indicated by the solid edges). This is also true for the *door_open* operation: after executing the operation, the *door_close* operation is always enabled in the next state.

The basic idea of the DFA-abstraction algorithm, which is the other algorithm available in ProB is to abstract away from operation arguments and to apply the classical minimization algorithm for Deterministic Finite Automatons (DFA) [ASU86; Hop79]. In contrast to the signature merge approach, the DFA-abstraction algorithm produces a visualization in which the transitions are equivalent to those in the original state space.

In [IL06; IS15], the authors present an abstraction technique based on animation for creating *behavioural views* (visualized as state transition diagrams) from classical-B and Event-B specifications. The animation based technique, also referred as an *under-approximation* approach is similar to the previously mentioned signature merge and DFA-abstraction algorithm, where the actual abstraction is applied to an exhaustively explored state space generated using animation or model-checking tools such as ProB [LB08]. In order to generate the abstracted state transition diagram of the state space the tool takes a disjoint set of user-defined predicates as input, where each predicate corresponds to an abstract state in the abstract view (if the predicates are adequately chosen). Then, the tool checks each predicate in each concrete state using AtelierB [Cle09]. If a predicate is discharged in a concrete state, it is merged with the corresponding abstract state. Thus, each concrete state of the original state space corresponds to an abstract state in the abstract state transition diagram. Moreover, the strict mapping of concrete states to abstract states allows the translation of the transitions of the original state space into the transitions of the abstract state space.

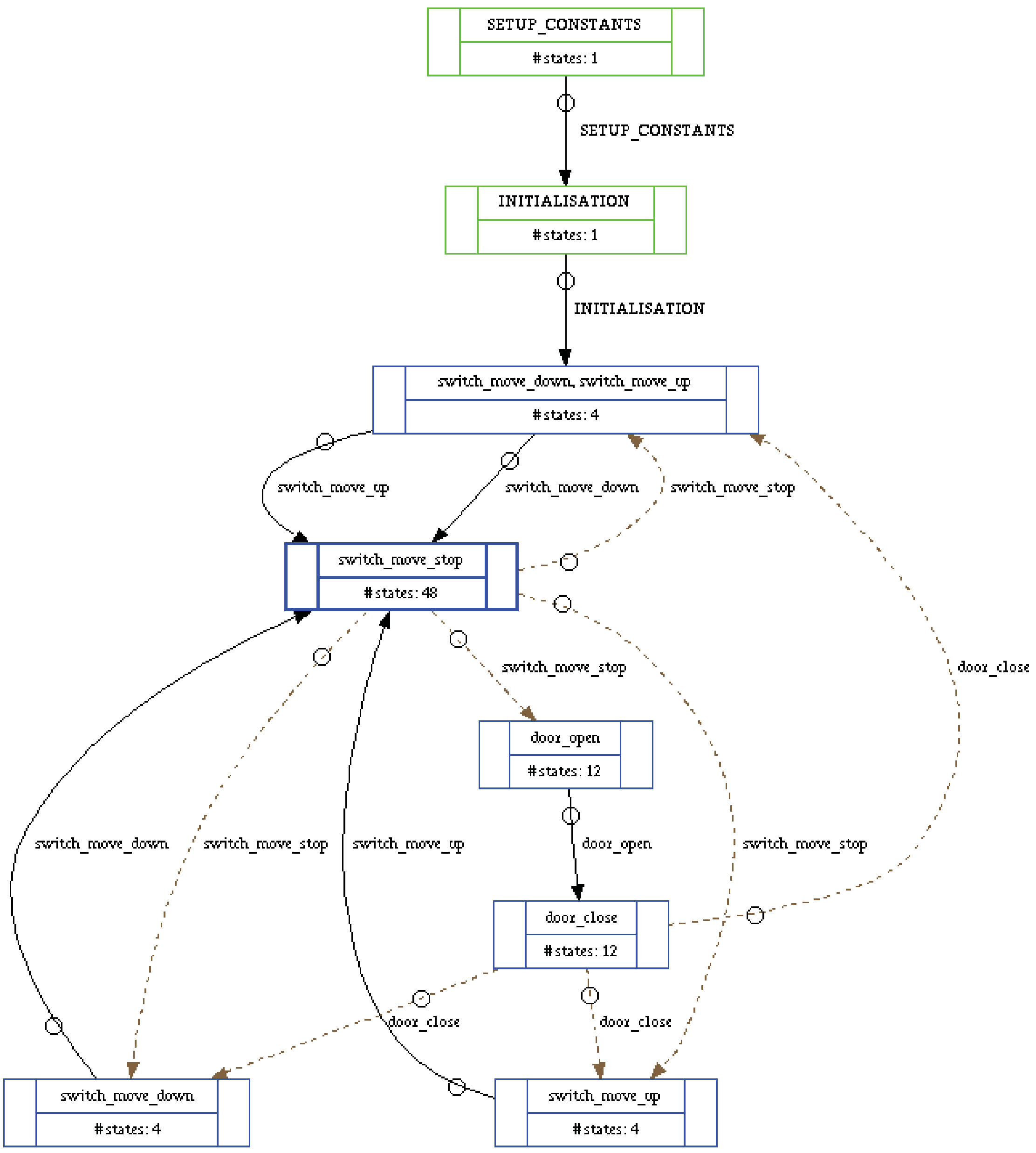


Figure 2.9.: ProB signature merge visualization of the classical-B simple lift system

Formula Visualization

The mathematical notation used in formal methods is often seen as a barrier which prevents users from applying formal methods. Visualization approaches may help to reduce these barriers. In this section, we consider visualization approaches which are capable of visualizing mathematical formulas of state-based formal methods.

The ProB tool supports the visualization of formulas (e.g. expressions and predicates). Two implementations exist: the first one is integrated into the ProB Tcl/Tk version [LB08], and the second one makes use of D3 [BOH11] and is described in [Cla13]. To demonstrate the ProB formula visualization approach, consider Fig. 2.10. The figure gives two examples of predicate visualizations produced with ProB Tcl/Tk. The left side of the figure shows the visualization of the invariant $move \in \{up, down\} \Rightarrow door = closed$, and the right side shows the visualization of the precondition of the *switch_move_up* operation of the classical-B simple lift system. The visualizations are represented as tree graphs (a special graph type), where the original formula is decomposed into sub-formulas. Formula visualizations can be produced at different stages on the animation. The fill color of the nodes indicates whether the (sub)-formula is true (the node is green) or false (the node is red) in the respective state. For instance, the visualizations in Fig. 2.10 have been produced at state $floor = 0, move = up, door = closed$. All nodes in the invariant visualization are colored green indicating that the invariant is satisfied in the respective state. On the other hand, one can see at a glance which sub-formula of the precondition visualization of the *switch_move_up* operation is false (indicated by the red filled nodes) and thus disables the operation in the respective state.

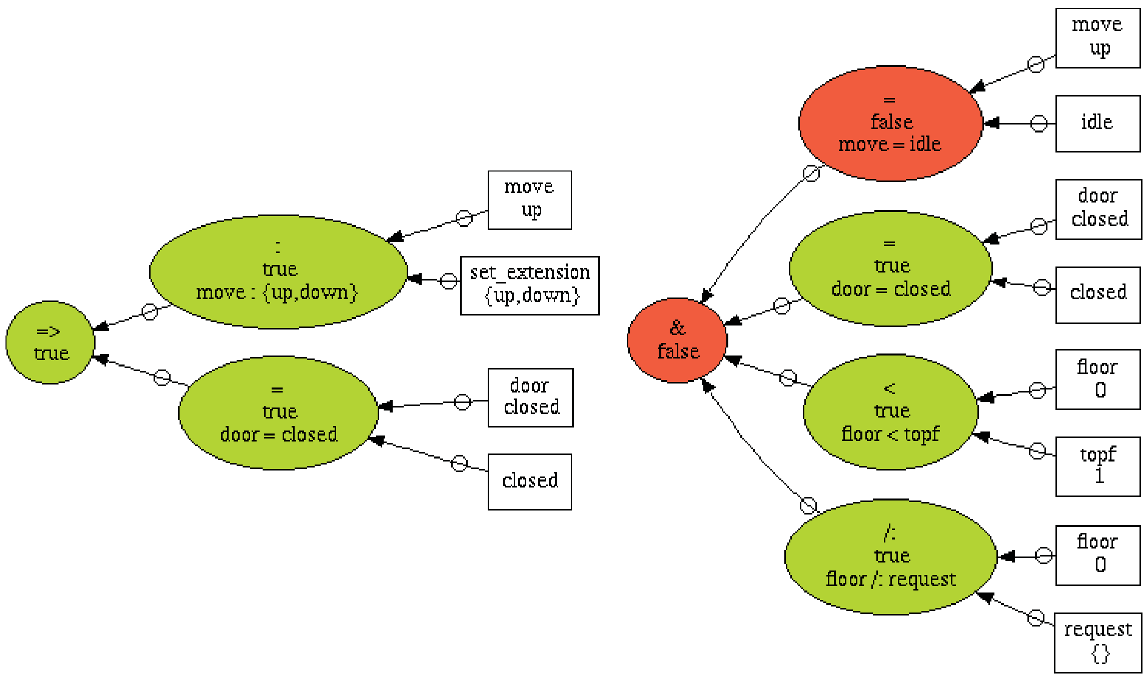


Figure 2.10.: Invariant (left) and guard (right) visualizations in ProB

The D3-based formula visualization approach of ProB is similar to the Tcl/Tk-based approach, as both approaches rely on the same data coming from the ProB animator. However, in contrast to the Tcl/Tk-based approach, the D3-based approached produces interactive visualizations: the user can expand or collapse subformulas by selecting the parent formula within the visualization. Furthermore, it is possible to zoom in and out of the visualization [Cla13]. This is particularly useful if the user is interested in specific parts of the formula or if the visualization becomes too large.

Advantages and Limitations of Graph Visualization Approaches

The graph visualization approaches that have been presented provide several advantages. A major benefit is that they are mainly generated automatically. There is no need to manually create the mapping between the visualization and the formal specification as required by the graphical visualization approaches presented in Section 2.2.2. Moreover, they may be useful to get a global view of the system and to identify structural similarities, symmetries, and unanticipated properties within the system. They can also be used to support human analysis of the specification by highlighting relevant aspects and behaviors of the system as demonstrated by the formula visualization feature and the different abstraction techniques. The abstraction techniques can also considerably reduce the complexity of large state spaces and thus enable the creation of suitable and readable visualizations.

However, the presented graph visualization approaches also have some disadvantages. They still may be to difficult for non-formal method experts since they need a certain level of knowledge about the mathematical notation to understand the meaning of a specific node and edge in the visualization (e.g. a node in the state space visualization that represents a state of the system). Furthermore, the abstraction techniques presented in [IL06; IS15] require the user to provide predicates as input. Obviously this is not trivial, especially if the user is not versed in formal methods.

2.2.4. GUI Frontends of Animation Tools

Some animation tools also provide graphical user interfaces (GUI) frontends such as ProB [LB08; Ben06a; Die09] and JeB [Yan13]. These GUIs typically display information about the current state of the animation (e.g. variable values) and allow the user to interact with the animation (e.g. by executing events). Figure 2.4 and Fig. 2.5 show the animator GUI frontends of JeB and ProB Tcl/Tk respectively. Although such GUIs may support the user in the analysis of formal specifications, they also have some barriers which prevent non-formal method experts from using them. One reason for this is that the GUIs of the animation tools are typically generic, i.e. they provide a default presentation regardless of the characteristics of the animated formal specification (e.g. its application domain). They also assume that the user is familiar with the animated formal specification (e.g. the user knows the meaning and function of events and variable names) and the mathematical notation of the respective formal method (e.g. to understand the meaning of a specific state). Finally, most animation tool GUIs do not scale well. A formal specification typically become very large in the process of developing a system. The amount of details (e.g. the number of variables) of the specification increases with the introduction of new refinement levels. This can make the examination of a specific state difficult as users are often only interested in specific parts of the specification (e.g. certain variables).

In addition to the existing GUIs, there are also approaches for creating *custom* GUIs for animation tools. Gaffe [Dal04] and its successor Gaffe2[7] are such tools. They support the creation of custom GUIs for the ZLive[8] animation tool based on the AWT GUI toolkit (Gaffee) and on the Swing GUI toolkit (Gaffe2).

2.2.5. Other Visualization Approaches

In [Cla13], the author presents a visualization approach called *value over time* that is capable of visualizing the state values of a given formula for a particular trace in an animation. The approach is based on the ProB animator and provides support for B specifications (classical-B and Event-B). Figure 2.11 illustrates the approach by visualizing the value of the *floor* variable while animating the classical-B simple lift system specification. The x-axis shows the animation steps performed so far and the y-axis shows the value of the *floor* variable in the respective animation step.

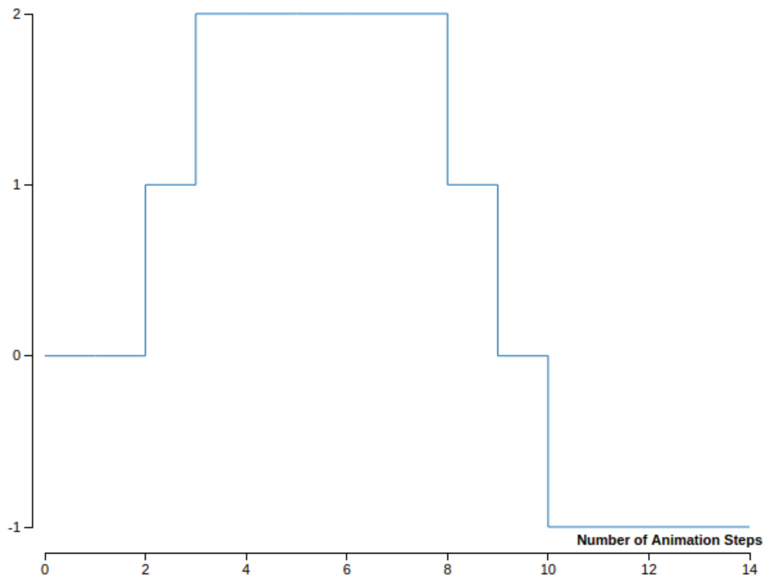


Figure 2.11.: ProB value over time visualization approach visualizing the floor variable of the classical-B simple lift system

[7]http://czt.sourceforge.net/gaffe2.
[8]http://czt.sourceforge.net/zlive.

Table 2.3.: Overview of visualization approaches for process algebras

Tool	Supported Formalisms	Process Visualization	Counter-Example Visualization	Other Visualization Approaches	Interactivity
ProB	CSPM	yes	yes	State Space	no
FDR3	CSPM	yes	yes	Process Communication	yes
PAT	CSP#	yes	yes	LTL Formula Visualization	unknown
TAPAs	CCSP (= CCS + CSP)	yes	no	Process Communication	unknown

2.3. Process Algebras

The second class of formal methods we are considering are called *process algebras*. Process algebras are mainly used for specifying and analyzing concurrent communicating systems. This includes notations such as CSP [Hoa83] with its two major dialects: CSPM [SA11] and CSP# [Sun+09a] and CCSP [Cal+08] (influenced by CCS [Mil89] with some operators from CSP).

Table 2.3 gives a quick overview of the tools and visualization approaches studied in this section. Two visualization approaches are common in the field of process algebras as can be seen in the table: process visualization (studied in Section 2.3.2) and the visualization of counter-examples (studied in Section 2.3.3). Finally, in Section 2.3.4 we study other visualization approaches for process algebras.

2.3.1. Running Example: The Dining Philosophers Problem

In this section, we use the CSPM specification of the dining philosophers problem specification from [Ros10] to support the assessment of the CSP related visualization approaches. The dining philosophers problem was originally invented by E. W. Dijkstra and is a commonly used example in concurrent systems to illustrate the complexity of synchronization issues. The problem statement is described in [Ros10]:

> "Five philosophers share a dining table at which they have allotted seats. In order to eat, a philosopher must pick up the forks on either side of him or her (i.e., both of them) but, there are only five forks. A philosopher who cannot pick up one or other fork has to wait."

2.3.2. Process Visualization

In process algebras the system is described by *processes*. A process is an independent component that can communicate with other processes via message-passing. A common visualization approach in the field of process algebras is their visualization. This is also reflected in Table 2.3: all considered tools provide support for process visualization. Figure 2.12 illustrates the visualization of the *PHIL*(1) process of the dining philosophers problem produced with the FDR3 tool [Gib+14]. As can be seen in the figure, a process is represented as a graph where each state of a process is represented by a circular node and the transitions are represented as edges labeled with the corresponding event name.

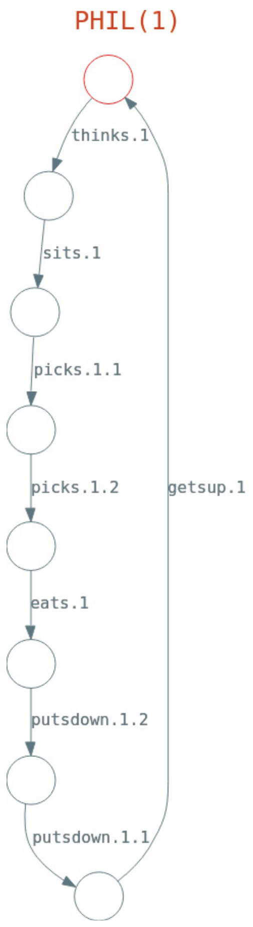


Figure 2.12.: Process visualization in FDR3 for *PHIL*(1) process of dining philosophers problem specification

Similar to the state space visualization approach for state-based formal methods (see Section 2.2.3), the process visualization approach is only suitable for small processes with a small number of states and transitions. Once a process consists of too many states and transitions, human inspection of the visualization may become a difficult task.

2.3.3. Visualizing Counter-Examples

The tools listed in Table 2.3 provide support for various assertion checks, including deadlock freedom, divergence, determinism, and custom LTL formulas [Pnu77]. If an assertion check fails, the tool typically generates a sequence of events (trace) that leads to the state in which assertion failed. This particular trace is also referred to as a *counter-example*. The ProB [LF08], FDR3 [Gib+14] and PAT [Sun+09b] tools are able to visualize counter-examples. Similar to the process visualization feature of the tools, the counter-example visualization is also displayed as a graph. For instance, Fig. 2.13 shows an excerpt of the counter-example visualization for deadlock freeness in the dining philosophers specification checked by ProB. The transitions and states of the counter-example are marked in red and are shown in the context of the the full state space of the specification.

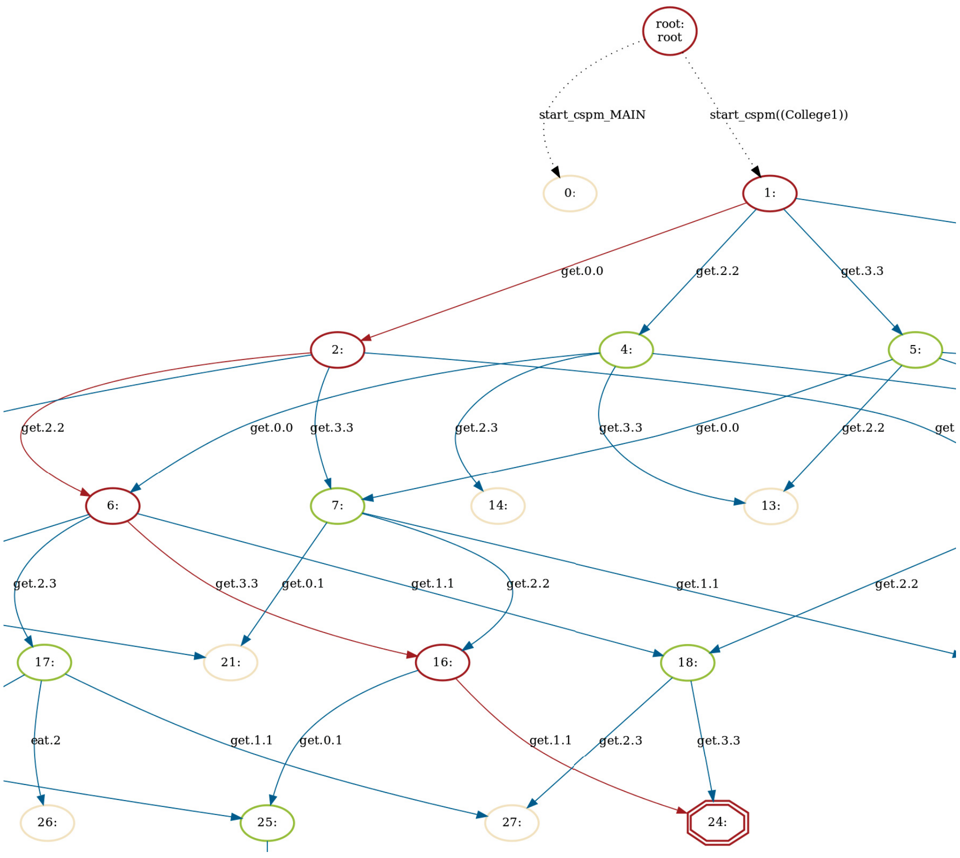


Figure 2.13.: Counter-Example visualization in ProB for deadlock freeness in dinning philosophers problem specification (excerpt)

Table 2.4.: Overview of visualization approaches for other formal methods

Approach / Tool	**Supported Formalisms**	**Visualization Approaches**	**Interactivity**
Uppaal	Timed Automata	Graphical Notation	unknown
Petri net	Petri net	Graphical Notation	unknown
PVSio-web	PVS	Prototype-Builder, State Space	yes

2.3.4. Other Visualization Approaches

The FDR3 [Gib+14] and TAPAs [Cal+08] tools also provide an approach to visualize the communicating processes of a specification. For instance, consider the two process communication visualizations for the dining philosophers problem specification shown in Fig. 2.14. The graphical representation in both tools is different: While FDR3 uses a tree layout (see left side of Fig. 2.14), TAPAs represents the communicating processes with a box containing the particular processes (see right side of Fig. 2.14).

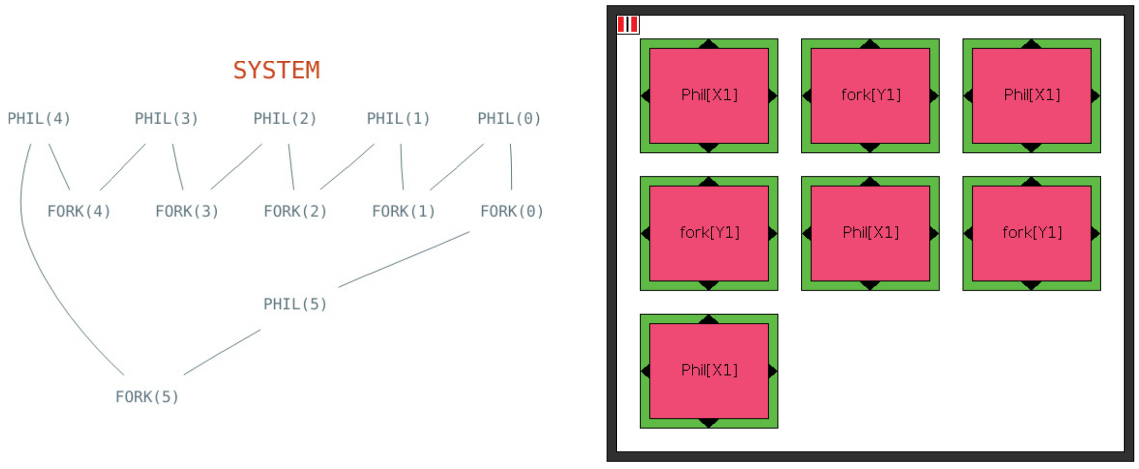


Figure 2.14.: Process communication visualization in FDR3 (left side) and TAPAs (right side) for the dining philosophers problem

2.4. Other Formal Methods

This section deals with tools and visualization approaches for other formal methods apart from state-based formal methods and process algebras which we have studied in our state-of-the-art research. As shown in Table 2.4, this includes formal methods for real time systems (Uppaal [LPY97]), distributed systems (Petri Net [Pet81]), and variants of the $\lambda-calculus$ (PVS [ORS92]).

The tools Uppaal and the tools that support Petri nets are innately linked with graphical notations. Uppaal is a tool that includes a modelling and simulation environment for the specification and analysis of real-time systems based on timed automata [AD94]. A Petri net is a graphical notation for the specification and analysis of distributed systems. In particular, a Petri net is a graph which is made up of places, transitions and tokens. Figure 2.15 shows an example Petri net specification created with [DKS09]. There are many other tools that provide support for Petri nets.[9]

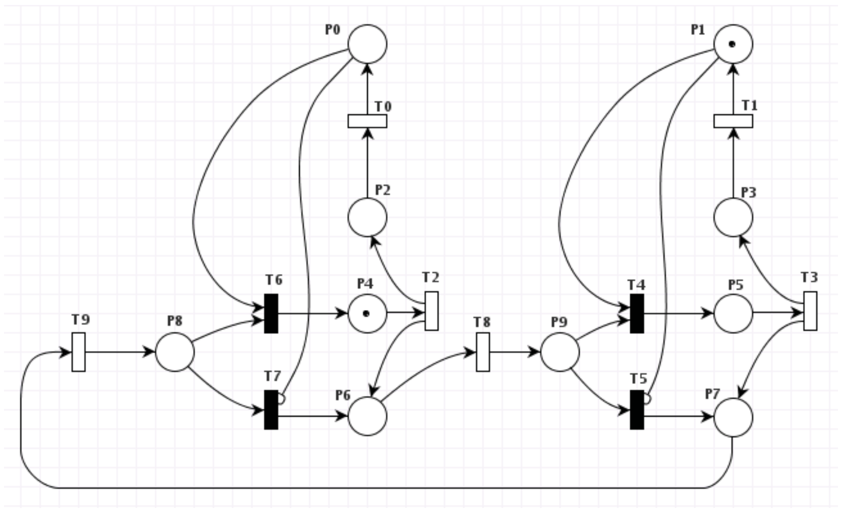


Figure 2.15.: Petri net graphical notation created with [DKS09]

Graphical notations may aid humans in understanding the specification. However, they still need a certain level of knowledge about the underlying mathematical notation. For instance, in Uppaal the user needs knowledge about the theory of timed automata.

PVSio-web [Ola+13; Mas+15b] is a web environment including the animation engine PVSio [Mun05] and a visual editor for creating interactive prototypes for the PVS formalism [ORS92]. The visual editor allows users to choose an image that represents the layout of the system user interface and to set up button and display areas. A button area is an interactive component which is wired to a specific function of the PVS specification and a display area can show a state value of the animated PVS specification. The concept and motivation is similar to that of the graphical visualization approaches for state based formal methods concerned in Section 2.2.2: the animation engine executes the specification and provides the necessary information to be visualized (e.g. the state values for the display areas). The prototype can be used for discussing the specification with non-formal method experts (e.g. domain experts) and to walk though scenarios by interacting with the prototype. Figure 2.16 shows the graphical editor of PVSio-web for the example prototype of the data entry system of an infusion pump user interface taken from [Ola+13]. PVSio-web was successfully used for supporting the validation of the user interfaces for various medical devices [Mas+15a; Mas+14; Ola+13].

[9]http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html.

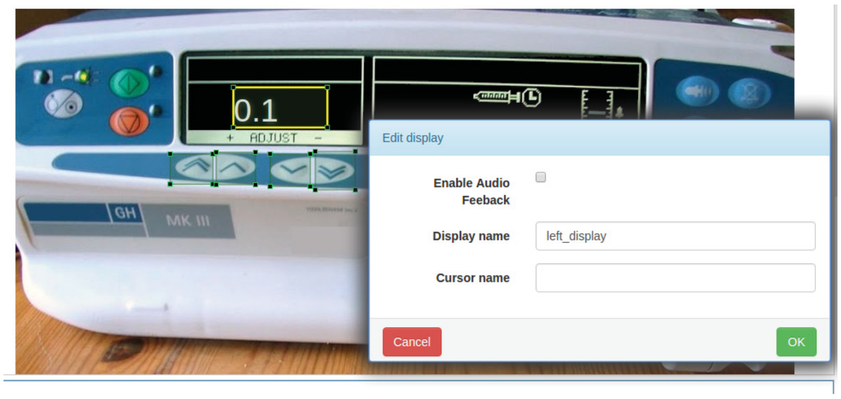


Figure 2.16.: The visual editor of PVSio-web taken from [Ola+13]

2.5. Summary and Final Assessment

In this chapter we have covered animation-based visualization approaches for formal methods with respect to the research problem stated in Section 1.2 and based on a set of criteria introduced in Section 2.1. The chapter discusses different classes of formal methods: state-based formal methods, process algebras and other formal methods such as for the specification of real time systems and distributed systems, as well as variants of the $\lambda - calculus$.

For state-based formal methods, we have studied several graphical visualization approaches (see Section 2.2.2). Graphical visualization in general is a promising approach for involving non-formal method experts, such as domain experts, in order to validate formal specifications. This is because they make it possible to create visualizations that are inspired by the application domain of the formal specification. However, they differ in their implementation and usability. We have identified two classes of graphical visualization approaches for creating the graphical representation and for mapping a state to its graphical representation (gluing code):

- The first class considers approaches that require additional knowledge for creating the graphical representation and the gluing code (e.g. using techniques like HTML5 or programming languages like Java, JavaScript, or ActionScript); however, these enable the developer to create rich interactive visualizations. For instance, in the Overture built-in approach for VDM [NLL12] the developer can make use of different Java GUI toolkits and techniques like the MVC pattern in order to create a graphical visualization.
- The second class considers approaches that use the mathematical notation of the respective formal method to create the gluing code. For instance, the ProB Tcl/Tk-based approach requires an *animation function* written in classical-B and in BMotionStudio the user set up observers by writing Event-B expressions and predicates. The graphical representation is realized using simple images (ProB Tcl/Tk) or by

means of a visual editor with predefined graphical elements like shapes, labels and images (BMotionStudio). Although these approaches provide a convenient and fast approach for creating simple graphical visualizations, they make it difficult to use and apply them when creating complex visualizations with many repeated elements.

The graphical visualization approaches make use of different techniques for developing a visualization. We particularly highlight the approaches based on web-technologies, since the final visualization can be deployed on a web-server. This makes the visualization accessible from other devices such as mobile phones and tablets. Furthermore, the visualization can be shared with other stakeholders (e.g. during an online project meeting).

Some of the considered animation tools provide generic GUI frontends that display information (typically in text form) about the current state of the animation (e.g. variable values and invariants). These frontends may become inaccessible to non-formal method experts since they require knowledge of the mathematical notation of the respective formal method to understand the meaning of a state. We believe that a customization of the GUI frontends according to the application domain may reduce the barriers preventing non-formal method experts from using such GUI frontends.

For process algebras, we have identified two common visualizations approaches which may support the user in analyzing CSP specifications: the visualization of processes and counter-examples. To our surprise, there are no graphical visualization approaches for process algebras.

Some of the visualization approaches, such as the different graph visualization approaches for state-based formal methods (see Section 2.2.3) and the process algebra related visualization approaches (see Section 2.3) have the advantage that they are mainly generated automatically with the respective approach. We believe that combining them with graphical visualization approaches could reduce the barriers preventing non-formal method experts from using such visualizations.

Finally, interactivity is one of the most important criteria to increase the usability of visualization approaches that we have identified. Unfortunately, only a few graphical visualization approaches provide interactive features.

3 BMotionWeb: Concept

3.1. Introduction

The main goal of this thesis is to develop an approach (concept and tool) to complement the use of animation with interactive data visualization and thereby support the validation of formal specifications. To reach this goal, this thesis presents a novel graphical environment called BMotionWeb. BMotionWeb builds on the ideas of BMotionStudio (described in Section 2.2.2) to provide support for the rapid creation of *interactive formal prototypes*. Note that BMotionWeb is the successor and not a replacement for BMotionStudio. In the course of developing BMotionWeb, we take account of the results obtained from the state-of-the-art research study in Chapter 2. In this chapter we describe the concept of BMotionWeb for *state-based formal methods*[1] and define the goals for developing the tool (Section 3.3).

3.2. Interactive Formal Prototyping

The concept of *prototyping* is central for this thesis. Prototyping is a common approach for supporting the software validation process. Its purpose is to create a prototype, i.e. a first (incomplete) version of a software to gain feedback (e.g. from end-users or customers) in the early development phase. A classical approach for developing a software prototype is to use general-purpose programming languages like Java or C. Traditional prototypes are often equipped with graphical user interfaces or represented as extensive 2D and 3D visualizations or even as physical models. However, a factor that must not be underestimated in prototyping is the creation and maintenance of a prototype during the development process. The expenditure of time and overhead can vary depending on the size of the project [Som06].

Using an animation tool for executing a formal specification is often compared with prototyping. Although far more abstract than a Java or C prototype, the executable specification allows users to explore the behavior of the system early in the development process and may thus provide the same useful feedback as traditional prototypes.

[1]In Chapter 6, we will also adapt BMotionWeb to event-based formal methods.

However, no additional programming code needs to be created and maintained. In this thesis, we extend the concept of animation to the use of data visualization and interactive techniques to provide sophisticated prototyping features as known from traditional prototypes. The result is an *interactive formal prototype* that binds the intended functionality of the system to an interactive visualization. An interactive formal prototype consists of the following components: the animated formal specification, a visualization, interactive handlers, and the mapping between the animated formal specification and the visualization (gluing code). In the following sections we describe each component in more detail.

3.2.1. Animated Formal Specification

The animated formal specification (also known as the executed formal specification) is the basic part of an interactive formal prototype. It explores the reachable states of the specification by evaluating transitions and exposes the information encoded in the states (e.g. the concrete values of state variables) to the user. In order to animate a formal specification and depending on the used animation engine, the user must either provide initial values to the animator at startup or the values are actually provided by the given animation engine (e.g. values of the constants of a classical-B or Event-B specification). Once, a formal specification is animated, the user can analyze the formal specification by walking through possible functional behavioral scenarios based on the provided initial values. Different animation tools exists for classical-B and Event-B, such as ProB [HLP13; LB08], Brama [Ser06], AnimB [Mét] and JeB [Yan13].

3.2.2. Visualization

As stated in Section 1.3, the main goal of this thesis is to support the use of animation techniques and to make animation techniques more accessible to non-formal method experts by means of *interactive data visualization*. Hence, visualization is also an important component for an interactive formal prototype. The basic idea is to provide a visual representation for the animation of a formal specification. To do this, we first need to identify the data in the animated formal specification which is to be visualized. Regarding state-based formal methods, this can be the data encoded in the states of a formal specification, such as the state values of variables, results of expression evaluations or transition data. As a next step, we need to find an adequate visual representation based on the identified data. Here we also need to take the users into account who should benefit from the visual representation. These can be users such as formal engineers, end-users, and since a formal specification is typically associated with a domain (e.g. railway, automotive or aerospace), also domain experts. A visual representation that may benefit these users and that is used in this thesis is a *domain specific visualization.*

Domain Specific Visualization

In order to give a precise definition of what we mean by "domain specific visualization", we will start by defining each word in the term. When we talk about a *domain* in the field of software engineering, we mean applications that share a common set of terminology and requirements. For instance, a cruise-control and a lane-departure warning system have their origin in the automotive domain. Another example for an application domain is the railway domain with applications like interlocking systems and autonomous systems such as the COPP system [Enga].

Based on the definition of a domain and in matters of data visualization, we can now define the term "domain specific visualization". A domain specific visualization is the visual representation of a *specific* domain. When considering a domain specific visualization of an interlocking system, we could create a visual representation of the switches, signals and trains using appropriate pictures.

Finally, if we are talking about domain specific visualizations with respect to animated formal specifications, we mean that the data to be visualized is coming from the (animated) formal specification. Considering the previously mentioned example of an interlocking system, the data to be visualized could be the state value of a switch (e.g. left or right), a signal (e.g. stop or go), or the position of a train. The graphical visualization approach for state-based formal methods presented in the state-of-the-art research (Section 2.2.2) also use domain specific visualizations as visual representations.

Graphical Elements

A visualization of an interactive formal prototype is composed of several *graphical elements*. Figure 3.1 illustrates the concept of a graphical element. A graphical element could be a shape, an image, or a label. Each graphical element has attributes that define its appearance and layout. This includes common attributes like the coordinates of the graphical element defining its position in the visualization or individual attributes like the fill color of a shape. In our example of the domain specific visualization of the interlocking system, we can use graphical elements to emulate the different aspects of the interlocking domain. For instance, we could use a green circle to denote that a train can pass and a red circle to denote that the train must stop.

3.2.3. Observers

The gluing code for an interactive formal prototype defines the mapping between the animated formal specification and its visual representation. A naive approach for creating the gluing code is certainly to map each individual state of an animated formal specification to an appropriate visualization. However, a formal specification typically contains a lot of different states so that simply mapping each state to a visualization is

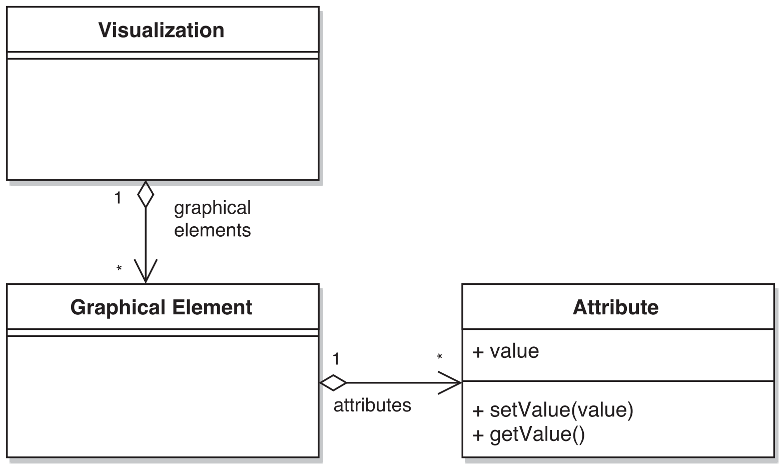


Figure 3.1.: A visualization in BMotionWeb is composed of graphical elements

an almost impossible task. To overcome this challenge, we introduce a new approach for creating the gluing code based on decomposition into *observers*.

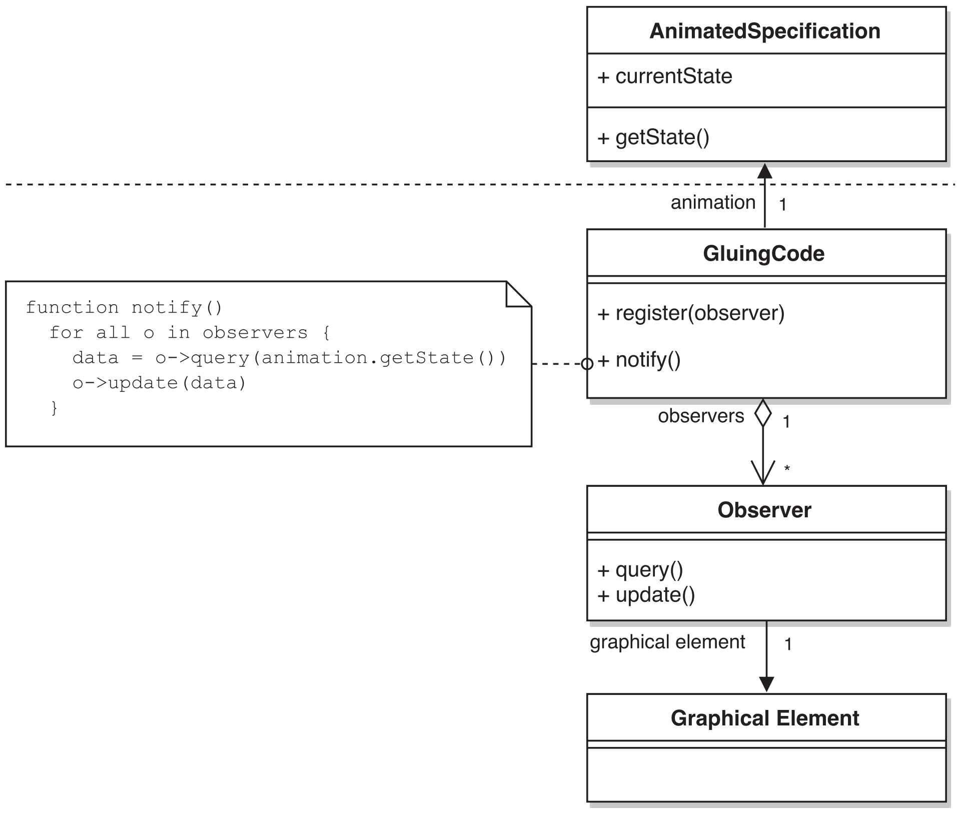
Algorithm 1: Compute the representation of a graphical element for a given state

```
1 function computeStateBased(state $s$, graphicalElement $elem$)
2     foreach $o \in collectObservers(elem)$ do
3         $o.update(o.query(s))$
4     end foreach
5     return $elem$
6 end function
```

Figure 3.2 shows the architecture of the observer-based gluing code approach, and Fig. 3.3 illustrates the approach during an animation. The basic idea for an observer is to *observe* specific data within the animated formal specification (e.g. a specific variable or an expression) and to determine the appearance of a linked graphical element according to the observed data for a given state (e.g. the value of a variable or the evaluation of an expression). For this, an observer can be registered in the *gluing code* of the interactive formal prototype and is notified whenever a state change has occurred. Once an observer is notified, it can *query* the current state of the animated formal specification and *update* the linked graphical element by changing its attributes according to the observed data in the current state.

Formally, one can describe the approach using Algorithm 1. The algorithm computes the representation of a graphical element *elem* for a specific state s. For each observer o of the observers which are linked to the graphical element *elem* (line 2), we query the data needed to update the representation of *elem* in state s (line 3). Finally, the updated graphical element *ele* is returned.

The observer architecture is inspired by the *Model-View-Controller pattern* (MVC) and



```
function notify()
  for all o in observers {
    data = o->query(animation.getState())
    o->update(data)
  }
```

Figure 3.2.: Architecture of the observer-based gluing code approach

the *observer pattern* described in [Gam95]. This has several advantages:

- The animated formal specification and the visualization are independent components of the interactive formal prototype. This makes the architecture generic so that different animation engines and visualization techniques can be used.
- The animation engine does not need to know about the observers and what state data is required for them. On the other hand, observers are free to determine what they want to query of the state without relying on the animation engine to send the correct information. State data that is not required is not sent to the observers (see the pull model of the observer pattern in [Gam95]).
- By having multiple observers observing different aspects of the state, we can decompose the gluing code into smaller parts rather than having a large single gluing code (e.g. see the gluing code for the Overture VDM-SL and the JeB Event-B graphical visualization presented in Section 2.2.2).
- The visualization and the gluing code can be created and maintained independently. For instance, a domain expert can create the visualization without having knowledge of the actual formalism, and the formal engineer can define the observers.

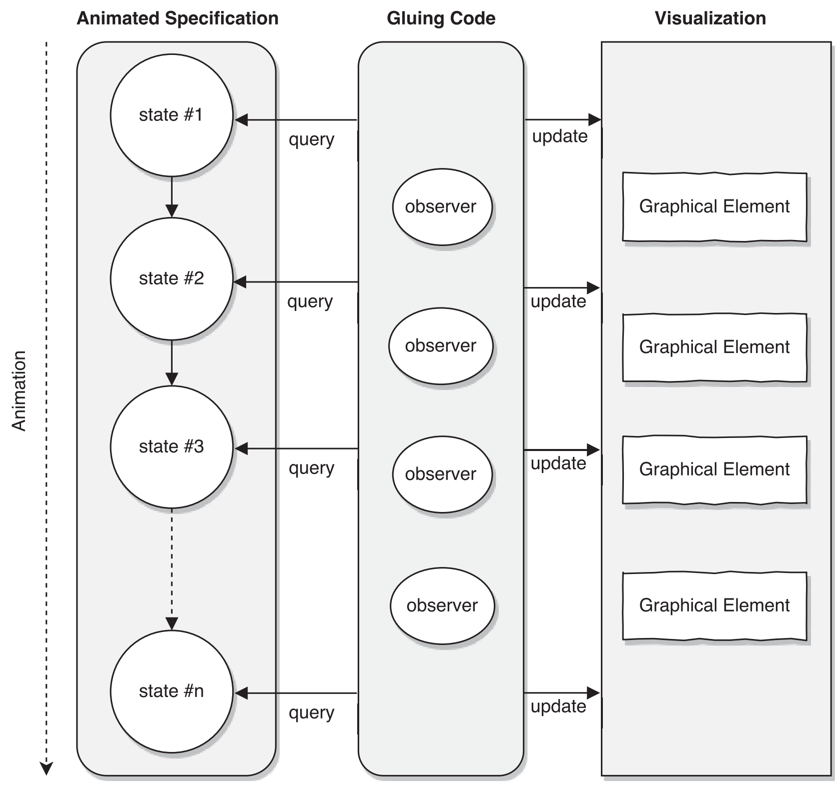


Figure 3.3.: Observer-based gluing code

- The visualization and the gluing code can be reused for creating interactive formal prototypes of other formal specifications.
- The architecture is scalable and facilitates the incremental development of the interactive formal prototype: when new elements are added to the formal specification (e.g. variables), the interactive formal prototype can be extended by adding new graphical elements and observers.

The observer architecture also implies some requirements:

1. The formal specification must be executable by an animation engine.
2. The animation engine must provide an interface for listening to state changes and appropriate methods for querying the observed state data.
3. The visualization technique used must provide visualizations which are composed of individual graphical elements (e.g. shapes and images) with attributes that define their style and layout (e.g. color or position).

3.2.4. Interactive Handlers

In the state-of-the-art study (see Chapter 2), we found that the *interaction* between a human and the computer is an important aspect of data visualization. So far, however, we have only defined the gluing code of an interactive formal prototype that maps a state to its visual representation, where the visual representation is adapted whenever a state change occurred in the formal specification. In state-based formal methods, a state change is triggered by the execution of a transition. Most animation tools provide GUIs for executing transitions. As an example, see the ProB based approaches and the JeB approach described in Section 2.2.2. However, these GUIs are typically generic and assume that the user is familiar with the animated formal specification (e.g. the user knows the meaning and function of a transition). To overcome this challenge, we present an approach for wiring actions to graphical elements, for instance by executing transitions based on point and click interactions [Dix+03] with the graphical elements. This makes the visualization interactive and gives a transition a visual meaning.

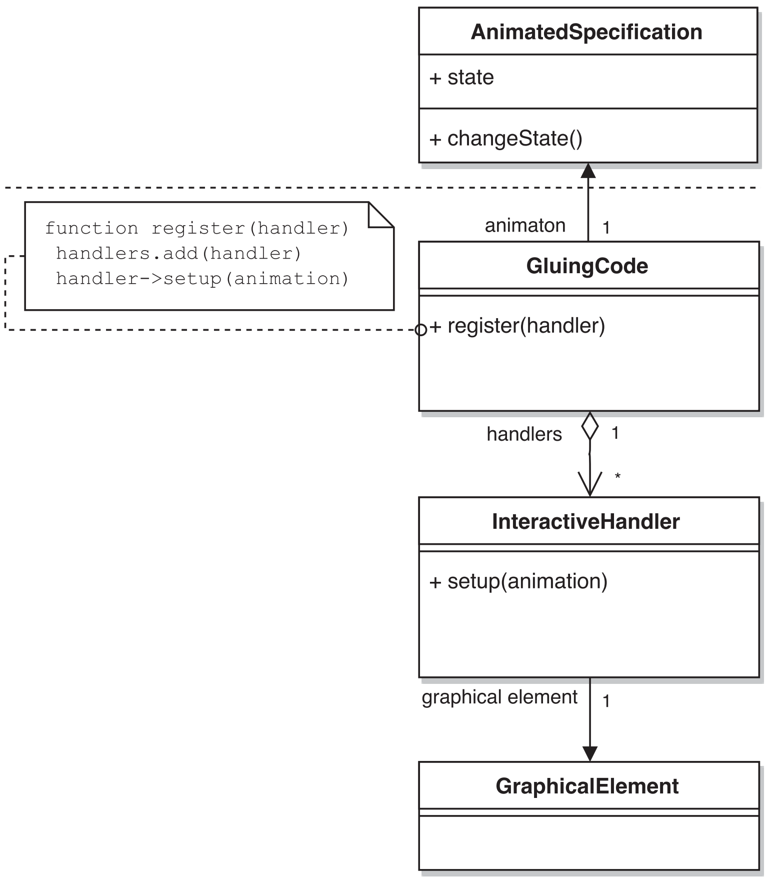


Figure 3.4.: Architecture of interactive handlers

Figure 3.4 shows the architecture for wiring an interactive action to a graphical element. We extend the gluing code to include the concept of interactive handlers in addition to observers (see Section 3.2.3). When an interactive handler is registered, the *setup* function of the interactive handler is called with a reference to the animated formal specification. As an example, the setup function may register a click handler on the graphical element that executes a transition when the user clicks on the element.[2]

Similar to the observer architecture introduced in Section 3.2.3, the interactive handler architecture provides several advantages: since an interactive handler is registered in the gluing code, it becomes an independent component of the interactive formal prototype. The reference to the animation engine is realized indirectly via the setup function of the interactive handler (see Fig. 3.4). The architecture is also scalable: when new transitions are added to the formal specification, the interactive formal prototype can be extended by adding new interactive handlers which execute the new transitions.

3.3. Goals for BMotionWeb

Based on the concept of interactive formal prototyping, we have defined the following goals for implementing BMotionWeb:

(1) Develop a tool that supports the creation and execution of interactive formal prototypes.

(2) The tool should provide a graphical environment that enables the user to rapidly create the visualization and the gluing code (observers and interactive handlers).

(3) When dealing with complex interactive formal prototypes (e.g. with numerous or repeated graphical elements), the tool should provide a scripting language.

(4) The tool should be generic: existing animation engines must be adaptable.

[2]The provided interactive features (e.g. click handler) depends on the used visualization technique.

4 BMotionWeb: Implementation

4.1. Introduction

In this chapter, we present the implementation of BMotionWeb and discuss our design decisions and implementation choices. To illustrate this, we have integrated the ProB animator [LB08] with BMotionWeb. The reason for choosing ProB was that it provides already an elaborated API [STU] for executing the state-based formal methods classical-B and Event-B. To exemplify the functionality of the implementation, we use the Event-B specification of the simple lift system introduced in Section 2.2.1.

This chapter is organized as follows: first, we give some background information which is relevant for the implementation of BMotionWeb in Section 4.2. Section 4.3 gives an overview of the architecture of BMotionWeb and discusses our design decisions and implementation choices. In the subsequent sections (Section 4.4 to Section 4.10), we describe the different components of the architecture in more detail. In Section 4.8 we present our implementation of the observers and interactive handlers concept introduced in Chapter 3. Section 4.9 demonstrates how the developer of an interactive formal prototype can programatically control the integrated animation engine and Section 4.10 presents the visual editor of BMotionWeb. Finally, in Section 4.11, we describe the versions of BMotionWeb which are currently available.

4.2. Background

This section gives background information about ProB (Section 4.2.2) and the supported state-based formal methods classical-B and Event-B (Section 4.2.1).

4.2.1. The B-Method

The B-Method [Sch01], originally developed by Jean-Raymond Abrial, is a formal approach to the specification and development of *reliable* safety-critical systems. There are two specification languages associated with the B-method: classical-B [Sch01; Abr96] and its successor Event-B [Abr10]. While classical-B aims at specifying and analyzing software systems, Event-B is typically applied in the field of reactive systems. They are based on the Abstract Machine Notation (AMN). AMN consists of three levels: (1) the initial specification of the system, where a first abstract specification of the system is developed; (2) the stepwise refinement of the abstract specification, where the abstract data is refined by more concrete data; and (3) the implementation, where the last refinement step is translated manually or via code generation into a suitable programming language (i.e. C or ADA).

A notation based on set theory and first order logic is used to express the components which make up a specification:

- *Variables, constants and sets*: Variables maintain state information, constants are abstract constant values, such as an interval or a natural number, and sets are collections of entities (either enumerated or deferred).
- *Invariants and properties*: These define the types of variables and constants respectively. Moreover, the invariants define conditions about the reachable states of the specification. Every state must satisfy these conditions otherwise the specification is not considered consistent.
- *Initialization*: Defines the possible initial states of the specification. An initial value must be assigned to each variable.
- *Operations*: Define the behavior of the system (i.e. they determine the transitions in the specification). An operation is characterized by its name, input parameters, output parameters (in classical-B), a condition under which the operation can be executed, and the effect of the operation (e.g. which state variables are changed by the execution of the operation).

The specification language uses three syntactic categories:

- *Expressions*: Expressions are formulas combining constants, variables, operators and functions which are evaluated in a specific state of the specification. Examples for expressions are the simple arithmetic addition $2 + 1$ or the intersection $A \cap B$.
- *Predicates*: Predicates are formulas which evaluate to true or false for a given state in a formal specification. For example, the predicate $3 = 5$ is false and the construct $2 \in \{1, 2\}$ is true.
- *Substitutions*: Substitutions define the replacement of a term by another term. For example in the assignment $x := 7$, x is replaced by the integer 7. Moreover, expressions can be used in substitutions (e.g. $x := 3 + 4$).

4.2.2. The ProB Animator

ProB [LB08] is a validation toolset with support for the state-based formal methods classical-B [LB08], Event-B [HLP13] and Z [PL07], and other formal methods like TLA$^+$

[HL12] and CSPM [LF08]. The main features of ProB are *model checking* and *animation*. The tool provides various model checking capabilities [PL10; LB03; Leu+01; Ben15] to uncover errors, counter examples, and deadlocks in a formal specification. On the other hand, the animator allows the user to check the presence of desired functionality and to inspect the dynamic behavior of a formal specification by *executing* it. Thus, the user can inspect a specific state of the formal specification in more detail, e.g. the user can obtain the particular values of the state variables, constants and the outgoing events (Event-B) or operations (classical-B) for the respective state. This thesis uses the animation capabilities of ProB as a reference animation engine to implement BMotionWeb.

The core of ProB is written in SICStus Prolog.[1] Several tools and graphical frontends built on top of the ProB core are available. For instance, it exists an Eclipse based graphical frontend integrated into the Rodin platform (see Fig. 4.1). This frontend contain various views for supporting the user while animating a specification:

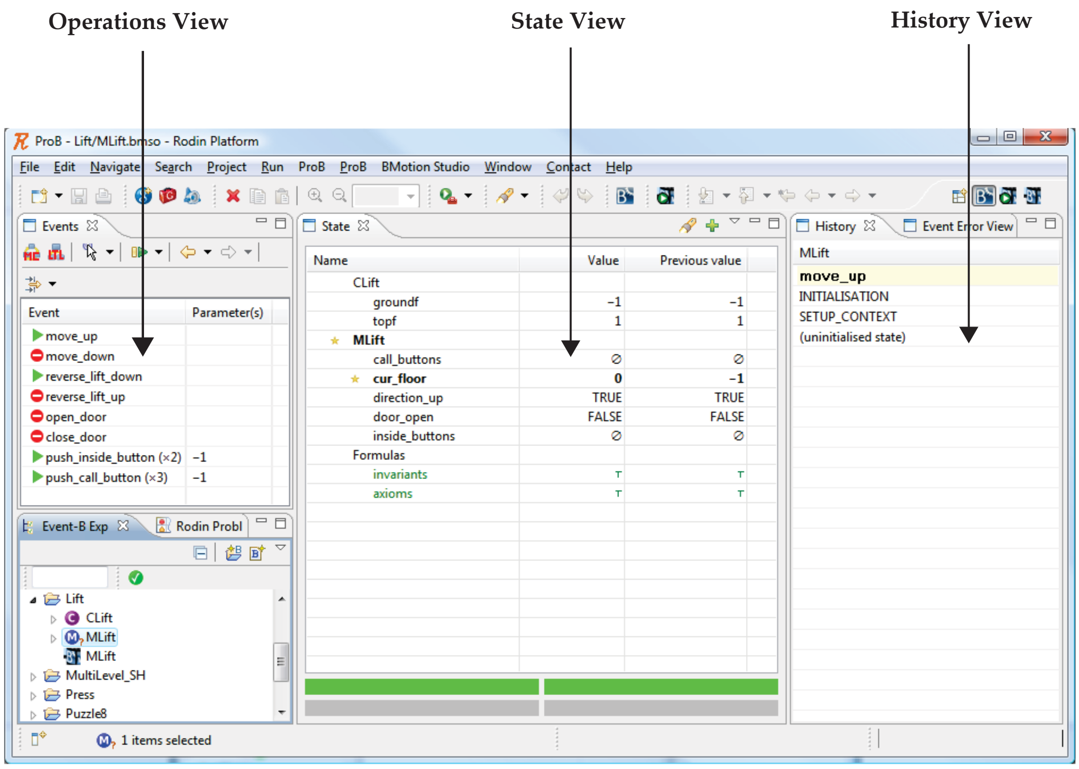
- *State View:* The state view shows the values of variables and constants for the current and previous state of the animated specification. In addition, custom formulas (e.g. expressions or predicates) can be registered which are evaluated in each state.
- *Events View:* The transition view shows a list of all transitions including parameters that are declared in the specification. It differentiates between transitions which are applied in the current state (enabled transitions) and those which are not (disabled transitions). The user can click on an enabled transition causing a state change in the animated formal specification.
- *History View:* The history view shows a list of executed transitions and enables the user to jump back to a previous state.

The ProB 2.0 project [STU] is another tool based on the ProB core. It provides an API for the programmatic control of the ProB animation capabilities written in Groovy, a dynamically typed JVM language [Koe+07]. This includes functions to execute transitions and to evaluate formulas (e.g. expressions or predicates) in a specific state of the animation.

4.3. Architecture of BMotionWeb

The use of web-technologies for developing software has become increasingly popular with the rise of HTML5 [W3C14] and JavaScript frameworks like Google's AngularJS[2] and Facebook's React.[3] Moreover, web-technologies like SVG [W3C11] and data visualization tools like D3 [BOH11] enables the developer to create sophisticated and modern

[1]https://sicstus.sics.se.
[2]https://angularjs.org.
[3]https://facebook.github.io/react.



Event	Parameter(s)
move_up	
move_down	
reverse_lift_down	
reverse_lift_up	
open_door	
close_door	
push_inside_button (×2)	−1
push_call_button (×3)	−1

Name	Value	Previous value
CLift		
groundf	−1	−1
topf	1	1
MLift		
call_buttons	∅	∅
cur_floor	0	−1
direction_up	TRUE	TRUE
door_open	FALSE	FALSE
inside_buttons	∅	∅
Formulas		
invariants	⊤	⊤
axioms	⊤	⊤

MLift
move_up
INITIALISATION
SETUP_CONTEXT
(uninitialised state)

Figure 4.1.: ProB integrated into the Rodin platform

visualizations. This was also confirmed by the state-of-the-art study (see Chapter 2): an increasing number of tools that support formal methods are based on web-technologies. This fact and the large number of libraries available for creating visualizations motivated us to use web-technologies for the development of BMotionWeb.

Web-technologies like JavaScript, HTML5 and SVG are usually supported by runtime environments on the client side (e.g. web-browsers). To be more flexible and to realize the goal of supporting the adoption of BMotionWeb within existing animation tools, we have decided to make also use of a language that can be used on the server side. Here we decided to use the Java language, which is an object oriented language running on the Java virtual machine (JVM). The main reasons for using Java are that it is capable of running compiled code across different platforms and it enables the integration with other languages that are also running on the JVM. For example, the ProB tool introduced in Section 4.2.2 provides a Groovy API for working with different formalisms (e.g. classical-B, Event-B, Z and CSPM). Java provides seamless integration with Groovy applications, since Groovy is a language that also runs on the JVM.

Figure 4.2 gives an overview of the architecture of BMotionWeb. The architecture is divided into a client front-end and a server back-end, where web-sockets [FM11] are used to realize the communication between client and server. The client front-end consists of a *visual editor* and a *simulation engine*. The visual editor allows the user to create and edit a *visualization template*, whereas the simulation engine is responsible for executing

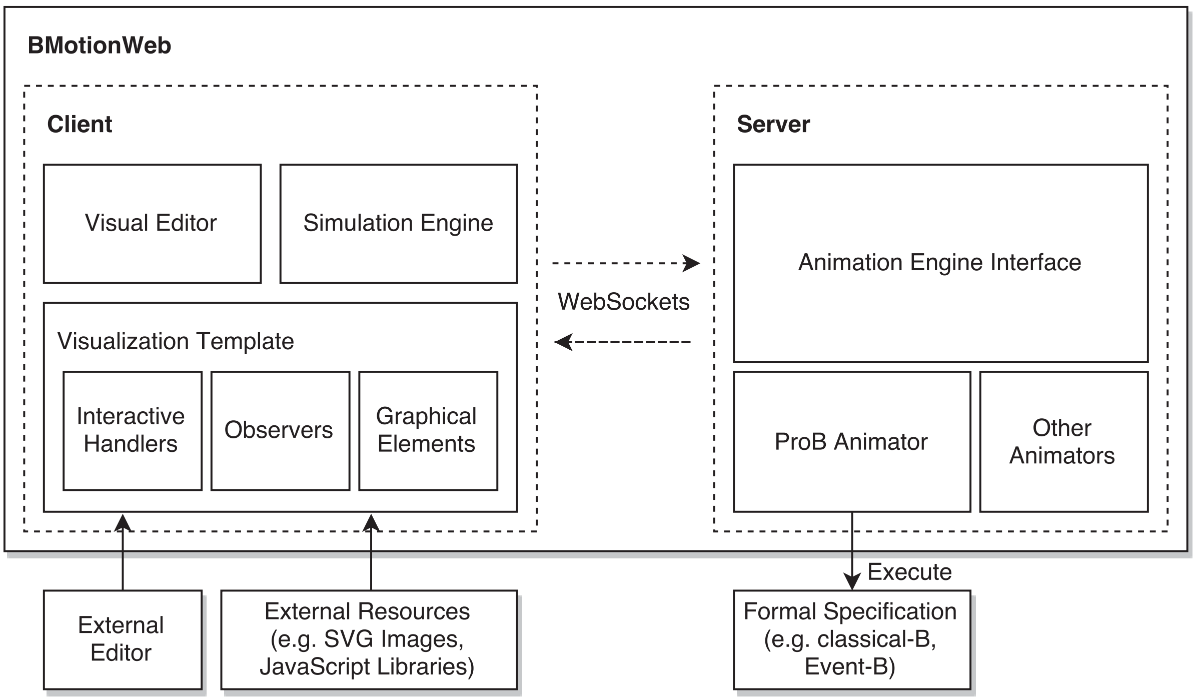


Figure 4.2.: Architecture of BMotionWeb

visualization templates. On the other hand, the server back-end provides an *animation engine interface* capable of integrating external animation engines with BMotionWeb. In other words: the visualization of an interactive formal prototype is developed and executed on the client side and the animated formal specification of an interactive formal prototype "lives" on the server side. In the following sections we describe the components of the client and server in more detail.

4.4. Visualization Template

At the heart of an interactive formal prototype, one finds a *visualization template*. It is the part of the interactive formal prototype that is developed by the user. It describes the visualization and the gluing code (observers and interactive handlers) for the interactive formal prototype. An important design decision was to make the full web-technology stack available to the developer for creating a visualization template. The benefit of this design decision is that the visualization template becomes flexible since external resources, such as SVG images and third party JavaScript libraries, can be reused. Indeed, this can save time for developing an interactive formal prototype and may provide a large selection of reusable SVG images and JavaScript libraries. This design decision can also help the developer to create complex interactive formal prototypes, e.g. with numerous or repeated graphical elements. In general, a visualization template consists of several files which are described in the following subsections.

Table 4.1.: Available options for BMotionWeb manifest file

Name	Type	Required	Description
id	string	yes	Unique id of the interactive formal prototype.
name	string	no	The name of the interactive formal prototype.
template	string	yes	The relative path to the HTML template file (e.g. "template.html").
groovy	string	yes	The relative path to the groovy script file (e.g. "script.groovy").
model	string	yes	The relative path to the formal specification file that should be animated (e.g. "model/mymodel.mch").
modelOptions	map	no	A key/value map defining the options for loading the model - The available options are dependent on the animator and formalism.
autoOpen	array	no	The user can specify the ProB views which should be opened automatically when running the interactive formal prototype - The following views are available for ProB animations (Event-B, classical-B and CSPM): *CurrentTrace*, *Events*, *StateInspector* and *ModelCheckingUI*.
views	list	no	List of additional views - A view object has the following options:
id	string	yes	Unique id of the view.
name	string	no	The name of the view.
template	string	yes	The relative path to the HTML template file of the view (e.g. "view1.html").
width	numeric	no	The width of the view.
height	numeric	no	The height of the view.

4.4.1. Manifest File

A visualization template is identified by a *manifest file*. The manifest file is the root file of every interactive formal prototype. It contains the configuration for the interactive formal prototype in JSON (JavaScript Object Notation) format.[4] Table 4.1 gives an overview of the available options. The table shows the option's name, its type, a short description, and denotes if the option is required or optional. Listing 4.1 exemplifies the use of a manifest file based on the interactive formal prototype of the Event-B simple lift system.

[4]http://www.json.org.

```
 1 {
 2   "id": "lift",
 3   "name": "Simple lift system",
 4   "template": "lift.html",
 5   "groovy": "script.groovy",
 6   "model": "model/m2.bcm",
 7   "autoOpen": [
 8     "CurrentTrace",
 9     "Events"
10   ]
11 }
```

Listing 4.1: Example manifest file for the simple lift system (JSON)

4.4.2. Visualization Files

The HTML template file that is linked in the manifest (see line 4 in Listing 4.1) is the starting point for developing the actual visualization for the interactive formal prototype. The snippet in Listing 4.2 shows an example HTML template file for the simple lift system.

```
 1 <html>
 2 <head>
 3   <title>Simple lift system visualization</title>
 4 </head>
 5 <body>
 6   <script src="bms.api.js"></script>
 7   <script src="lift.js"></script>
 8   <div bms-svg="lift.svg"></div>
 9 </body>
10 </html>
```

Listing 4.2: HTML template file for simple lift system (HTML)

In general, a visualization in BMotionWeb makes use of SVG. For this, BMotionWeb provides a special attribute called *bms-svg* that takes a relative path to an SVG image file as its value (see line 8). The attribute renders the entered SVG image file within the visualization and registers it in the visualization template. A registered SVG image file can be edited by means of the built-in visual editor in BMotionWeb which is described in Section 4.10. Since the SVG image file is an external file it can also be edited with any other SVG editor. As an example, consider Fig. 4.3. The left side of the figure shows the SVG image file for the simple lift system and the right side demonstrates the SVG file rendered in the interactive formal prototype. In addition to SVG, BMotionWeb also makes the full web-technology stack available to the user in order to create a visualization (i.e. the user can apply other web-techniques with HTML5, CSS and JavaScript).

In line 6 we reference the JavaScript file *bms.api.js*. This provides the BMotionWeb JavaScript API with functions, e.g. to register observers and interactive handlers. The developer can make use of this API by referencing an additional JavaScript file which contains custom JavaScript code (see *lift.js* in line 7). The BMotionWeb JavaScript API is described in Section 4.8.

4.4.3. Groovy Script File

The developer can optionally define a Groovy[5] script file (see line 5 in Listing 4.1) to link custom Groovy or Java code to the interactive formal prototype that is evaluated on the server side. Within the Groovy script file the developer can also make use of the BMotionWeb Groovy API with functions, e.g. to control the integrated animation engine or to register external methods that can be triggered from the client side (JavaScript). As an example, using the Groovy API the developer may query an external database or make some complex computations based on the information coming from the animated formal specification (e.g. state information). The Groovy API is described in Section 4.9 in more detail.

4.5. Working with Graphical Elements

The web-technologies used for developing a visualization provide us with several predefined graphical elements and techniques for creating and styling them. For instance, HTML provides elements like tables, buttons and lists, and SVG provides elements like shapes and images. These elements are also referred to *DOM* elements.[6] With CSS we have a comprehensive technique to define the style and layout for HTML and SVG elements. BMotionWeb uses these techniques as the basis for implementing the graphical element concept introduced in Section 3.2.2 and for linking them to observers and interactive handlers.

In order to identify and to manipulate a graphical element within a visualization, BMotionWeb uses jQuery.[7] jQuery is a JavaScript library for *selecting* and *manipulating* DOM elements in an HTML document. It extends the CSS selector syntax[8] to provide selectors based on the id, name, classes, types, attributes and many more properties of a DOM element. As an example, consider the JavaScript snippet in Listing 4.3. It shows two examples for selecting and manipulating graphical elements based on the SVG image shown in Fig. 4.3. With jQuery we can select the graphical element that represents the lift door by its id as demonstrated in line 1 (the prefix "#" is used to match a graphical element by its id). Once an element is selected, jQuery provides us with a reference

[5]http://groovy-lang.org.
[6]http://www.w3schools.com/js/js_htmldom.asp.
[7]https://jquery.com.
[8]https://www.w3.org/TR/css3-selectors.

to the selected element and allows us to manipulate it, e.g. by changing its attributes. For instance, in line 2 we set the *fill* attribute of the door to the color *gray*. We can also select and manipulate multiple graphical elements as demonstrated in lines 4 and 5, where we select all *ellipse* graphical elements which have a *data-floor* attribute (line 4) and color them all green (line 5). For a comprehensive list of jQuery selectors we refer the reader to the jQuery selectors API documentation.[9]

```
1 var door = $("#door");
2 door.attr("fill", "gray");
3
4 var allRequestButtons = $("ellipse[data-floor]");
5 allRequestButtons.attr("fill", "green");
```

Listing 4.3: Example for selecting and manipulating elements using jQuery (JS)

```
<svg width="220" height="340"
  xmlns="http://www.w3.org/2000/svg">
 <g id="lift_system">
  <g id="lift">
   <rect fill="white" stroke="black"
     height="330" width="100" y="5" x="50"/>
   <rect id="door" fill="gray" stroke="black"
     height="80" width="70" y="245" x="65" />
   <text fill="black" y="58" x="165">Floor 1</text>
   <text fill="black" y="182" x="165">Floor 0</text>
   <text fill="black" y="290" x="165">Floor
       -1</text>
  </g>
  <g id="request_buttons">
   <ellipse id="bt_1" data-floor="1"
     ry="11" rx="11" cy="54" cx="22" fill="gray"/>
   <ellipse id="bt_0" data-floor="0"
     ry="11" rx="11" cy="177" cx="22" fill="gray"/>
   <ellipse id="bt_-1" data-floor="-1"
     ry="11" rx="11" cy="285" cx="22" fill="gray"/>
  </g>
 </g>
</svg>
```

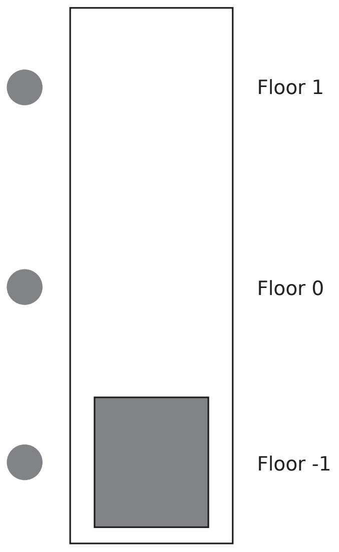


Figure 4.3.: SVG image of simple lift system: source (left) and rendered (right)

[9]http://api.jquery.com/category/selectors.

4.6. Animation Engine Interface

The animation engine interface manages the communication between the client front-end and the integrated animation engine via web-sockets. It exposes functions for controlling the animation engine and for interacting with the animated formal specification to the client front-end. Each integrated animation engine must implement some default functions, such as functions to execute a formal specification, to execute a transition, or to evaluate a formula. In this thesis, we have implemented the animation engine interface for supporting the ProB animation engine (see Section 4.2.2). Moreover, we have started to implement the interface for supporting CoreASM [FGG07], an animation engine for executing ASM specifications.

4.7. Simulation Engine

The *simulation engine* allows users to interact with the interactive formal prototype and to explore its behavior. For this purpose, it renders a visualization template and manages the communication between the client and server. In particular, it sends requests from a registered observer (e.g. which evaluates an expression) to the animation engine via the animation engine interface on the server side and forwards the results from the animation engine back to the observer. It also triggers state changes in the animated formal specification based on the registered interactive handlers.

4.8. Observers and Interactive Handlers

In this section we present our implementation of the observer and interactive handler concept introduced in Section 3.2.3 and Section 3.2.4 respectively. BMotionWeb implements various observers and interactive handlers with different functions. They are implemented in JavaScript and follow the uniform schema shown in Listing 4.4, where *bms* is a global variable pointing to the BMotionWeb JavaScript API, *observe* a function to register an observer (see line 1) and *handler* a function to register an interactive handler (see line 2). The functions have two arguments: the first argument defines the *type* of the observer or interactive handler, and the second argument defines a list of *options* that are passed to the respective function. The options are defined as a *key/value* map, where *key* is the option's name, and *value* is the option's value. The options may be of different types (e.g. string, integer, boolean, or a function). In order to link graphical elements to observers and interactive handlers, each observer and interactive handler can define the *selector* option. The selector option determines the graphical elements to which the observer or interactive handler will be attached using the jQuery selector syntax (see Section 4.5).

Table 4.2.: Available options for formula observer

Name	**Type**	**Required**	**Description**
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
formulas	list	yes	A list of *formulas* (e.g. expressions, predicates or single variables) which should be evaluated in each state. For instance, $['x','card(x)']$ observes the variable x and the expression $card(x)$ (the cardinality of the variable x).
translate	boolean	no	In general the result of the formulas will be strings. This option should be set to *true* to *translate* B-structures to JavaScript objects.
trigger	function	yes	The *trigger* function will be called after every state change with its *origin* reference set to the graphical element that the observer is linked to and with the *values* of the formulas at the new state. The *values* parameter is an array containing the values of the formulas, e.g. use *values[0]* to obtain the result of the first formula. If no selector is defined, the *trigger* function is called only with the *values* parameter.

```
bms.observe(<type>, <options>);
bms.handler(<type>, <options>);
```

Listing 4.4: Implementation schema for observers and interactive handlers (JS)

In the following subsections we present the various observer and interactive handler types. In each section, we first give a brief description of the characteristics of the respective observer or interactive handler and list their available options in a table. The table defines the option's name, its type, a short description and denotes if the option is required or optional. To illustrate the behavior of an observer or interactive handler, we apply it to the simple lift system presented in Fig. 4.3.

4.8.1. Formula Observer

The formula observer watches a list of formulas (e.g. expressions, predicates or single variables) and triggers a function whenever a state change occurred in the animated formal specification. The values of the formulas and the origin (the reference to the graphical element that the observer is attached to) are passed to the trigger function. Within the trigger function, the user can manipulate the origin (e.g. change its attributes) based on the values of the formulas in the respective state. Table 4.2 gives an overview of the available options for the formula observer.

```
 1 bms.observe("formula", {
 2   selector: "#door",
 3   formulas: ["floor"],
 4   translate: true,
 5   trigger: function (origin, values) {
 6     switch (values[0]) {
 7       case 1: origin.attr("y", "20"); break
 8       case 0: origin.attr("y", "140"); break
 9       case -1: origin.attr("y", "250"); break
10     }
11   }
12 });
```

Listing 4.5: Example formula observer (JS)

Listing 4.5 shows how the formula observer is used in the simple lift system. In line 1 we register a new formula observer to the graphical element that matches the selector "#door", i.e. the graphical element that represents the door of the simple lift system (line 2). Line 3 states that the observer should observe the variable *floor* during the animation. In line 4 we set the translate option to true. By default the results of evaluating the formulas are strings. Setting the translation option to true translates the string results into JavaScript objects. Table 4.3 gives an overview of the mapping between B (classical-B and Event-B) constructs represented as strings and JavaScript objects. For instance, the value "TRUE" is translated into the JavaScript object *true* which can be then used in the JavaScript context (e.g. in a conditional statement). In lines 5 to 11 we define a trigger function that is called whenever a state change has occurred. The reference to the matched graphical element (*origin*) and the state values of the observed formulas (*values*) are passed as arguments to the trigger function. The trigger function in Listing 4.5 defines the position of the lift cabin (see lines 6 to 10). For this, it maps the *y* coordinate attribute of the origin to the desired value based on the state value of the *floor* variable (*values[0]*).

4.8.2. Predicate Observer

The predicate observer observes a predicate and triggers a function depending on the evaluation of the predicate in the respective state (true or false). The reference to the graphical element to which the observer is attached is passed to the particular function. Table 4.4 gives an overview of the available options for the predicate observer.

As an example, Listing 4.6 shows a predicate observer for the simple lift system. The purpose of the observer is to set the *fill* attribute of the door to the color *white* (denoting that the door is opened) or to *gray* (denoting that the door is closed) based on to the evaluation of the predicate in the respective state (true or false). To do this, we register a new predicate observer (line 1) for the graphical element that matches the selector "#door" (line 2). In line 3 we define the predicate $door = open$ that should be observed

Table 4.3.: Overview of translating B constructs to JavaScript objects

		Example	
B Construct	**JavaScript**	**B as String**	**JavaScript**
BOOL	Boolean	“TRUE”	true
Naturals	Number	“2”	2
Integers	Number	“-2”	-2
Sets	Array	“{2, 3}”	[2, 3]
Sets of Sets	Array	“{{2}, {2, 3}, {2, 3, 4}}”	[[2], [2, 3], [2, 3, 4]]
Relations	Array	“{(2, 3), (3, 4)}”	[[2, 3], [3, 4]]
Nested Relations	Array	“{({(2, 3)}, 3)}”	[[[[0, 0]], 0]]
Functions	Array	“{(2, 3), (3, 4)}”	[[2, 3], [3, 4]]

Table 4.4.: Available options for predicate observer

Name	**Type**	**Required**	**Description**
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
predicate	string	yes	A *predicate* which should be evaluated in each state.
true	function	yes	The *true* function will be called whenever the predicate evaluates to true in the respective state with its *origin* reference set to the graphical element that the observer is linked to. If no selector is defined, the *true* function is called without parameters.
false	function	yes	The *false* function will be called whenever the predicate evaluates to true in the respective state with its *origin* reference set to the graphical element that the observer is linked to. If no selector is defined, the *false* function is called without parameters.

during the animation. Lines 4 to 6 define the function that is called whenever the predicate is true in the respective state. When this is not the case the false function is called (see lines 7 to 9).

```
 1 bms.observe("predicate", {
 2   selector: "#door",
 3   predicate: "door = open",
 4   true: function(origin) {
 5     origin.attr("fill", "white");
 6   },
 7   false: function(origin) {
 8     origin.attr("fill", "gray");
 9   }
10 });
```

Listing 4.6: Example predicate observer (JS)

4.8.3. Set Observer

The state-based formal methods classical-B and Event-B are based on set theory. Thus, the different aspects of the system are often expressed as sets. As an example, consider a formal specification of an interlocking system, where the occupied block segments of a track are expressed as a set. It would be useful to identify graphical elements based on the elements of this set and to color all of them red at once (denoting that the blocks are occupied). To do this, we present an observer called *set observer* that is capable of selecting graphical elements based on a user-defined set expression. Table 4.5 shows the available options for the set observer.

To illustrate the use of the set observer consider Listing 4.7. The purpose of the observer is to set the *fill* of all pressed request buttons to *green*. To do this, we define the set selector based on the variable *request* which defines the set of floor numbers where the request button has been pressed (line 3). Since the ids of the graphical elements that represent the request buttons have the form "bt_*nr*", where *nr* is the respective floor number (-1, 0 or 1), we override the prefix using the convert function (lines 4 to 6). The returned prefix is composed of the string "#bt_" and the floor number (e.g. "#bt_0"). Finally, in lines 7 to 10 we define the *actions* triggered on the graphical elements that matches the composed *set* selector: we color the graphical elements in *green* (denoting the buttons that are pressed).

Table 4.5.: Available options for set observer

Name	Type	Required	Description
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
set	string	yes	The result of the defined *set* expression is used to establish a *set selector* which in turn is used to find child graphical elements of the graphical element that matches the *selector* of the observer. The elements of the set are joined with the prefix "#" (e.g. "#ele1,#ele2,#ele3,...").
convert	function	no	The *convert* function is called for each element of the defined set. It returns an element selector of the form "#id", where *id* is the identifier of the element. The user can also override the method.
actions	list	yes	A list of *actions* that determine the appearance and the behaviour of the *set* graphical elements.
attr	string	yes	The *attribute* of the elements that should be modified.
value	string	yes	The new *value* of the attribute.

```
bms.observe("set", {
  selector: "#request_buttons",
  set: "request",
  convert: function(element) {
   return "#bt_" + element;
  },
  actions: [{
   attr: "fill",
   value: "green"
  }]
});
```

Listing 4.7: Example set observer (JS)

4.8.4. Refinement Observer

Refinement is an important concept in the state-based formal methods classical-B and Event-B. It can be used to structure the development of a formal specification and to gradually introduce complexity and details (e.g. new variables or events). In order to support refinement in interactive formal prototypes, we introduce an appropriate observer with the available options shown in Table 4.6.

Listing 4.8 shows the use of the refinement observer based on the Event-B simple lift system. The purpose of the observer is to show the request buttons of the visualization (see Fig. 4.3) only if the corresponding refinement (the machine *m*2 where the buttons are

Table 4.6.: Available options for refinement observer

Name	Type	Required	Description
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
refinement	string	yes	The *refinement* that should be observed. The option accepts the name of a classical-B or Event-B machine.
enable	function	yes	The *enable* function is called whenever the defined *refinement* is part of the animation with its origin reference set to the graphical element that the observer is linked to. If no selector is defined, the *enable* function is called without parameters.
disable	function	yes	The *disable* function is called whenever the defined *refinement* is not part of the animation with its origin reference set to the graphical element that the observer is linked to. If no selector is defined, the *disable* function is called without parameters.

introduced) is part of the animation, otherwise the request buttons should be hidden. To do this, we register a new refinement observer to the group of request button graphical elements ("#request_buttons"). In line 3 we define the refinement (the name of the machine) that introduces the request buttons: $m2$. Lines 4 to 6 define the *enable* function that sets the *opacity* attribute of the graphical element to the value *1* (showing the request buttons) whenever the defined refinement is part of the animation. Otherwise, the *disable* function (lines 7 to 9) is called which sets the *opacity* attribute of the graphical element to the value *0* (hiding the request buttons).

```
 1 bms.observe("refinement", {
 2   selector: "#request_buttons",
 3   refinement: "m2",
 4   enable: function (origin) {
 5     origin.attr("opacity", "1")
 6   },
 7   disable: function (origin) {
 8     origin.attr("opacity", "0")
 9   }
10 });
```

Listing 4.8: Example refinement observer (JS)

4.8.5. Illustration of Observers

Figure 4.4 illustrates the effect of the example formula (Listing 4.5), predicate (Listing 4.6) and set (Listing 4.7) observers on the simple lift system (Fig. 4.3). Some

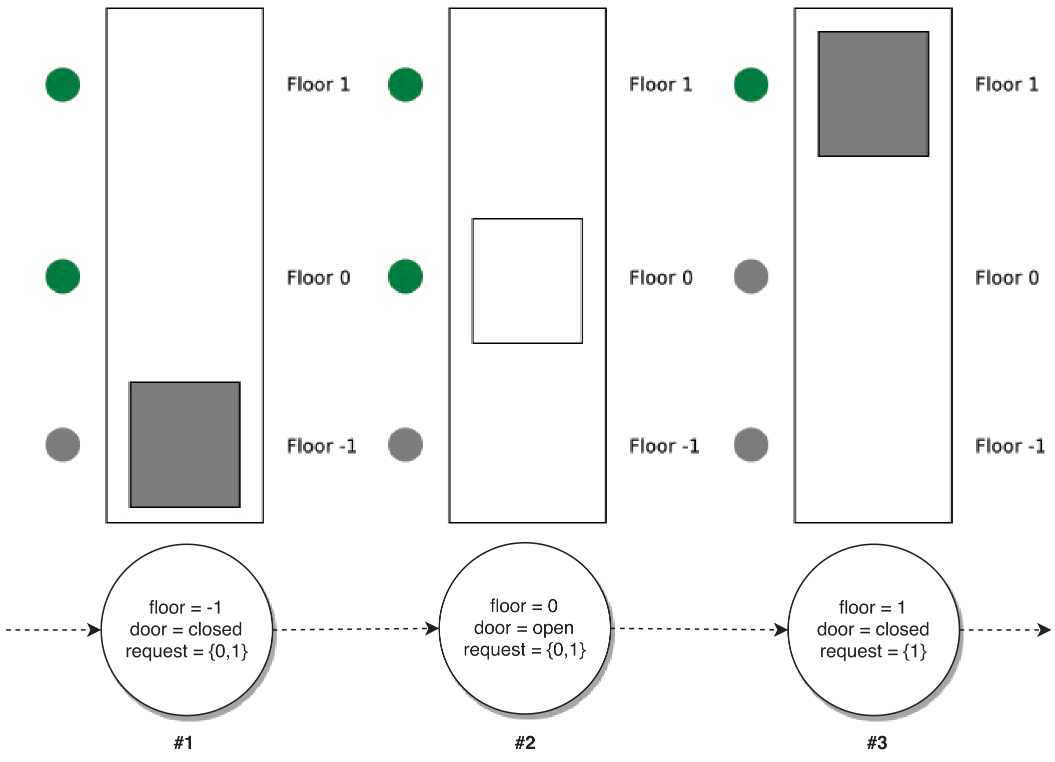


Figure 4.4.: Effect of formula, predicate and set observers on simple lift system

example states and their variable configurations are shown at the bottom of the figure. The effect of applying the observers is shown at the top of the figure. As can be seen in the figure, the effect of the formula observer is to change the y coordinate based on the current state value of the variable *floor* (denoting the movement of the door between floors). The effect of the predicate observer is to set the *fill* color of the lift door according to the evaluation of the predicate $door = open$. For instance, in state $\#2$ the predicate is *true*. Hence, the door is *white* denoting the door is opened. Finally, the set observer colors all pressed request buttons to *green* based on the set variable *request*.

4.8.6. Execute Event Handler

The execute event handler wires a list of classical-B operations or Event-B events to graphical elements. Table 4.7 shows the available options for the execute event handler.

Listing 4.9 shows how the execute event handler is used. In line 1, we register a new execute event handler for the graphical element that represents the request button for floor 0 (line 2). In lines 3 to 8, we define the event with the event's *name* (line 5) and

Table 4.7.: Available options for execute event handler

Name	Type	Required	Description
selector	string	yes	The *selector* matches a set of graphical elements which should be linked to the interactive handler.
events	list	yes	A list of *events* which should be wired with the graphical element.
name	string	yes	The *name* of the event.
predicate	string	no	The *predicate* for the event.
label	function	no	The *label* function returns a custom label as a string to be shown in the tooltip. The user can also return an HTML element. The function provides two arguments: the *origin* reference set to the graphical element to which the handler is linked and the *event* data.
callback	function	no	The *callback* function will be called after the event has been executed. If the event returns a value (e.g. when executing a classical-B operation with return value), the return *value* is passed to the *callback* function.

predicate[10] (line 6) which should be wired to the graphical element. Finally, in lines 9 to 11 we define a custom label based on the data of the *event* object which contains the name (event.name) and the predicate (event.predicate) of the defined event.

```
 1 bms.handler("executeEvent", {
 2   selector: "#bt_0",
 3   events: [
 4     {
 5       name: "send_request",
 6       predicate: "f=0"
 7     }
 8   ],
 9   label: function(origin, event) {
10      return "Push button " + event.predicate;
11   }
12 });
```

Listing 4.9: Example execute event handler (JS)

Figure 4.5 illustrates the effect of the execute event handler. A tooltip that lists all available events (disabled and enabled) will be shown when hovering over the graphical element or when clicking on the graphical element and if all events are disabled or more than one event is enabled. If only one event is enabled, it is executed directly when clicking on the graphical element. As an example, in the figure the user hovers over the

[10]The predicate defines the values of the parameters for the event.

request button on floor 0.

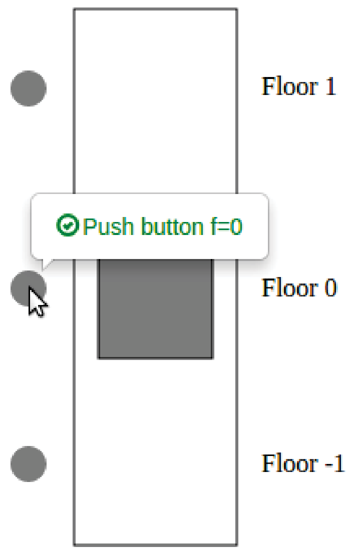


Figure 4.5.: Effect of execute event handler on simple lift system

4.8.7. Context-Sensitive Options

Each option for an observer or interactive handler (except of the selector option and the options that take a function as its value) can also define a function that returns its value. The *origin* (the reference to the graphical element that the observer or interactive handler is attached to) is passed to the value function as the first parameter. Defining a value function enables the user to determine the value of an option in the context of the linked graphical element. As an example, consider the execute event handler presented in Listing 4.9. The handler wires the *send_request* event with the predicate $f = 0$ to the graphical element that represents the lift request button on the floor 0 (*#bt_0*). Instead of creating similar execute event handlers for the other request buttons, we could also define a selector that selects all request buttons and a value function that returns the predicate in context of the matched graphical elements. Listing 4.10 shows an alternative execute event handler linked to all ellipse graphical elements that provide a *data-floor* attribute (*ellipse[data-floor]*). The *data-floor* attribute defines the floor number (-1, 0 or 1) of the respective request button. In line 6 to 8 we define a function that returns the predicate of the event *send_request* in context of the matched graphical elements, i.e. the function returns the predicate based on the *data-floor* attribute of the linked graphical element. For instance, the predicate function returns $f = 1$ for the graphical element where the *data-floor* attribute is set to 1. Based on context-sensitive options, we can create generic observers and interactive handlers: if we add more request floor buttons, the execute event handler in Listing 4.10 would be also valid for the new buttons.

```
bms.executeEvent({
  selector: "ellipse[data-floor]",
  events: [
    {
      name: "send_request",
      predicate: function (origin) {
        return "f=" + origin.attr("data-floor")
      }
    }
  ],
  label: function(origin, event) {
     return "Push button " + event.predicate;
  }
});
```

Listing 4.10: Context sensitive execute event handler (JS)

4.8.8. Other API Features

The BMotionWeb JavaScript API also provides some other features listed below:

Evaluate formulas manually. The JavaScript API provides the *bms.eval* function that takes a list of options defining the *formulas* to be evaluated and a *trigger* function that is called with the values of the formulas. The function is similar to the *formula observer*, except that the *bms.eval* function is executed once (in the current state) rather than after every state change.

Execute transitions manually. With the *bms.executeEvent* API function, the developer can execute a transition manually. The function takes a list of options, where the name and predicate options define the name and the predicate of the event to be executed respectively. Similar to the *execute event handler* the developer can optionally define a callback function that is called after the event has been executed. If the event returns a value (e.g. for a classical-B operation with a return value) the return *value* is passed to the *callback* function.

Initialization listener. The *bms.init* function takes a function as its parameter that is called whenever the animated formal specification is initialized. Thus, the developer could create the visualization according to static data coming from the formal specification (e.g. constants or external data from a database).

4.9. External Method Calls

BMotionWeb provides a Groovy API that can be accessed via the global variable *bms* within a Groovy script file. The Groovy API provides different functions to programatically control the integrated animation engine and to interact with the animated formal specification, e.g. to access the state space or trace of the animated formal specification. The developer can also register external methods that are evaluated on the server side. The registered methods accept arguments from the client and may also return data to the client. Listing 4.11 demonstrates the *bms.registerMethod* Groovy API function.[11] The method takes two arguments: the first argument defines the name under which the method should be registered, and the second argument is a closure that defines the actual method. For instance, in line 1 we register a method called *random* with a parameter n and the method body defined in lines 2 to 11. The purpose for this method is to randomly execute n events in the animated formal specification, where n is a number passed to the method. If a number below or equal zero has been passed to the method the method returns an error message. Otherwise the method randomly executes the event and returns a success message.

Since BMotionWeb integrates with the ProB animation engine, some of the ProB functionality is exposed to the user via the BMotionWeb Groovy API. ProB is tightly integrated with the Groovy scripting language. Everything from the constraint solver to the user interface is exposed via the scripting language.[12] For instance, in lines 5 and 7 in Listing 4.11 we access the current trace (*bms.getTrace()*) and execute a random event (*trace.anyEvent()*) of the animated formal specification respectively. These methods then call the appropriate methods within the ProB Java API.

```
 1 bms.registerMethod("random", { n ->
 2  if(n <= 0) {
 3   return "Only numbers greater than 0 are allowed.";
 4  } else {
 5   def trace = bms.getTrace();
 6   1.upto(n, {
 7    trace = trace.anyEvent();
 8   });
 9   bms.getAnimationSelector().traceChange(trace);
10   return n + " events have been executed.";
11  }
12 });
```

Listing 4.11: Register method on the server side (Groovy)

To use a registered method on the client side, the method can be wired to graphical elements (e.g. with an observer or interactive handler) or the developer can call the method manually (see lines 11 to 18 in Listing 4.12). Lines 2 to 9 in Listing 4.12

[11] A detailed list of the BMotionWeb Groovy API functions is given in Appendix C.
[12] A documentation of the ProB Java API is available at [STU].

Table 4.8.: Available options for method observer and execute method handler

Name	Type	Required	Description
selector	string	no (observer) yes (handler)	The *selector* matches a set of graphical elements which should be linked to the observer or handler.
name	string	yes	The *name* of the registered server side method.
args	list	no	The *args* that should be passed to the registered server side method.
callback	function	no	The *callback* function is called whenever the server side method returns a value with its origin reference set to the graphical element that the observer or handler is linked to and the return *value* of the method.

demonstrate the *execute method handler* that executes a registered server side method when the user clicks on the linked graphical element. In line 2 we register the handler on the graphical element that matches the selector *#button* (line 3). In line 4 and 5 we define the *name* and *args* (the arguments that should be passed to the method) of the method to be called. In lines 6 to 8 we define a *callback* function that is called whenever the method on the server side returns a value. The *origin* (the reference to the graphical element) and the returned *data* is passed to the callback function. In a similar fashion, an observer can be defined for observing a registered server side method (i.e. the method is called after every state change). Table 4.8 gives an overview of the available options for the method observer and execute method handler.

```
 1 // Register execute method handler
 2 bms.handler("method", {
 3   selector: "#button",
 4   name: "random",
 5   args: [10],
 6   callback: function(origin, data) {
 7    alert(data);
 8   }
 9 });
10
11 // Call method on server side manually
12 bms.callMethod({
13  name: "random",
14  args: [10],
15  callback: function(msg) {
16   alert(msg);
17  }
18 });
```

Listing 4.12: Use registered method on client side (JS)

A student at the university of Düsseldorf used the Groovy API in his master thesis

[Hoe16] for implementing an interactive formal prototype of a classical-B chess engine. The purpose of the Groovy script is to compute productive moves in a running chess game. Another example application using the Groovy API is shown in Chapter 7, where we use the Groovy API to replay a user defined trace to demonstrate a specific behaviour of the developed system.

4.10. Visual Editor

An important component of BMotionWeb is the built-in visual editor. The overall goal of the editor is to facilitate the rapid creation of visualization templates. The editor has been implemented and adapted based on *method draw*, a web based SVG editor.[13] Figure 4.6 shows the editor while editing the visualization template of the simple lift system interactive formal prototype. The editor consists of a palette for creating graphical elements, like shapes, labels, and images and a view for managing the properties of graphical elements. Graphical elements can be added to a canvas which provides like drag and drop, undo/redo, copy/paste and zooming.

The visual editor also supports the creation of observers and interactive handlers. Two additional views (one for creating observers and a second for creating interactive handlers) are available for this purpose. As an example, Fig. 4.7 shows the observers view. The view lists all observers with their corresponding options for the current edited visualization template. The user can edit the options of an observer directly in the observers view. If an option has a JavaScript function as its value, a JavaScript editor is shown when editing the option. For instance, the left side of Fig. 4.7 shows the JavaScript editor for the trigger function of the formula observer that is wired to the graphical element *#door*. The user only needs to provide the body of the function. The arguments (*origin* and *values*) are passed directly to the function body while running the interactive formal prototype.

4.11. BMotionWeb Application Versions

The following subsections describe the available application versions of BMotionWeb and their characteristics.

4.11.1. Desktop Application

The client and server of BMotionWeb can be run in a standalone desktop application using *electron*, a framework for building cross-platform desktop applications using

[13]https://github.com/duopixel/Method-Draw.

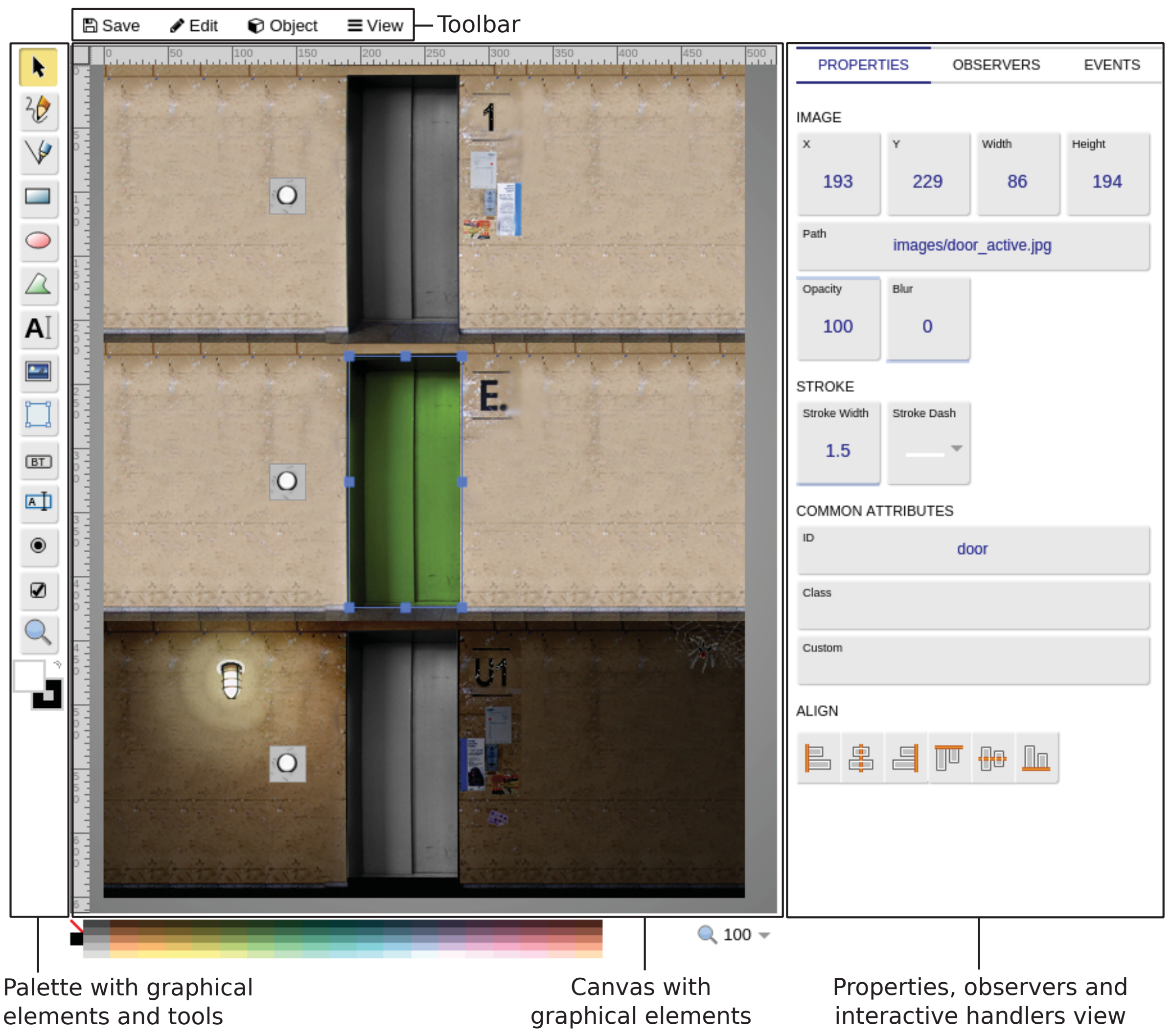
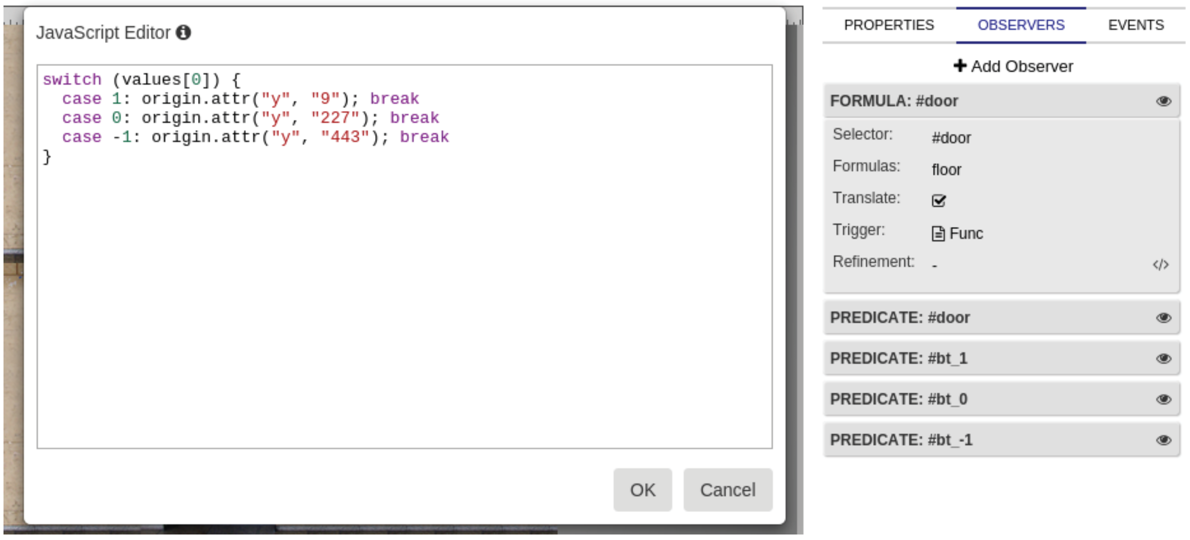


Figure 4.6.: Built-in visual editor of BMotionWeb

JavaScript.[14] There is no need to start the server as a separate process since it is started automatically in the background and stopped when closing the application. Figure 4.8 shows a screenshot of the desktop application running two variants of the simple lift system interactive formal prototype. Once the application is started, the user can open a visualization template by clicking on the box in the middle of the lower left window shown in Fig. 4.8 and select the BMotionWeb manifest file (see Section 4.4.1) of the visualization template or just drag and drop the manifest file into the box. The user can also open a visualization template via the top menu: File ⟩ Open Visualization. Opening the manifest file will start the interactive formal prototype in a separate window. This includes animating the linked formal specification and rendering the visualization template.

[14]http://electron.atom.io.


JavaScript Editor

```
switch (values[0]) {
  case 1: origin.attr("y", "9"); break
  case 0: origin.attr("y", "227"); break
  case -1: origin.attr("y", "443"); break
}
```

FORMULA: #door

Selector: #door
Formulas: floor
Translate: ✓
Trigger: Func
Refinement: -

PREDICATE: #door
PREDICATE: #bt_1
PREDICATE: #bt_0
PREDICATE: #bt_-1

Figure 4.7.: Observers view in visual editor of BMotionWeb

A major benefit of the desktop application is that is supports running multiple instances of the same or of different interactive formal prototypes simultaneously. This enables the user to compare different states within an interactive formal prototype or two variants of interactive formal prototypes for the same formal specification at the same time as demonstrated in Fig. 4.8. Furthermore, the desktop application provides features for creating and editing visualization templates. To do this, it integrates the visual editor described in Section 4.10.

4.11.2. Rodin Integration

Rodin [Abr+10] is an openly available and extendable platform based on Eclipse RCP for developing Event-B specifications. There are a several external plug-ins for Rodin available, such as tools for proving, animation, model checking and visualization.[15] Since a plug-in for the ProB animator already exists, we also have created a Rodin plug-in for BMotionWeb. This enables the user to start an interactive formal prototype of an Event-B specification within the Rodin platform. Figure 4.9 demonstrates the BMotionWeb Rodin integration while running the interactive formal prototype of the simple lift Event-B specification.

To integrate a visualization template with an existing Event-B project, the user only needs to place the visualization files into a subfolder of the Event-B project. The Rodin integration displays and marks the subfolders which contain a BMotionWeb manifest file (*bmotion.json*) in the *Event-B explorer* as demonstrated in the lower left corner of Fig. 4.9. Once the corresponding Event-B specification is animated using the ProB

[15] A comprehensive list of plug-ins for the Rodin platform is available at http://wiki.event-b.org/index.php/Rodin_Plug-ins.

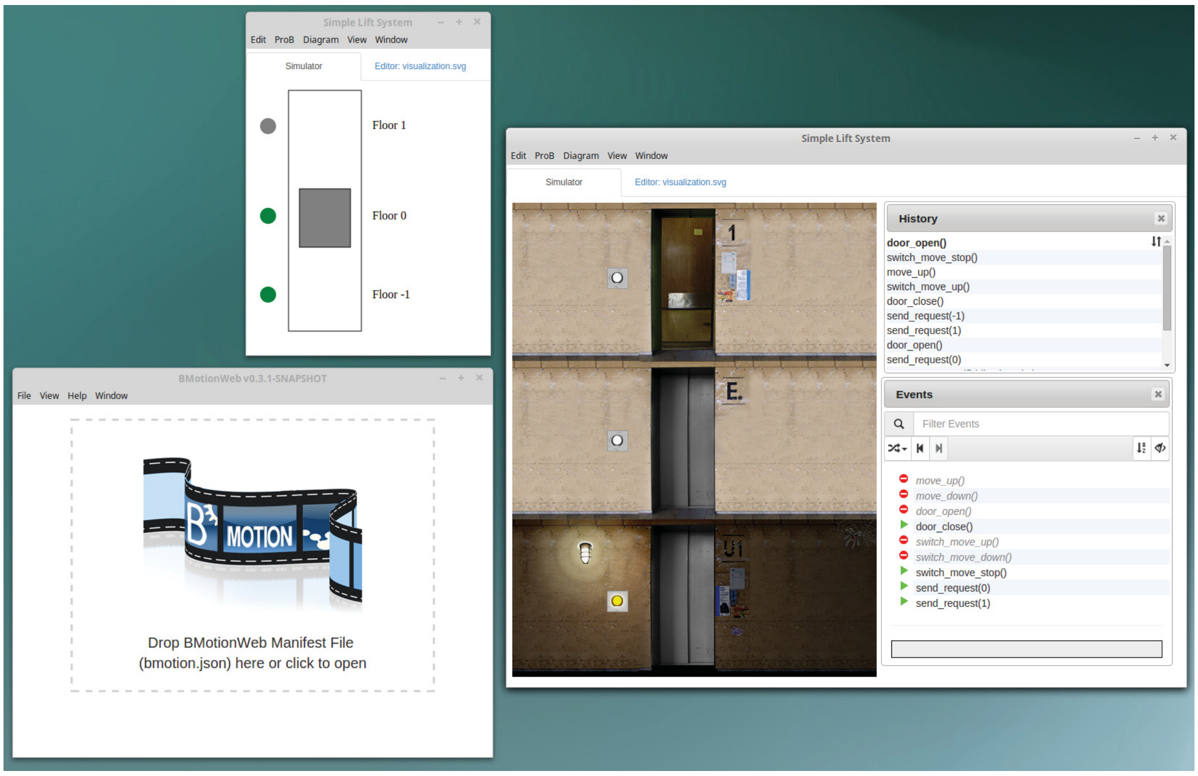
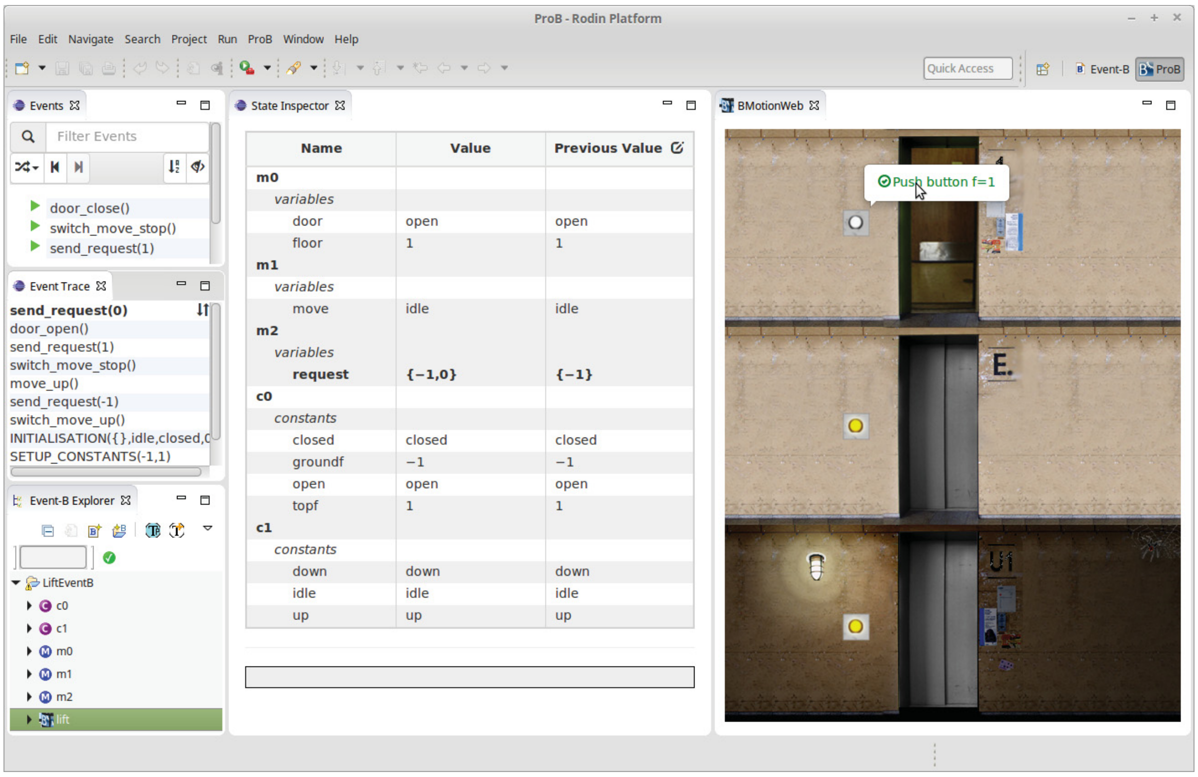


Figure 4.8.: BMotionWeb desktop application running two variants of the simple lift system interactive formal prototype

animation plug-in, the user can open the visualization template by double-clicking on the BMotionWeb icon shown in the Event-B explorer. The visualization template that opens is automatically linked to the current running animation.

4.11.3. Online Interactive Formal Prototypes

The BMotionWeb server can also be used to deploy interactive formal prototypes online. This can be in particularly useful for accessing an interactive formal prototype from other devices, such as tablets and mobile phones, and for sharing an interactive formal prototype with other stakeholders (e.g. during an online project meeting). For example, a domain expert could demonstrate a specific scenario in the system by interacting with the interactive formal prototype. All updates made on the interactive formal prototype are automatically shown to other stakeholders who have opened the same interactive formal prototype.



Events:

- door_close()
- switch_move_stop()
- send_request(1)

Event Trace:

```
send_request(0)
door_open()
send_request(1)
switch_move_stop()
move_up()
send_request(-1)
switch_move_up()
INITIALISATION({},idle,closed,c
SETUP_CONSTANTS(-1,1)
```

Event-B Explorer:

```
LiftEventB
  c0
  c1
  m0
  m1
  m2
  lift
```

Name	Value	Previous Value
m0		
variables		
door	open	open
floor	1	1
m1		
variables		
move	idle	idle
m2		
variables		
request	**{−1,0}**	**{−1}**
c0		
constants		
closed	closed	closed
groundf	−1	−1
open	open	open
topf	1	1
c1		
constants		
down	down	down
idle	idle	idle
up	up	up

Figure 4.9.: BMotionWeb Rodin Platform Integration

5 Combining BMotionWeb with other Visualization Techniques

5.1. Projection Diagrams

In the state-of-the-art study (see Chapter 2) we already have identified the potential of state space visualization approaches. However, the study also reveals limitations and disadvantages of these approaches with respect to the research problem stated in Chapter 1. The state space explosion problem in particular can make the creation and inspection of a state space visualization difficult or even impossible. To overcome this challenge, we present an approach to considerably reduce the complexity of a state space visualization by creating *projection diagrams*.[1] The main objective of the approach is to support human analysis of the system by highlighting relevant aspects of the formal specification (e.g. certain variables or a particular behavior) while hiding information that is not relevant from the diagram. The approach has been implemented into the ProB toolset with support for Event-B, Classical-B, TLA$^+$ and Z specifications. However, it is generic, so it can also be integrated into another tool that is capable of producing a state space for a formal specification.

The second part of this section extends the approach by combining a projection diagram with an interactive formal prototype. The resulting projection diagram consists of the basic projection diagram enhanced with graphical elements that come from the combined interactive formal prototype. A major benefit of this approach is the fact that the diagram can be generated from the interactive formal prototype directly without the user having to know the variables of the formal specification or having to type expressions in a formal language. In this section, we explain the approach and provide an implementation that comes as an extension of BMotionWeb.

5.1.1. Basic Projection Diagram Algorithm

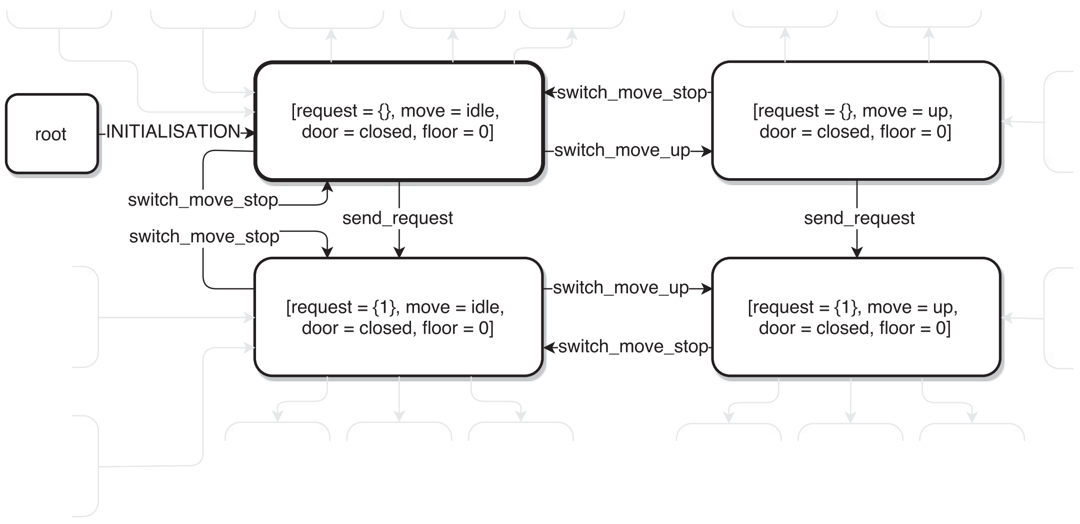
We explain our approach based on the Event-B simple lift system specification shown in Appendix A.1.2. The starting point of our approach is to explore the state space of

[1]This chapter presents a joint work with Michael Leuschel and is described in [LL15].

a formal model. This can be achieved via model-checking [CGP99] or interactively with animation [HLP13]. Note that for our approach it is not mandatory to exhaustively explore the full state space of the formal specification. The algorithm can also be applied on partial explored state spaces and provides feedback about which states have not yet been fully explored (see Section 5.1.1). The state space can be viewed as a non-deterministic labeled transition system (LTS):

Definition 1 (LTS) *An LTS is a 4-tuple* (Q, Σ, q_0, δ) *where* Q *is the set of states,* Σ *the alphabet for labelling the transitions,* q_0 *the initial state and* $\delta \subseteq Q \times \Sigma \times Q$ *is the transition relation. By* $q \xrightarrow{e} q'$ *we denote that* $qeq' \in \delta$*.*

Figure 5.1 shows an excerpt of an example LTS for the simple lift system (the full LTS covers 86 states and 242 transitions). Each node in the graph represents a state within the specification, where each state is defined by a particular configuration of the variables in the specification. We use the notation $[v_1 = r_1, ..., v_n = r_n]$ to denote the configuration of a state, where $v_1 = r_1, ..., v_n = r_n$ are the variables (v_x) and their values (r_x) in the respective state. For instance, the initial state q_0 (the node with the incoming transition labeled with *INITIALISATION*) has the configuration $[\mathit{request} = \{\}, \mathit{move} = \mathit{idle}, \mathit{door} = \mathit{closed}, \mathit{floor} = 0]$.



root
INITIALISATION
[request = {}, move = idle, door = closed, floor = 0]
switch_move_stop
switch_move_up
[request = {}, move = up, door = closed, floor = 0]
switch_move_stop
send_request
switch_move_stop
send_request
[request = {1}, move = idle, door = closed, floor = 0]
switch_move_up
switch_move_stop
[request = {1}, move = up, door = closed, floor = 0]

Figure 5.1.: LTS of the simple lift system (excerpt)

The edges in the graph represent the possible transitions δ of the LTS. In Event-B, a transition is the execution of an *event*, which is specified as a generalised substitution allowing deterministic and non-deterministic assignments to be specified. Each transition is labeled with the corresponding event name, where

$\Sigma = \{\mathit{move\_up}, \mathit{move\_down}, \mathit{door\_open}, \mathit{door\_close}, \mathit{send\_request},$

$$switch\_move\_up, switch\_move\_down, switch\_move\_stop\}$$

defines the names of the possible events. For instance, the event *switch_move_up* can modify the value of the variable *move* from *idle* to *up*, which is denoted by the transition

$$[request = \{\}, move = idle, door = closed, floor = 0] \xrightarrow{switch\_move\_up} [request = \{\}, move = up, door = closed, floor = 0]$$

shown in Figure 5.1.

The next step in the construction of a projection diagram for an LTS consists of defining a *projection function*. All states with the same value for the projection function are merged into an *equivalence class*. A transition leads from one equivalence class C to another C' if there is a transition from one state $q \in C$ to a state $q' \in C'$. Formally, one can define the projection of an LTS as follows:

Definition 2 (Projection) *Let $L = (Q, \Sigma, q_0, \delta)$ be an LTS and p a projection function with domain Q. The projection of the LTS using p, denoted by L^p, is defined to be the LTS $(Q^p, \Sigma, p(q_0), \delta^p)$, with $Q^p = \{p(q) \mid q \in Q\}$ and $\delta^p = \{p(q) \xrightarrow{e} p(q') \mid q \xrightarrow{e} q' \in \delta)\}$.*

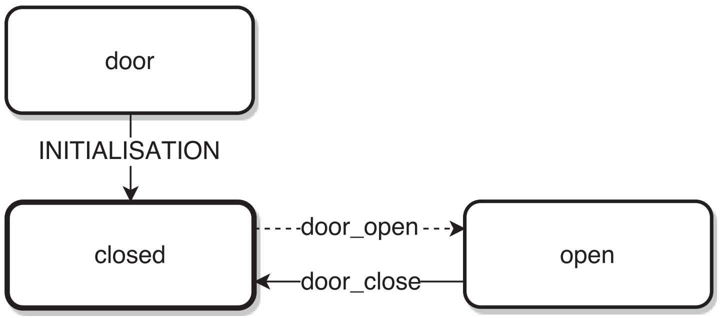


Figure 5.2.: Projection of the LTS onto the variable *door* for the simple lift system

Each element in Q^p represents an equivalence class, where each equivalence class merges the states of Q (the states of the original LTS) that have the same value for the projection function p. To illustrate the idea of a projection, consider Fig. 5.2. The diagram shows the projection of the LTS of Fig. 5.1 onto the variable *door* using the projection function $p([request = rv, move = rx, door = ry, floor = rz]) = ry$. Since the *door* variable can have the two values *closed* and *open*, we have two equivalence classes: one that merges all states with $door = closed$ and one that merges all states with $door = open$. Obviously, the projection of an LTS may not be equivalent to the original LTS, since the sequences of the events are not necessarily possible in the original LTS. However, all sequences of the original LTS are possible in any projection of it.[2]

[2]I.e., the original LTS is a trace refinement of the projection LTS.

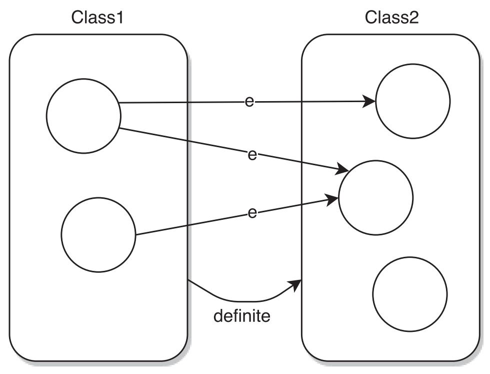


Figure 5.3.: Definite edge

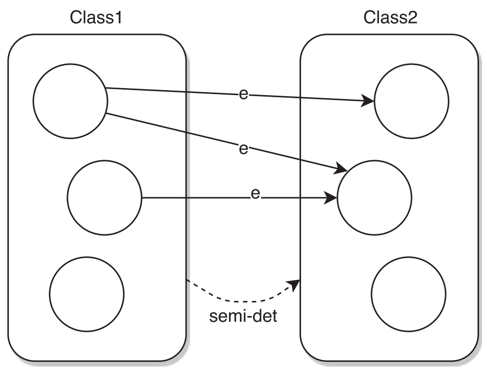


Figure 5.4.: Semi-deterministic edge

Categorizing Edges and Equivalence Classes

To provide a more refined projection, we categorize the equivalence classes and edges. We distinguish between *definite* and *non-definite* and between *deterministic* and *non-deterministic* edges. In addition, we distinguish between three types of equivalence classes: the equivalence classes that contain only a single state, the equivalence classes that merges at least two states, and the equivalence classes that have not yet been fully explored (e.g., if the state space has not been explored exhaustively).

In the following subsections, we explain the different types of edges and equivalence classes and illustrate them with an example. To do this, let $L = (Q, \Sigma, q_0, \delta)$ be an LTS and $L^p = (Q^p, \Sigma, p(q_0), \delta^p, E)$ its projection. Given an edge $x \xrightarrow{e} y \in \delta^p$, we denote x as the *source* and y as the *target* equivalence class. Moreover, we call an edge $x \xrightarrow{e} y \in \delta^p$ *enabled* for a particular state q, with $q \in x$ if $\exists\, q' \cdot (q' \in y \wedge q \xrightarrow{e} q' \in \delta)$.

Definite Edges. An edge is definite, if and only if it is enabled in *all* states of the source equivalence class. Thus, the set of all definite edges of L^p can be defined as follows:

$$Definite = \{x \xrightarrow{e} y \mid x \xrightarrow{e} y \in \delta^p \wedge \forall\, q \cdot (q \in x \Rightarrow \exists\, q' \cdot (q' \in y \wedge q \xrightarrow{e} q' \in \delta))\}.$$

Figure 5.3 illustrates the idea of a definite edge: there is a definite edge between the equivalence classes *Class1* and *Class2* whenever e is enabled in all states from the source equivalence class (*Class1*). An edge is non-definite if it is not definite. In order to distinguish the different edge types in the projection diagram, definite edges are drawn as solid lines while non-definite edges are drawn as dashed lines. An example can be seen in Fig. 5.2. The *door_close* edge is possible in all states with *door* = *open* and is the only definite edge in the diagram. The other edge *door_open* is semi-deterministic as described in the next section.

Semi and Non-Deterministic Edges. An edge e is called semi-deterministic if the corresponding event always leads to the same target equivalence class (*Class2*) from the source equivalence class (*Class1*). However, it does not have to be enabled in all states from the source equivalence class (*Class1*). This is illustrated in in Fig. 5.4. The dashed edge in Fig. 5.2 is also semi-deterministic since for some states merged into the source equivalence class the *door_open* event is not enabled (e.g. for states in which the lift cabin is moving). Thus, the set of all semi-deterministic edges of L^p is defined as follows:

$$SemiDet = \{x \xrightarrow{e} y \mid x \xrightarrow{e} y \in \delta^p \wedge \neg(\exists z \cdot (z \neq y \wedge x \xrightarrow{e} z \in \delta^p))\}.$$

Furthermore, we denote an edge as non-deterministic if it is *not* semi-deterministic. Thus, the set of all non-deterministic edges of L^p is composed of all edges (δ^p) apart from the semi-deterministic edges ($SemiDet$):

$$NonDet = \delta^p \setminus SemiDet.$$

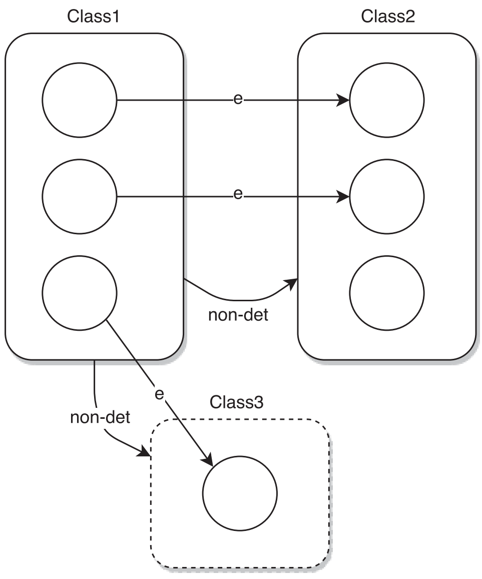


Figure 5.5.: Non-deterministic edge

Figure 5.5 illustrates a non-deterministic edge. Given the three equivalence classes *Class1*, *Class2* and *Class3*, the edge e is non-deterministic if e is enabled and it leads to at least two distinct target equivalence classes (e.g. *Class2* and *Class3*).

An example can be seen in Fig. 5.6. The figure demonstrates the projection onto the variable *request* for the simple lift system. For instance, the outgoing edges labeled with *send_request* for the equivalence class $[request = \{\}]$ are non-deterministic as they lead to the three distinct target equivalence classes $[request = \{0\}]$, $[request = \{-1\}]$ and $[request = \{1\}]$. Indeed this is to be expected: in the initial state (the state where no requests have been made yet) we can request the lift on all three floors and thereby enter a distinct state in the system.

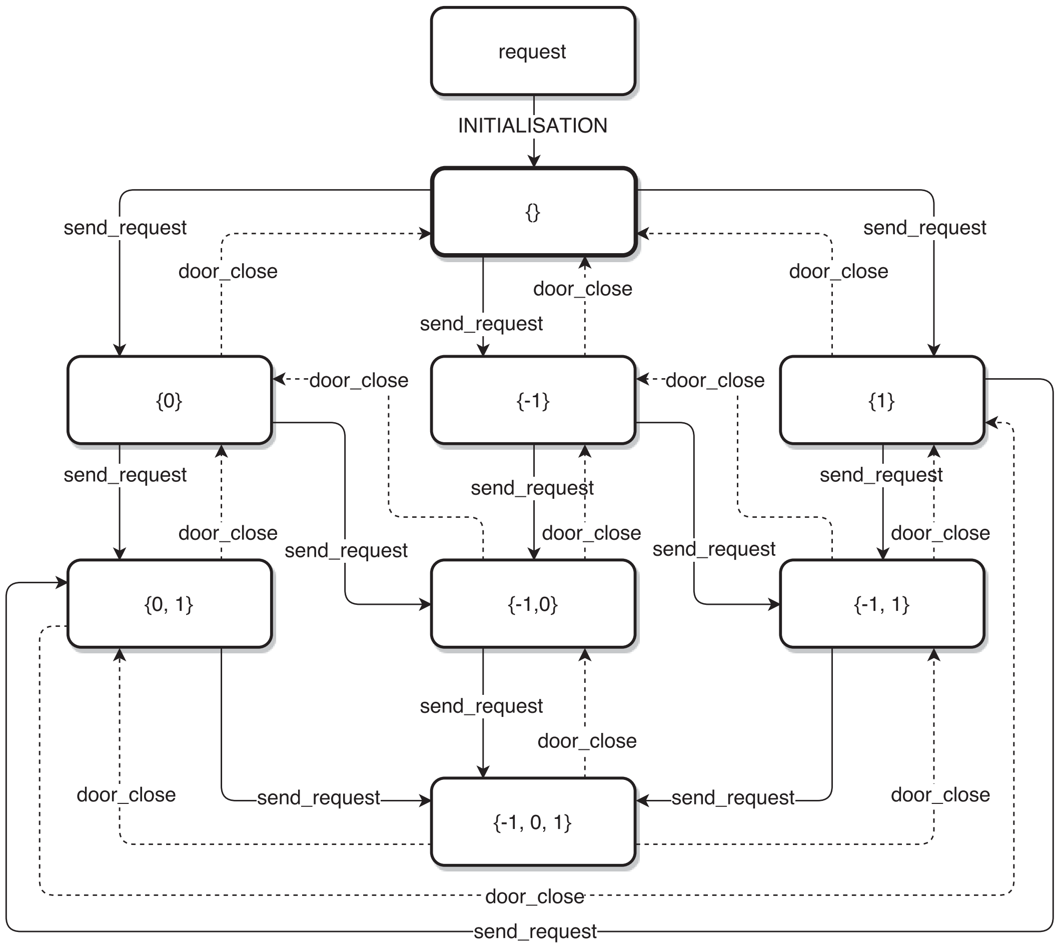


Figure 5.6.: Projection on the LTS onto the variable *request* in the simple lift system

Deterministic and Non-Deterministic Definite Edges. The set of all edges of L^p that are deterministic and definite is defined as:

$$DetDef = SemiDet \cap Definite.$$

As an example, the edge e shown in Fig. 5.3 is definite and deterministic. This is because e is enabled in all states in the source equivalence class (*Class1*) and leads to the same target equivalence class (*Class2*).

Moreover, edges that are non-deterministic and definite are defined as follows:

$$NonDetDef = NonDet \cap Definite.$$

For instance, the edge shown in Fig. 5.5 is definite and non-deterministic since it is enabled for all states in the source equivalence class (*Class1*) and leads to two distinct target equivalence classes (*Class2* and *Class3*). Another example is shown in Fig. 5.6: the outgoing edges labeled with *send_request* (apart from the outgoing edges of the equivalence classes $[request = \{0, 1\}]$, $[request = \{-1, 0\}]$ and $[request = \{-1, 1\}]$) are definite and non-deterministic.

Single State and Partial Equivalence Classes. An equivalence class is *single*, if the equivalence class contains only one state. Thus, the set of all single equivalence classes can be defined as follows:

$$\mathit{Single} = \{x \mid x \in Q^p \wedge \mathit{card}(\{q \mid q \in Q \wedge p(q) = x\}) = 1\}.$$

For instance, the equivalence class *Class3* in Fig. 5.5 is *single* since it contains only one state.

Furthermore, we highlight any equivalence classes that have not yet been fully explored, which can happen when not all states in the class have been reached by the model checker or animator. This means that additional outgoing edges and new equivalence classes could appear after further exploration of the state space. In order to distinguish the different types of equivalence classes in the projection diagram, single state equivalence classes are drawn with a dashed border, partial equivalence classes are drawn with a dotted border, and equivalence classes that are fully explored and contain at least two states are drawn with a solid border. In this thesis, however, we always suppose that the full state space has been explored and as such no equivalence classes with dotted borders appear in the projection diagrams.

5.1.2. Custom Expressions

In this section we present some further example applications of the projection diagram. From now on we will use projection functions of the form $p(q) = \mathit{eval}(E, q)$, where $q \in Q$, E is an expression over the variables and constants of the formal specification, and *eval* is the function that evaluates the expression E in state q. The projection function is thus defined by a "custom" expression E. With this scheme, we can project the LTS for a specification onto a single variable v ($E = v$) or on a set of variables $v_1, \ldots, v_k$ ($E = (v_1 \mapsto \ldots \mapsto v_k)$). We can also project onto particular properties of a variable v, e.g., its cardinality ($E = \mathit{card}(v)$) or its range ($E = \mathit{ran}(v)$).

Simple Lift System. Figure 5.7 illustrates how one can combine various variables into a single expression. The figure shows a projection of the LTS of the simple lift system onto the two variables *move* and *door* ($E = \mathit{move} \mapsto \mathit{door}$). The projection shows that our invariant about the controller of the simple lift system that the door must always be closed when the lift cabin is moving ($\mathit{move} \in \{\mathit{up}, \mathit{down}\} \rightarrow \mathit{door} = \mathit{closed}$)): in the only equivalence class where the door is *open*, the controller for moving the door is set to *idle* ($[\mathit{move} \mapsto \mathit{door} = \mathit{idle} \mapsto \mathit{open}]$). In all other equivalence classes the door is *closed*. Moreover, the door cannot be opened in a state where the controller is set to moving *up* or *down* since no *door_open* transition is enabled in the respective equivalence classes ($[\mathit{move} \mapsto \mathit{door} = \mathit{down} \mapsto \mathit{closed}]$) and ($[\mathit{move} \mapsto \mathit{door} = \mathit{up} \mapsto \mathit{closed}]$). The projection also confirms that the controller must first stop (*switch_move_stop*) the lift cabin before the door can be opened. Since the *switch_move_stop* transitions are definite edges they can be executed in all states merged in the respective source equivalence class which denotes that the lift cabin can always be stopped when the lift cabin is moving.

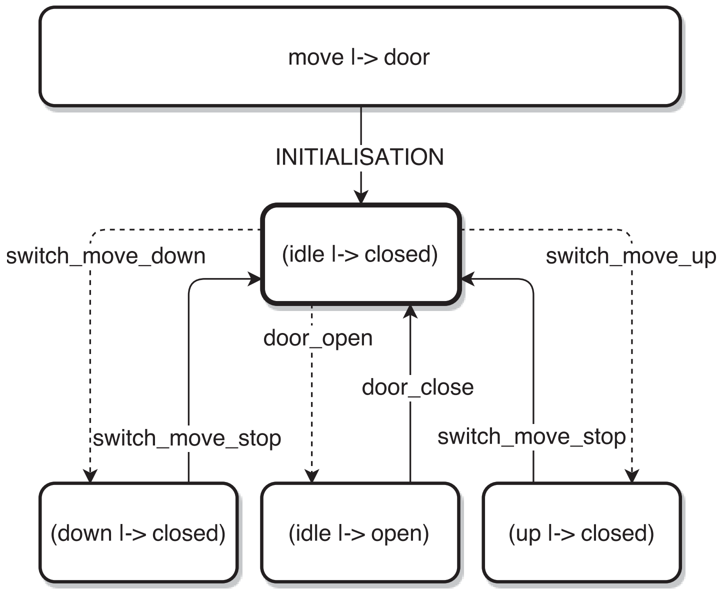


Figure 5.7.: Projection on LTS of the simple lift system onto the two variables *move* and *door* ($move \mapsto door$)

Scheduler. Figure 5.8 shows a projection of the LTS for the "standard" scheduler benchmark example from [LPU02] (also used in [LT05]), which schedules processes and keeps disjoint sets of waiting, ready and active processes. In the projection we abstract away from the process identities by computing the cardinality of these sets. Furthermore, we add these sets together to project onto the total number of processes ($E = card(ready) + card(waiting) + card(active)$). One can clearly see that only two events change the total number of processes: *new* and *del*. Moreover, *new* is always enabled when less than three processes exist, while *del* is only possible when more than one process exists, and is not always possible. This confirms what we intuitively assume, since active processes cannot be deleted right away. Figure 5.8 shows how one can focus on very specific aspects of a formal specification using the projection diagrams. We believe that one should generate a variety of projection diagrams for any particular formal specification — a different one for very specific aspects — and that they can or should be incorporated into the documentation accompanying the formal specification.

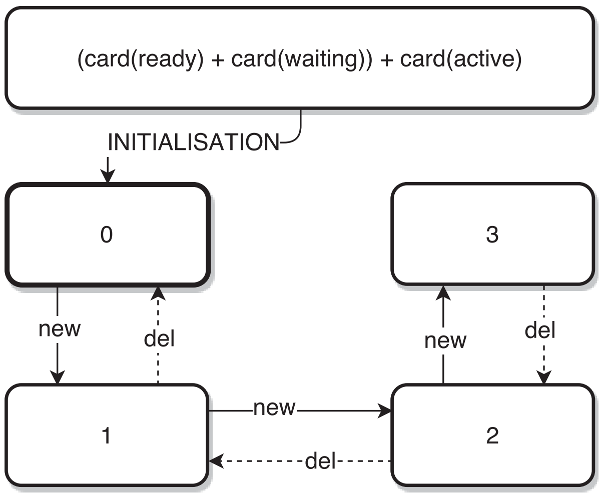


Figure 5.8.: Projection of the LTS of a classical-B process scheduler onto custom expression $card(ready) + card(waiting) + card(active)$

5.1.3. Combining with Interactive Formal Prototypes

In this section we present an extension of the approach that combines a projection diagram with an interactive formal prototype. The basic idea of the extension is to create a projection based on user-selected graphical elements. Based on the approach for computing a basic projection diagram described in Section 5.1.1, we can use Algorithm 2 to build a projection diagram combined with graphical elements for a given LTS and a set of user-selected graphical elements.

In line 2 we define an empty set of *nodes* that define the mapping from equivalence classes to a set of graphical elements which should represent the equivalence class. In lines 4 to 8, we determine the observers of the selected graphical elements and derive the formulas f_i that are required to draw the state of the selected graphical elements. Based on the formulas, we construct the projection expression $E = f_1 \mapsto ... \mapsto f_n$ (line 9) and compute the basic projection diagram (line 11) using the projection function $p(s) = eval(E, s)$ (line 10) as described in Section 5.1.1. For each equivalence class in the projection diagram, we compute the representation of the selected graphical elements according to the value of the projection function of the respective equivalence class $p(s)$ (lines 12 to 19). Note that if computed separately, all states in this equivalence class will yield the *same* representation for the selected graphical elements. In line 17 we assign the adapted graphical elements to the corresponding equivalence classes. If all equivalence classes in the projection L^p are processed, the algorithm builds the diagram based on the collected *nodes* (the states and the computed representations of the graphical elements) and the transitions of the projection δ^p (line 20).

The algorithm has been implemented into BMotionWeb with support for the state-based formal methods Event-B and classical-B. For generating the diagram, we use

Algorithm 2: Create projection diagram combined with graphical elements

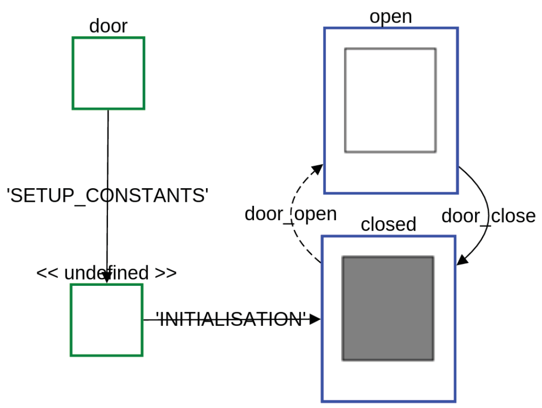
```
function createProjectionDiagram(($Q, \Sigma, q_0, \delta$) $L$ , graphicalElements $elems$)
    $nodes := \varnothing$
    $formulas := \varnothing$
    foreach $elem \in elems$ do
        foreach $o \in collectObservers(elem)$ do
            $formulas := formulas \cup collectFormulas(o)$
        end foreach
    end foreach
    $E := f_i \mapsto \ldots \mapsto f_n$, where $f_i \in formulas$
    $p(s) := eval(E, s)$
    $L^p := computeProjection(L, p(s))$
    foreach $q \in Q^p$ do
        foreach $elem \in elems$ do
            foreach $o \in collectObservers(elem)$ do
                $o.update(q)$
            end foreach
            $nodes(q) := nodes(q) \cup \{elem\}$
        end foreach
    end foreach
    $buildDiagram(nodes, \delta^p)$
end function
```

Cytoscape.js[3], a JavaScript library for the analysis and visualization of graphs. One main advantage of Cytoscape.js is that the diagram is interactive so that the user can rearrange the nodes and edges as desired.

To illustrate this idea, consider the interactive formal prototype of the simple lift system (see Appendix B.2). Based on the interactive formal prototype we can create the projection diagram as demonstrated in Fig. 5.9 (left side). The figure shows the same projection as in Fig. 5.2. However, it was created based on the graphical element that represents the lift cabin door (#door) using the projection function $p(s) = eval(E, s)$, where $E = door$ is automatically derived from the *formula observer* shown on the right side of the figure. Each rectangle in the diagram represents an equivalence class (all states with the same value for the expression E) and is labeled with the associated expression value. A directed edge between two equivalence classes represents a transition which is labeled with the associated event name and is styled (solid or dashed) according to the approach presented in Section 5.1. The green nodes are not part of the specification's state space. They are artificially introduced and represent the system before it is initialized. To compute the representation of the graphical element for an equivalence class, we apply the formula observer using the value of the projection function of the

[3]http://js.cytoscape.org.

respective class. The computed representation of the graphical element is then assigned to the equivalence class. For instance, the diagram shows the two possible states for the cabin door (*open* and *closed*) and its graphical representation.



```
bms.observe("formula", {
  selector: "#door",
  formulas: ["door"],
  trigger: function(origin, val) {
    if(val[0]==="open") {
      origin.attr("fill", "white");
    } else if(val[0]==="closed") {
      origin.attr("fill", "gray");
    }
  }
});
```

Figure 5.9.: Projection on door graphical element (left) and door observer (right)

Although developing a visualization requires extra effort, the benefits of combining it with a projection diagram can be considerable. As an example, consider the projection on the graphical element that represents the simple lift system with the three floors and the cabin door (#lift) shown in Fig. 5.10. The projection diagram was generated based on the projection function $p(s) = eval(E, s)$ where $E = door \mapsto floor$ is automatically derived from the formula observers observing the *door* variable and the *floor* variable respectively. One can see at a glance (without knowing about the underlying specification or the formalism used) that the cabin is able to stop and to open the door at all three floors. The diagram also confirms that the cabin door must always be closed (indicated by a gray door) before the lift can move (*move_down* or *move_up*). This is indicated by the solid edges labeled with the event *door_close*. More example applications of the projection diagram feature are demonstrated in Chapter 7.

5.1.4. Evaluation

Table 5.1 shows some evaluation statistics obtained after applying the basic projection approach introduced in Section 5.1.1 (*runtime BP*) and the combined projection diagram described in Section 5.1.3 (*runtime CP*) to the case studies presented in this thesis. Moreover it shows the number of nodes and edges in the full state space and in the produced projection diagram (third and fourth column respectively). The statistics were obtained after the corresponding state space had been fully explored with ProB. We use the projection function $p(q) = eval(E, q)$, where $q \in Q$ and E is the *projection expression* (second column of Table 5.1). The measured time includes the actual runtime for both algorithms (implemented in ProB and BMotionWeb) without the time needed to exhaustively explore the full state space (i.e. the model checking time) and without the time needed for generating and producing the layout for the actual diagram. The

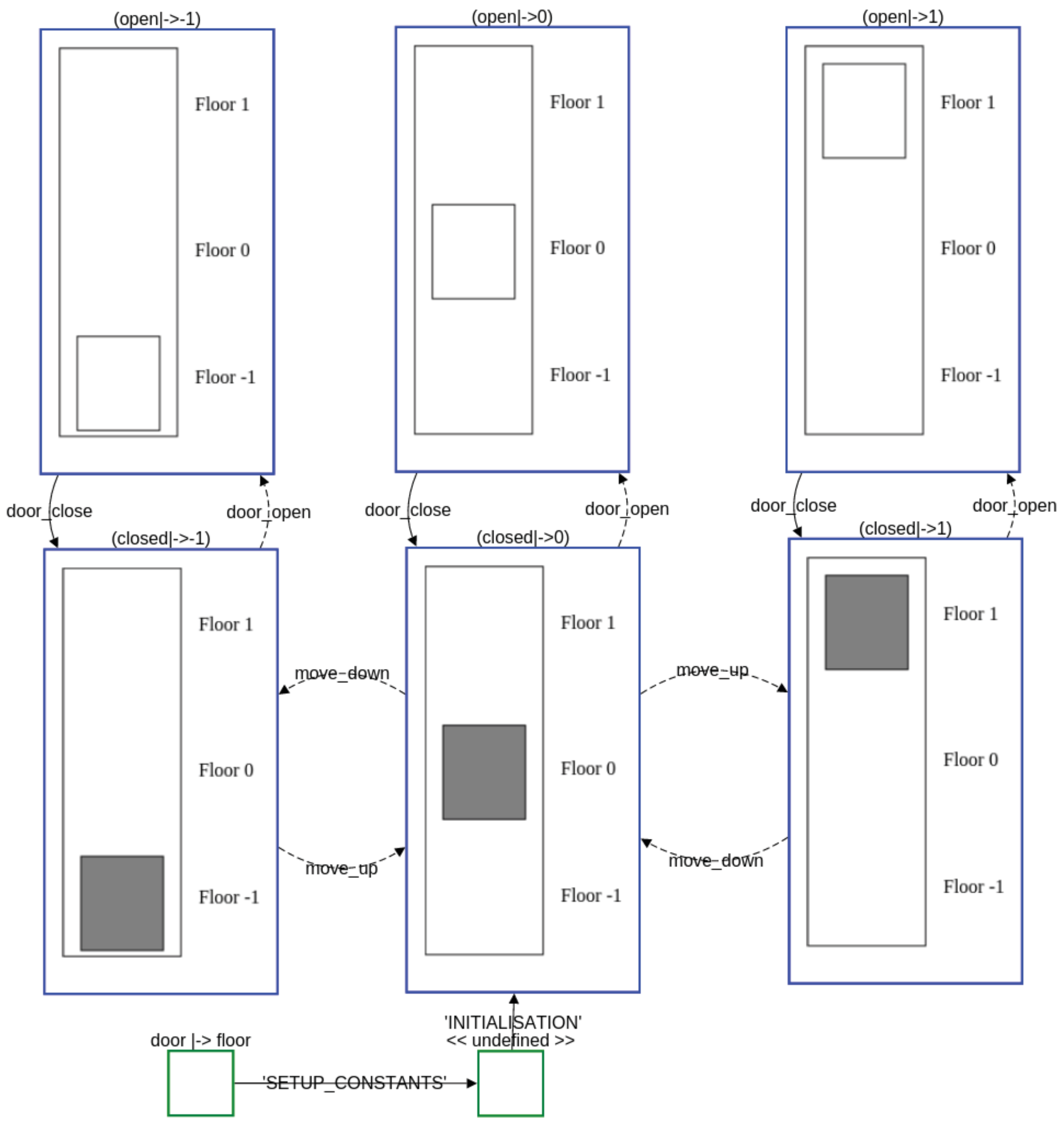


Figure 5.10.: Projection on the cabin of the simple lift system (#lift)

model checking and layouting time is not included because it depends on the model checker and layouting tool respectively. Moreover, the state space needs to be explored only once in order to generate multiple projection diagrams.

In general, the runtime of the CP takes longer than the BP. This is because the CP uses the BP to compute the actual data and needs some additional time to generate the graphical representation of the equivalence classes. Table 5.1 also confirms that the runtime of both algorithms (CP and BP) increases with the number of nodes and

Table 5.1.: Evaluation of projection diagram approach

		Nodes / Edges		**Runtime**	
Formal Spec.	**Expression**	**Full**	**BP**	**BP**	**CP**
Scheduler	$card(ready)$ $+card(waiting)$ $+card(active)$	36/121	5/7	0.01 s	-
Landing Gear 4th Ref (old)	$door(front) \mapsto gear(front)$	6,283/31,299	10/25	1.07 s	-
Landing Gear 4th Ref (early)	$handle \mapsto gear(front)$	6,283/31,299	7/13	0.34 s	-
Landing Gear 4th Ref	$doors(front) \mapsto gears(front)$	2,529/16,097	8/17	0.08 s	-
Landing Gear 5th Ref	$ran(gears) \mapsto handle$	25,217/149,041	15/34	0.83 s	-
	$doors(front)$		4/7	0.69 s	1.69 s
	$handle$		3/3	0.59 s	1.01 s
	$valve\_close\_door \mapsto close\_EV$		5/6	0.60 s	2.10 s
	$handle \mapsto gears(front)$		7/17	0.71 s	3.25 s
Landing Gear 6th Ref	$analog\_switch \mapsto general\_EV$	131,329/884,369	5/9	3.39 s	-
Simple Lift	$door$	186/838	3/3	0.01 s	0.40 s
	$floor \mapsto door$		8/12	0.01 s	0.82 s
	$move \mapsto door$		5/7	0.01 s	0.83 s
	$request$		9/25	0.01 s	-

(BP=basic projection, CP=combined projection with graphical elements) transitions of the full state space.

5.2. Trace Diagrams

The state-of-the-art study (see Chapter 2) showed that trace visualization techniques, i.e. visualization techniques that are based on a particular sequence of events, like for visualizing a particular trace/process or counter-example (see Section 2.2.3, Section 2.3.2 and Section 2.3.3) can be useful to support the validation of state-based and event-based formal specifications. In this section, we demonstrate the combination of such trace visualization techniques with state-based interactive formal prototypes.[4] To do this, we start with a definition of a *trace*:

Definition 3 (Trace) *Let $L = (Q, \Sigma, q_0, \delta)$ be an LTS. A trace ϕ of an LTS is a sequence of states and events starting with the initial state q_0 and ending with a state of Q, $\phi = q_0 e_1 q_1 e_2 \ldots e_n q_n$ such that $q_i \xrightarrow{e_{i+1}} q_{i+1}$ for all $0 \leq i < n$, where $n \geq 0$.*

ProB has different techniques for generating a trace for classical-B and Event-B formal specifications including animation, model-checking (e.g. checking invariant violations or deadlocks) or assertion checks based on LTL formulas [PL10]. We use the trace information provided by ProB as a base to generate the trace diagram. Given a particular trace and a set of user selected graphical elements, we can compute the representation of the graphical elements using Algorithm 3.

Algorithm 3: Create state based trace diagram combined with graphical elements

```
1  function createSBTraceDiagram(tr $\langle q_0 e_1 \ldots e_n q_n \rangle$, graphicalElements $elems$)
2      $nodes := \varnothing$
3      $edges := \varnothing$
4      for $i$=0 to $n$ do
5          $edges := edges \cup \{q_i e_{i+1} q_{i+1}\}$
6          foreach $ele \in elems$ do
7              $nodes(q_i) := nodes(q_i) \cup computeStateBased(q_i, ele)$
8          end foreach
9      end for
10     $buildDiagram(nodes, edges)$
11 end function
```

In line 2 and 3 we define an empty set of *nodes* and *edges* that define the mapping from states in a trace to a set of graphical elements which should represent the state and the transitions between the states respectively. The algorithm iterates through each state q_i of *tr* sequentially with $i \in \{0..n\}$ and computes the representation for each graphical element *ele* of the set of user selected graphical elements *elems*. For state-based

[4]In Chapter 6, we will also adapt the approach for event-based formal methods.

formal methods we assign the computed representation of the graphical element using the algorithm *computeStateBased* (see Algorithm 1 in Section 3.2) with the currently processed state q_i and the graphical element *ele* (line 7). If all the states in trace *tr* are processed, the algorithm builds the diagram with the collected *nodes* (the states and the computed representations of the graphical elements) and the *edges* (line 10).

The algorithm has been implemented into BMotionWeb with support for the state-based formal methods Event-B and classical-B. As with the projection diagram approach (see Section 5.1.3), we also use Cytoscape.js for generating the diagram.

Figure 5.11 shows an example trace diagram for the trace

$$\begin{aligned}&\langle \mathit{setup\_contants}, \mathit{initialise\_machine}, \mathit{send\_request}(1), \mathit{switch\_move\_up}(),\\ &\qquad \mathit{move\_up}(), \mathit{switch\_move\_stop}(), \mathit{door\_open}(), \mathit{door\_close}(), \mathit{send\_request}(-1)\rangle\end{aligned}$$

in the simple lift Event-B specification. Each rectangle represents a state including the corresponding representation of the selected graphical elements. A directed edge between two states represents the event and is labeled with the event's name.

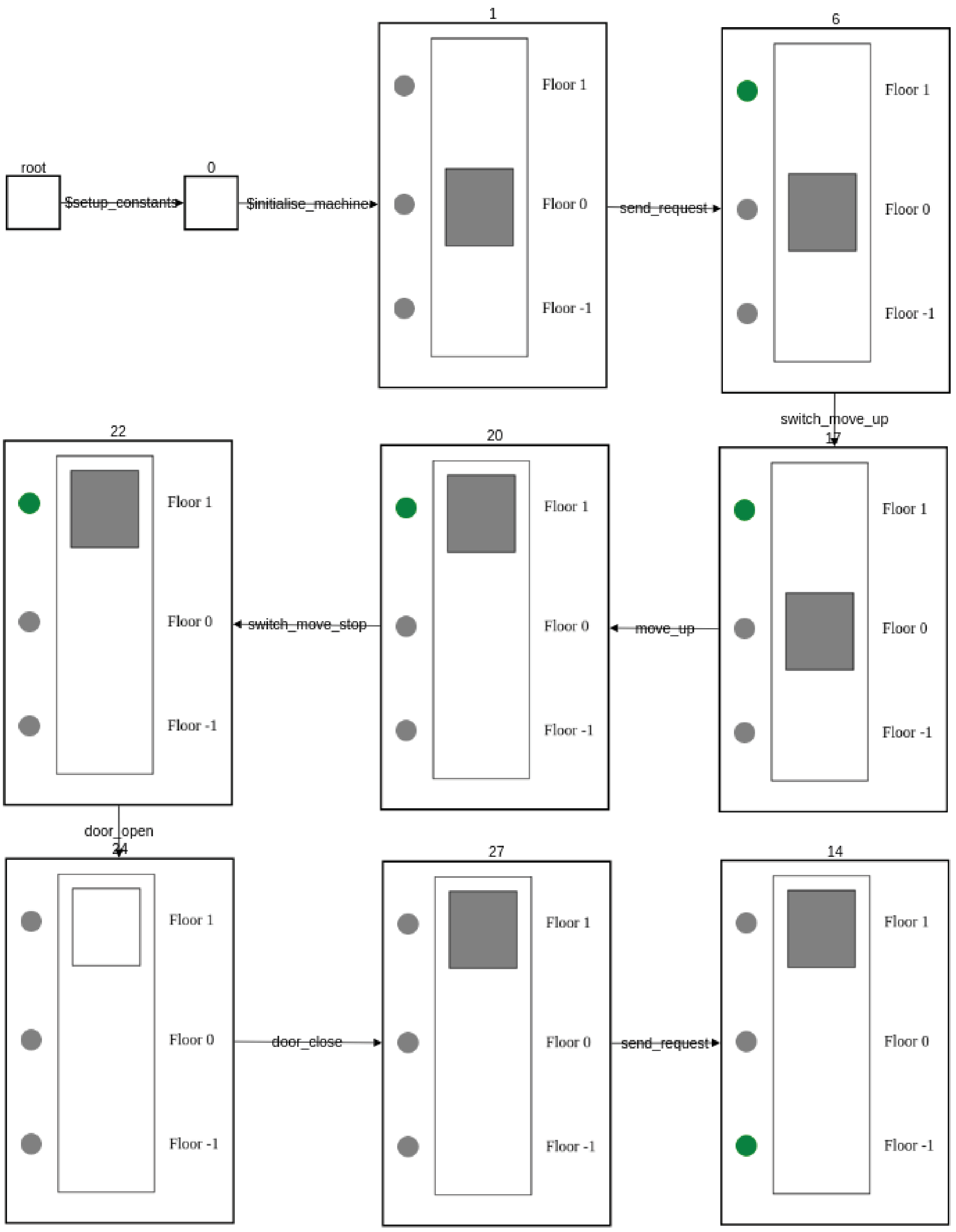


Figure 5.11.: Trace diagram for the simple lift system

6 Extending BMotionWeb for CSPM

6.1. Introduction

Inspired by the successful creation of interactive formal prototypes for state-based formal specifications, we developed an approach for creating interactive formal prototypes for CSP (Communicating Sequential Processes).[1] CSP is a notation mainly used for describing concurrent and distributed systems. There are two major CSP dialects: CSPM [SA11] and CSP# [Sun+09a]. The most popular tools that support model checking for CSPM specifications are FDR [Gib+14] and ProB [LF08]. Support for animating processes of CSPM specifications is provided by ProB and ProBE [For]. The more recent CSP# [Sun+09a] is supported by the PAT system [Sun+09b]. In this chapter we present an approach (method and implementation) for creating interactive formal prototypes of CSPM specifications. We describe the method and present an implementation that extends BMotionWeb.

The difference between the approach presented in this chapter and the state-based approach (see Chapter 3) is imposed by the specifics of the CSPM formal language. The basic idea of the state-based approach is to visualize the information that is encoded in the states of a state-based formal specification (e.g. the values of variables), where each state of the formal specification is mapped to a particular visualization. In contrast to this, in CSPM the states of the specified system are left uninterpreted and the behavior is defined in terms of sequences of events (*traces*). Thus, the concept of the state-based approach is not longer applicable for event-based formalisms such as CSPM. The intention of our approach is to visualize the traces of the underlying CSPM specification.

In order to demonstrate our approach, we have created interactive formal prototypes for various CSPM specifications that we have found in the literature. In this chapter, we focus on the presentation of the interactive formal prototype of the bully algorithm [Ros10] and of a level crossing gate [RHB97]. We also discuss how our approach can be of use in the process of analyzing and validating CSPM specifications.

This chapter is organized as follows: In a first step, we describe the general approach for creating the gluing code of a CSP interactive formal prototype (Section 6.2). In

[1]This chapter presents a joint work with Ivaylo Dobrikov and Michael Leuschel and is described in [LDL14].

Section 6.3, we present the implementation of the approach. In Section 6.4, we demonstrate the application of the approach based on two case studies: the bully algorithm (Section 6.4.1) and the level crossing gate (Section 6.4.2). Finally, Section 6.5 extends the trace diagram approach presented in Section 5.2 to CSPM.

6.2. General Approach

The mathematical semantics of CSP are mainly based on *traces*. A trace is a sequence of events performed by a process that can communicate and interact with other processes within the CSP specification (see also Definition 3 in Section 5.2). The basic idea of our approach is to compute the representation of graphical elements based on the information encoded in the given sequence of events (*trace*). However, a process may perform many different traces and thus creating a graphical representation manually for each possible trace is an almost impossible task. To overcome this challenge, we present a method based on observers. Formally, one can describe the method by means of Algorithm 4.

Algorithm 4: Compute the representation of a graphical element for a given trace

```
1  function computeEventBased(tr $\langle q_0 e_1 \ldots e_n q_n \rangle$, graphicalElement $elem$)
2      $obs := collectObservers(elem)$
3      for $i$=1 to $n$ do
4          foreach $o \in obs$ do
5              if $member(e_i, o.expression)$ then
6                  $o.update()$
7              end if
8          end foreach
9      end for
10     return $elem$
11 end function
```

For creating the representation of a graphical element *elem* and a particular trace $tr = \langle q_0 e_1 \ldots e_n q_n \rangle$, we sequentially go through each event e_i of *tr* with $i \in \{1..n\}$ and execute all observers *obs* of *elem* for e_i. Note that by "creating a representation of a graphical element for a particular trace" we mean the representation of the state reached after the sequential execution of the events of a trace.

Each observer o has a user-defined CSP expression $o.expression$ that constitutes a set of observed events. For instance, the CSP expression $\{e.x \mid x \leftarrow \{0..3\}\}$ will constitute the set of observed events $\{e.0, e.1, e.2, e.3\}$. The graphical element *elem* is only updated when the currently processed event e_i of the given trace *tr* is a member of the respective set of observed events defined by $o.expression$. More precisely, *elem* is updated (line 6) whenever the expression $member(e_i, o.expression)$ evaluates to *true* (line 5). Finally,

Table 6.1.: Available options for CSP event observer

Name	Type	Required	Description
selector	string	yes	The *selector* matches a set of graphical elements which should be linked to the interactive handler.
observers	list	yes	A list of *observers*.
expression	string	yes	A user-defined CSP *expression* that constitutes a set of observed events.
actions	list	yes	A list of *actions* that determine the appearance and the behaviour of the graphical elements.
selector	string	yes	The *selector* that matches a set of element which should be modified (the elements are selected originated from the “main” selector option of the observer).
attribute	string	yes	The *attribute* of the elements that should be modified.
value	string	yes	The new *value* of the attribute.

the adapted graphical element *elem* (after processing all events of *tr*) is returned (line 10).

6.3. CSP Event Observer

We have implemented the method presented in Section 6.2 as an extension for BMotionWeb. In order to obtain trace information (e.g. event names and arguments) of a process and to evaluate expressions for a given CSPM specification, we use the ProB animator which is capable of executing CSPM specifications [LF08].

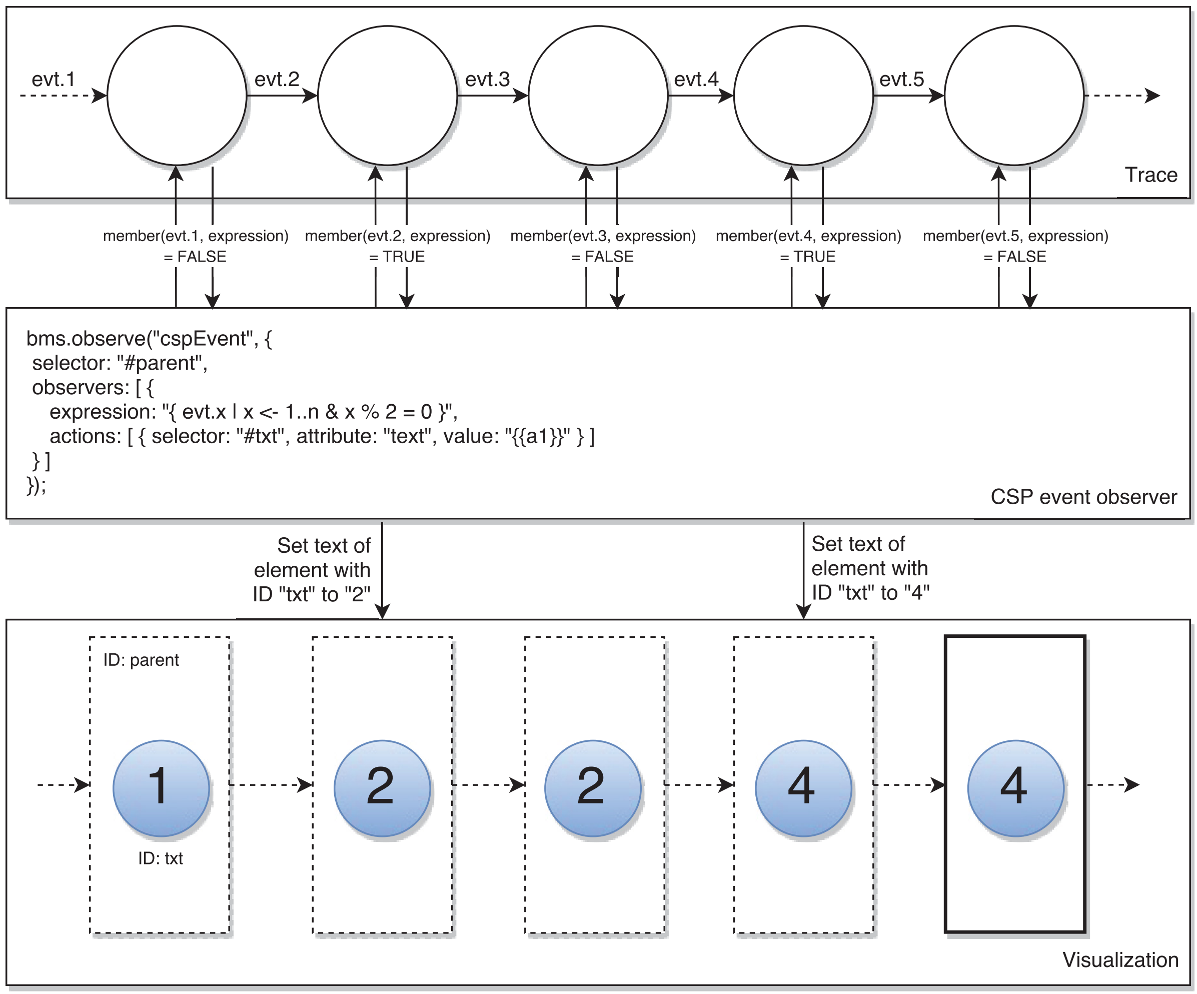
The extension contributes a new observer called *CSP event observer*. Table 6.1 shows the available options for the CSP event observer. The observer is attached to a set of graphical elements that matches a *selector* (similar to the presented state-based observers). Moreover, it defines a list of *observers*, where each observer entry has a *user-defined CSP expression* and a list of *actions*. The user-defined expression constitutes the set of observed events according to the method presented in Section 6.2. An action defines the *selector* that matches the elements which should be modified. Note that the observer tries to match the elements originating from the graphical element that match the “main” *selector* option of the observer. An action also defines an *attribute* (e.g. “fill” for coloring the interior of a graphical element like a circle shape) and a corresponding *value* that will be set as the new value of the attribute when the action is triggered. As stated in Section 6.2, the actions are triggered when the currently processed event is in the set of observed events. Note that the ordering of the observer entries is important since they are triggered sequentially. Thus, the actions of an observer entry may override the actions of a previous observer entry.

The user can also refer to the information given by the arguments of the currently processed event within the action fields (*selector*, *attribute* and *value*). This is achieved with the pattern "{{aN}}" where aN refers to the N-th argument of the event. For instance, if the event has two arguments, then the first and the second one can be obtained with "{{a1}}" and "{{a2}}", respectively. To illustrate the use of the pattern, consider an event $evt.x$ with $x \leftarrow 0..4$. One may want to use the information given by the first argument x of evt within the *selector* option in order to select graphical elements that have an ID of the form "elem$x$". To do this, we can define the *selector* option as "#elem{{a1}}". The construct "{{a1}}" will be replaced by the value of the first argument of the currently processed event. For instance, if the currently processed event is $evt.2$, the value of the *selector* option "#elem{{a1}}" will become "#elem2".

```
bms.observe("cspEvent", {
  selector: "#parent",
  observers: [
    {
      expression: "{ evt.x | x <- 1..n & x mod 2 = 0 }",
      actions: [
        { selector : "#txt", attribute : "text", value : "{{a1}}" }
      ]
    }
  ]
});
```

Listing 6.13: Example CSP event observer (JS)

Figure 6.1 illustrates the use of the CSP event observer shown in Listing 6.13 for the trace $tr = \langle evt.1, evt.2, \ldots, evt.5\rangle$ and the visualization shown in the lower side of the figure. The visualization consists of an SVG graphic with a text field element that has the ID "txt". The CSP event observer defines an expression that constitutes the set of observed events $evt = \{evt.2, evt.4, evt.6, ..\}$ and one action $act1$ that changes the value of the text attribute of the graphical element with the ID "txt" (the text field) to "{{a1}}". According to our approach (see Section 6.2), the observer is executed for each event of the trace. This means that whenever the currently processed event is in the set of observed events evt, the observer will trigger the defined action $act1$. Considering trace tr, the execution of the event $evt.2$ causes the observer to set the value of the text field element to "2" and the execution of $evt.4$ causes the observer to set the value of the text field element to "4" as demonstrated in Fig. 6.1. Here, the value "{{a1}}" is replaced with the first argument of the respective event. The final visualization after completing the last event of the trace ($evt.5$) is shown in the last box marked with a solid border.

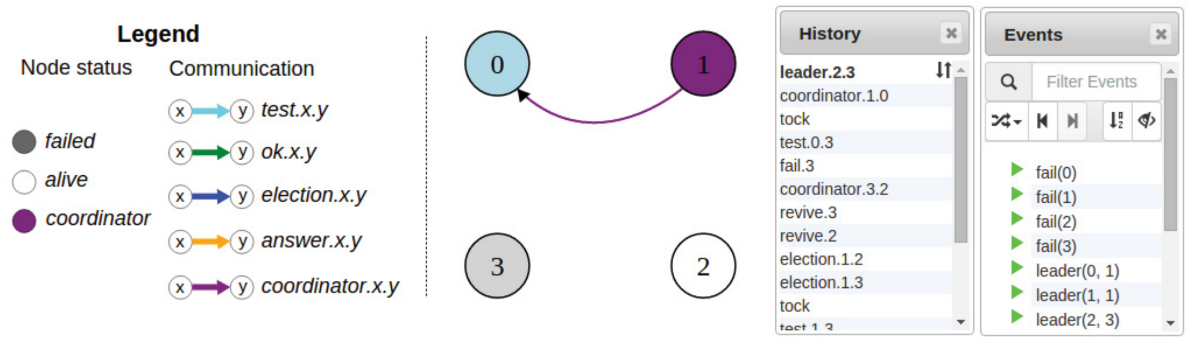

evt.1
evt.2
evt.3
evt.4
evt.5
Trace
member(evt.1, expression) = FALSE
member(evt.2, expression) = TRUE
member(evt.3, expression) = FALSE
member(evt.4, expression) = TRUE
member(evt.5, expression) = FALSE
CSP event observer
Set text of element with ID "txt" to "2"
Set text of element with ID "txt" to "4"
ID: parent
ID: txt
1
2
2
4
4
Visualization

```
bms.observe("cspEvent", {
 selector: "#parent",
 observers: [ {
   expression: "{ evt.x | x <- 1..n & x % 2 = 0 }",
   actions: [ { selector: "#txt", attribute: "text", value: "{{a1}}" } ]
 } ]
});
```

Figure 6.1.: Illustration of the CSP event observer for a given trace

6.4. Case Studies

In order to test our approach, we have successfully created various interactive formal prototypes for CSPM specifications that we have found in the literature. In this section we present the interactive formal prototype of the bully algorithm specification from [Ros10] and of the level crossing gate specification from [RHB97]. The specifications have not been modified for the interactive formal prototype we have created. Both interactive formal prototypes were created using the built-in visual editor of BMotionWeb (see Section 4.10). However, for presentation purposes the observers of the interactive formal prototypes are described in the JavaScript notation in this section.



History:

leader.2.3
coordinator.1.0
tock
test.0.3
fail.3
coordinator.3.2
revive.3
revive.2
election.1.2
election.1.3
tock

Events:

fail(0)
fail(1)
fail(2)
fail(3)
leader(0, 1)
leader(1, 1)
leader(2, 3)

Figure 6.2.: Interactive formal prototype of the Bully Algorithm

6.4.1. The Bully Algorithm

The algorithm represents a method of distributed computing for electing a node to be the coordinator amongst a group of nodes. Each node has a unique ID and the algorithm intends to select the node with the highest ID to be the coordinator. It is assumed that the nodes may fail and revive from time to time and the communication between the nodes is reliable. Three types of messages are defined within the design of the algorithm: *election* (announcing an election), *answer* (responding to an election message), and *coordinator* (announcing the identity of the coordinator).

The specification from [Ros10] defines six additional types of events needed for the formalisation of the algorithm in CSP: the *fail* and *revive* events (for modeling the failing and reviving of a node), the *test* and *ok* events (for simulating a test-response communication), the *leader* events (for indicating the coordinator of a living node), and the *tock* event (for modelling timeouts and time).

Visualizing the Bully Algorithm. In general, we want to visualize the process of electing a coordinator in the network. More precisely, we aim to visualize the *Network* process of the CSP specification. As the bully algorithm specification in [Ros10] is presented for a network with four nodes, we have created a visualization for four nodes (the nodes are enumerated from 0 to 3). Figure 6.2 demonstrates the visualization of a particular trace.

There are two major aspects of the specification that we want to visualize: the nodes and the communication between the nodes. Each node is visualized by means of a circle which contains the respective ID of the node. The communication between the nodes is illustrated with directed arrows.

For each graphical element in the visualization, we assign a unique ID referring to the elements in the CSP specification. Thus, the node with ID x in the CSP specification is presented by the circle with ID “n-x” in the visualization. Additionally, a message transfer from the node with ID x to the node with ID y is represented by the line with ID "l-x-y" and the arrowhead with ID "p-y" (i.e. the arrow connecting "n-x" and "n-y"). In this section, both symbols x and y stand for an integer ranging from 0 to 3.

We can classify all types of events in the specification into the following groups:

- **status:** Events that can change the status of a particular node x: $fail.x$, $revive.x$ and $coordinator.x.y$.
- **message:** Events illustrating a message transfer from node x to node y: $test.x.y$, $ok.x.y$, $election.x.y$, $answer.x.y$, and $coordinator.x.y$.
- **hidden:** Events that are not considered in the visualization: *tock* and $leader.x.y$.

Thus, we can infer that there are two general types of observers to define: the *status* and the *message* observers. Note that the *coordinator* event ($coordinator.x.y$) has been included in the first two groups above. This is because in the specification each of the *coordinator* events intends to identify the coordinator (x) and at the same time represents a message transfer (to node y).

The status of a node usually changes when one of the *status* events has been executed. Each node, apart from the node with the lowest ID[2], can have the following status: `failed`, `alive` or `coordinator`. A unique fill color has been selected for distinguishing each possible status of a node (see legend in Fig. 6.2).

CSP Event Observer of Bully Algorithm. We will now demonstrate the creation of the CSP event observer for the bully algorithm. To do this, we will explain the different observer entries of the observer (expression and actions). The CSP event observer is attached to the root element of the visualization that matches the selector #bully. The action elements are selected based on the action selectors.

In order to associate a *status* event from the CSPM specification with a node in the visualization, we use the selector "#n-{{a1}}" in the definition of the respective observer entry. The construct "{{a1}}" is used for obtaining the value of the first argument of the respective *status* event. For example, the observer entry for changing a status of a node to `failed` can be defined as follows:

```
{
  expression: "{fail.x | x <- {0..N-1}}",
  actions: [ { selector: "#n-{{a1}}", attribute: "fill", value: "lightgray"} ]
}
```

The observer entry will color the respective node lightgray whenever a *fail* event has been processed. For instance, the node with ID "n-3" will be colored lightgray when the event *fail*.3 has been processed. In a similar fashion, we have defined the observer entries for the other node status changes.

[2]The node with ID 0 can never be a coordinator as there is no node with a lower ID.

For creating the *message* observer entries we need to consider both arguments of the *message* events. The types of the messages are distinguished by different stroke patterns (see legend in Fig. 6.2). Thus, each *message* observer entry has two actions except for the *coordinator* observer entry which has three: one action causes the directed arrow (consisting of the line and arrowhead) to appear and one action changes the stroke pattern of the arrow. For instance, the observer entry for visualizing the election message can be defined as follows:

```
{
  expression: "{election.x.y | x <- {0..N-1}, y <- {0..N-1}}",
  actions: [ { selector: "#l-{{a1}}-{{a2}}, #p-{{a2}}",
               attribute: "class", value: "visible" },
             { selector: "#l-{{a1}}-{{a2}}",
               attribute: "stroke", value: "blue" } ]
}
```

To provide a good visualization an additional observer entry has been added to hide all arrows after performing an arbitrary event (expect for the *leader* events and the *tock* event[3]). This observer entry is applied on the currently processed event before all other defined entries, i.e. it is the first entry in the observer list.

The initial state of the specification and the visualization is the state in the network where all nodes are alive and the coordinator is the node with the ID 3 (the node with the greatest ID). No message exchanges are performed.

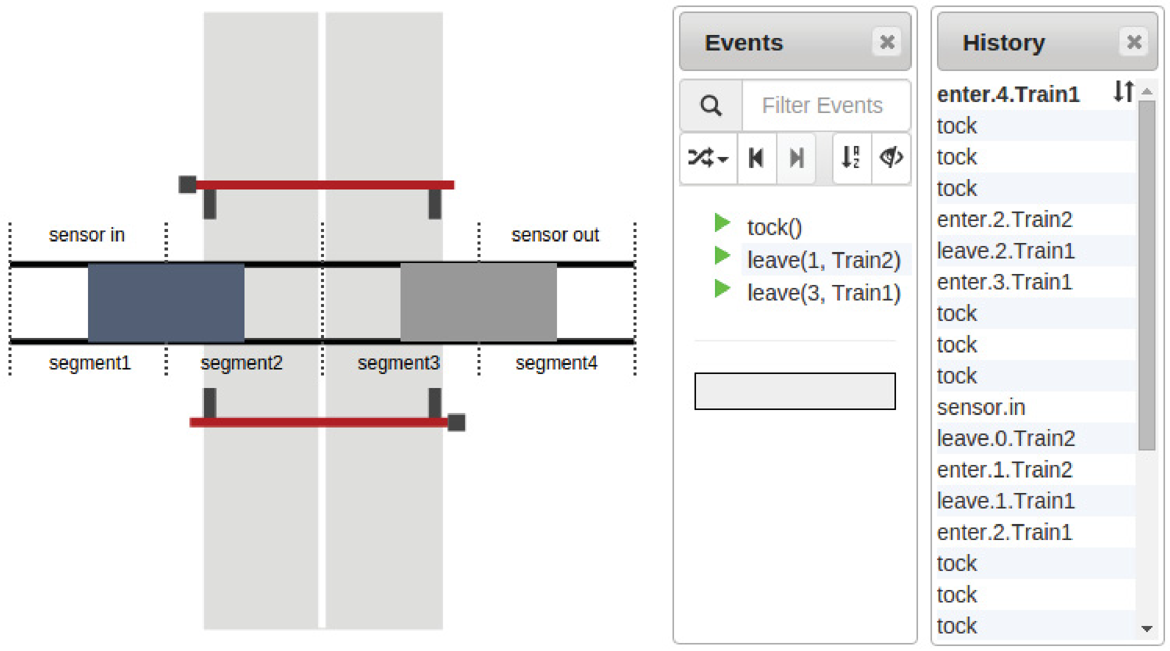
6.4.2. Level Crossing Gate

The first case study introduced in [RHB97] specifies a level crossing gate of a single railway track along which trains move only in one direction. The track is divided into segments such that each of the segments is at least as long as any train. There are five track segments considered for the level crossing gate where one of the track segments represents the outside world.

The track segments are numbered. The input sensor is placed in segment 1 and the crossing and output sensors are in segment 4. The outside world segment is identified with 0. A train enters segment $(i + 1)$ before it leaves segment i. Entering and leaving of a segment are specified by the events *enter* and *leave*, respectively. The entering of train t into segment j is described by $enter.j.t$. Accordingly, the leaving of train t from segment j is described with the event $leave.j.t$.

The sensors send control signals to the gate. The gate goes down after a train enters segment 1, and the gate goes up after the train leaves segment 3 and no train is moving along the segments 1 or 2. The control signals sent by the input and output sensors

[3]Since the *leader* events and the *tock* event are not considered in the visualization, they do not change its state.



tock()
leave(1, Train2)
leave(3, Train1)

enter.4.Train1
tock
tock
tock
enter.2.Train2
leave.2.Train1
enter.3.Train1
tock
tock
tock
sensor.in
leave.0.Train2
enter.1.Train2
leave.1.Train1
enter.2.Train1
tock
tock
tock

Figure 6.3.: Interactive formal prototype of the the level crossing gate

are specified by the events *sensor.in* and *sensor.out*, respectively. The communication between the controller and the gate processes is specified by the channel *gate* which defines four different events. The events *gate.go_down* and *gate.go_up* represent the commands from the controller to the gate for moving the barriers down or up. The events *gate.down* and *gate.up* denote the confirmations from the gate sensors that the barriers are down or up, respectively.

In addition, timing constraints are set for the trains moving on the tracks. The speed of each train is determined by how many units of time a train can spend per track segment. This additional property is required since the goal of the system is to guarantee via timing that the gate is up and down at appropriate moments. In the CSPM specification the speed of a train per track segment has been set to three time units. A unit of time is denoted by the *tock* event in the level crossing gate specification.

Visualizing the Level Crossing Gate. In our interactive formal prototype (see Fig. 6.3) we assume that the trains are moving from left to right. Track segments 1 to 4 are illustrated by rectangles separated by vertical, dotted lines. Segment 0, which represents the outside world, can be seen as the space to the left of track segment 1 and the space to the right of segment 4. A train leaves the outside world after entering track segment 1, and a train enters the outside world after leaving track segment 4. The length of each of the track segments 1 to 4 in the visualization is considered to be 100 pixels.

Since the specification from [RHB97] handles two trains, we only visualize these two trains (*Train*1 and *Train*2). The trains are represented by two boxes colored gray and slate gray, respectively. The movement of a train along the track is simulated by shifting the respective box from left to right. In order to simulate a movement between the track segments, we shift the respective box 50 pixels from the left to right. In doing so, the entrance into a new segment is represented such that the box is half on the new segment and half on the previous. On the other hand, when the train leaves a track segment, the box is moved fully onto the recently entered segment. In Fig. 6.3, the gray box representing $Train1$ is half on segment 4 and half on segment 3 after executing the event $enter.4.Train1$, whereas $Train2$ (the slate gray box) has moved fully onto segment 1 after performing the events $enter.1.Train2$ and $leave.0.Train2$ consecutively. Each box representing a train is also 100 pixels in length.

For visualizing the movement of the trains, we defined two observer entries that listen to the events $enter.j.t$ and $leave.j.t$. Both observer entries contain an action that changes the *transform* attribute [W3C11] of the matched graphical element. For instance, the *leave* observer entry is defined such that by executing an event $leave.j.t$ the graphical element with the ID "train-t" (t refers to the second argument of the *leave* events) will be moved 50 pixels to right by setting the *transform* attribute to the value $translate(50, 0)$. Thus, the observer entry for leaving a track is defined as follows:

```
{
  expression: "{leave.j.t | j <- {0..3}, t <- {Train1,Train2}}",
  actions: [ { selector: "#train-{{a2}}", attribute: "transform",
               value: "translate(50,0)" } ]
}
```

Note that the *leave* observer entry does not fire its actions when an event $leave.4.t$ is executed since in our visualization the respective box "train-t" needs to be moved to the left site of track segment 1 when the event $enter.0.t$ is executed. We decided to define the observer entries in this way because after entering the outside world (track segment 0) and leaving the last track segment 4, the same train can enter the crossing gate segments once again.

For the overall visualization we have defined five different observer entries. The other three observer entries are responsible for simulating the up and down movement of the barriers in the visualization after proceeding of the events $gate.up$ and $gate.down$, respectively. To do this, we created two graphical elements that illustrate the two possible states of the appropriate barrier: barrier is up and barrier is down. This means that we have four graphical elements illustrating the different positions of the barriers. When, for example, the event $gate.down$ is processed, the $go-down$ observer entry executes two actions. The first is to hide all barrier elements, and the second action is to display the graphical elements representing that the barriers are down. The hiding and displaying of the barriers is realized by setting the "opacity" attribute of the graphical elements to 0 and 100, respectively. The $go - down$ observer entry is defined as follows:

```
{
  expression: "{gate.down}",
  actions: [
    { selector: "g[id^=gate]", attribute: "opacity", value: "0" }
    { selector: "g[id^=gate-go_down]", attribute: "opacity", value: "100" }
  ]
}
```

We defined the $go - up$ observer entry in the same way. The initial state of the specification and its visualization is the state in which both trains are in the "outside world" track segment and both barriers are up.

6.5. Combined Trace Diagram for CSPM

Since the information to be visualized in stated-based formal methods and event-based formal methods like CSPM differs, we need to distinguish between them. Algorithm 5 shows the adapted algorithm for creating a trace diagram for event-based formal methods. The algorithm is very similar to Algorithm 3 presented in Section 5.2. The only difference is the computation of the representation of the graphical elements for a state: for event-based formal methods we assign the computed representation of a graphical element using the algorithm *computeEventBased* (see Algorithm 4) passing a segment of the trace *tr* that starts at $q_0 e_1$ and ends at the currently processed state $e_{i+1} q_{i+1}$ and the graphical element *ele* (line 7).

Algorithm 5: Create event based trace diagram combined with graphical elements

```
1  function createEBTraceDiagram(tr $\langle q_0 e_1 \ldots e_n q_n \rangle$, graphicalElements $elems$)
2      $nodes := \varnothing$
3      $edges := \varnothing$
4      for $i=0$ to $n$ do
5          $edges := edges \cup \{q_i e_{i+1} q_{i+1}\}$
6          foreach $ele \in elems$ do
7              $nodes(q_i) := nodes(q_i) \cup computeEventBased(\langle q_0 e_1 \ldots e_{i+1} q_{i+1} \rangle, ele)$
8          end foreach
9      end for
10     $buildDiagram(nodes, edges)$
11 end function
```

Using validation tools for performing various consistency checks of formal specifications automatically is a powerful technique for verifying the correctness of the analyzed specification. The failure of a consistency check is mostly reported by producing of a *counter-example* (very often presented as a trace leading to an error state). However, trying to understand the failure behavior of the model by simply examining the trace can sometimes be difficult as the error trace may, for example, be the result of the interaction of various components in the specified system. Thus, using a visualization in order to facilitate the effort of understanding the error trace can be very useful.

We will now show how the trace diagram feature may, for example, contribute to the better understanding of an erroneous behavior in the CSPM specifications.

For example, the trace of the *Network* process of the bully algorithm specification

$$\langle \mathit{fail}.2, \mathit{fail}.3, \mathit{test}.1.3, \mathit{tock}, \mathit{election}.1.3, \mathit{election}.1.2, \mathit{revive}.2, \mathit{revive}.3, \\ \mathit{coordinator}.3.2, \mathit{fail}.3, \mathit{test}.0.3, \mathit{tock}, \mathit{coordinator}.1.0, \mathit{leader}.2.3\rangle$$

represents a sequence of events leading to a state in the network in which the elected leader is not the living node with the greatest ID. In general, the false behaviour that is explicitly discussed in [Ros10] illustrates a problem that occurs with a certain combination of node failures and mixed up elections.

When examining the above error trace, it is hard for the user to reproduce and to see the actual problem. In contrast, Fig. 6.4 shows the trace diagram of the error trace. The user can see at a glance the erroneous behaviour that is shown in the last state of the trace diagram (after performing *leader*.2.3).

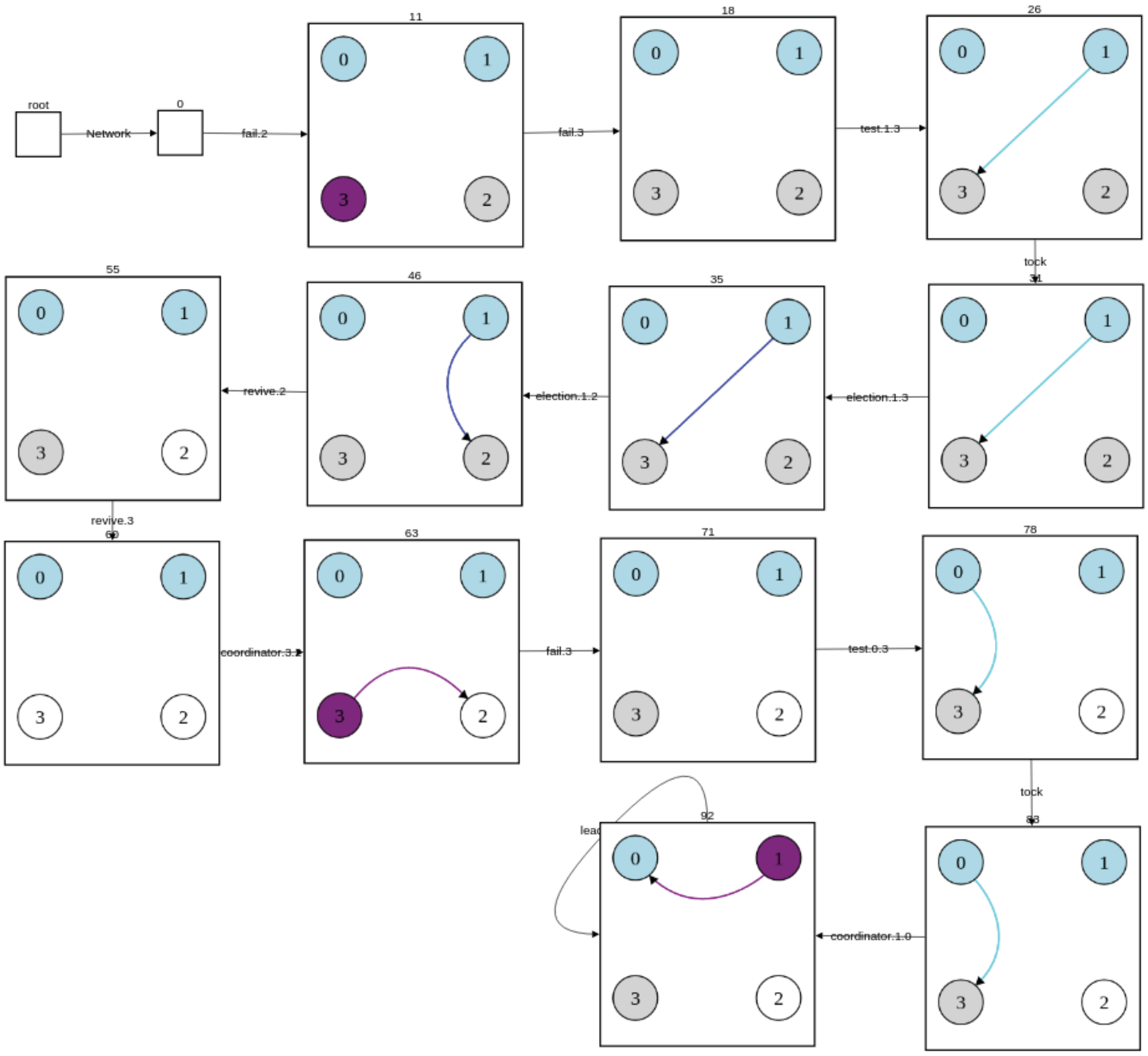


Figure 6.4.: Trace diagram for an erroneous behaviour in the bully algorithm specification

7 Applications of BMotionWeb

7.1. Introduction

In this chapter we demonstrate various applications of BMotionWeb. Section 7.2 describes the use of BMotionWeb for the lightweight validation of *interactive* safety critical systems based on a case study of a phonebook software user interface and a cruise control system device. In Section 7.3 we present two industrial case studies: a landing gear system and a hemodialysis machine.[1] Beside these two industrial case studies, we also give some brief success stories of applying BMotionWeb in the railway and smart energy domain (see Section 7.4 for more information). Finally, we describe the application of BMotionWeb for teaching formal methods in Section 7.5. Since BMotionWeb is the successor of BMotionStudio, we also cover success stories of applying BMotionStudio, as well as consider the use of BMotionStudio for teaching formal methods. Appendix B.1 gives a comprehensive list of existing case studies for BMotionWeb and BMotionStudio.

7.2. BMotionWeb for Interactive Systems

Nowadays, safety-critical systems, such as medical devices, airplane cockpits, or railway- and nuclear plant control systems, typically include interactive user interfaces (UI). Thus, the development of safety-critical systems is also required to properly account for user's cognition and to ensure the *usability* of the system. This task is typically performed by UI engineers. However, UI engineers are rarely trained in formal methods. On the other hand, formal engineers typically have no experience in common UI engineering techniques. Consequently barriers can arise when working in a multidisciplinary team which can compromise the success of the project.

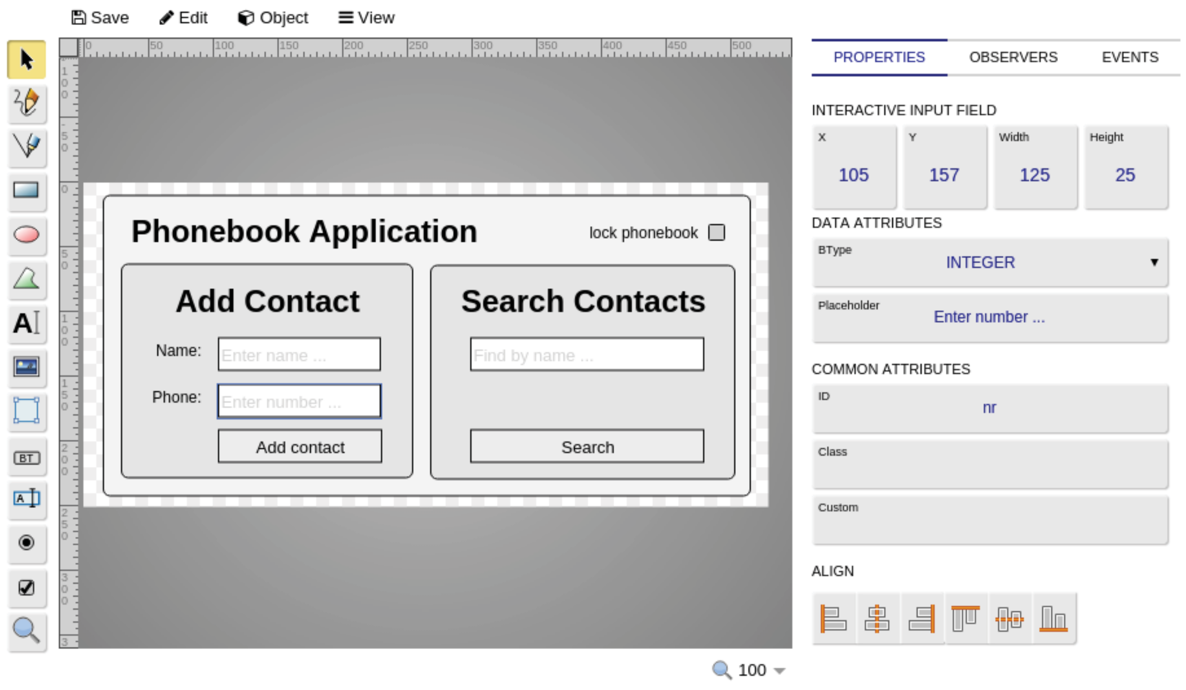
In this section, we demonstrate the use of BMotionWeb to support the validation of *interactive* systems, i.e. a system where a human interacts with the system, for instance, via a graphical user interface or device. We demonstrate two interactive systems: the

[1] The case studies are available in the GitHub BMotionWeb case study repository (see Appendix B.1).

UI of a simple phonebook software (Section 7.2.1) and the device for a cruise control system (Section 7.2.2).

7.2.1. Phonebook Software User Interface

The first interactive system considered in this section is a phonebook software UI specified in classical-B. The phonebook allows users to manage persons and telephone numbers and provides the following functionalities: adding and deleting persons with an associated number, searching for numbers, and activating or deactivating persons. Moreover, the user can lock the phonebook which means that the user can not add new entries or delete existing entries. The aim of this case study is to exemplify the creation of software UI mockups as well as to demonstrate how the gluing code between a software UI mockup and an animated formal specification can be established with BMotionWeb.



INTERACTIVE INPUT FIELD

X	Y	Width	Height
105	157	125	25

DATA ATTRIBUTES

BType: INTEGER

Placeholder: Enter number ...

COMMON ATTRIBUTES

ID: nr

Class:

Custom:

Figure 7.1.: UI mockup of the phonebook software in visual editor of BMotionWeb

Mockup software user interfaces. Figure 7.1 shows a snapshot of the visual editor in BMotionWeb while creating the mockup of the phonebook software UI. As can be seen in the figure, the UI is composed of different graphical elements like shapes and labels, as well as of *interactive* graphical elements like input fields, buttons and a checkbox. For instance, the UI provides two input fields for entering the name and the phone number of the person to be added to the database. This input (e.g. the name and phone number entered) can be used for defining interactive handlers. As an example, the "Add contact" button (“#btAdd”) shown in Fig. 7.2 is linked to the execute event handler defined in Listing 7.14. The handler executes the *add* operation in the classical-B specification of the phonebook software shown in Listing 7.15. In lines 4 to 9 we define a JavaScript function that returns a predicate determining the parameters of the *add* operation. The returned predicate (lines 7 to 8) is composed of the values of the name and number input fields (line 5 and 6). The result can be seen in Fig. 7.2 where the user hovers over the “Add contact” button in the interactive formal prototype of the phonebook software (see upper left side). Based on this principle we can execute operations or events in a formal specification based on user inputs and thus allow non-formal method experts such as UI engineers to work with the formal specification via a custom UI.

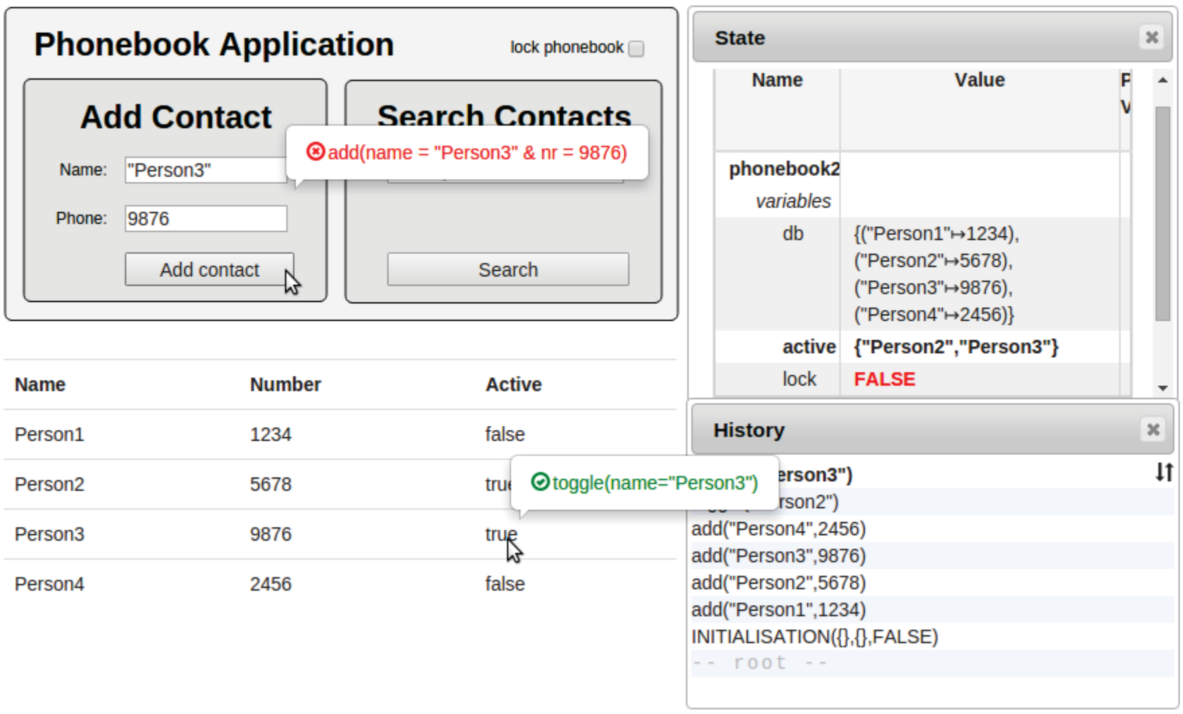
```
1 bms.executeEvent({
2  selector: "#btAdd",
3  name: "add",
4  predicate: function(ui) {
5   var name = ui.find("#name");
6   var nr = ui.find("#nr");
7   return "name=" + name.val() +
8           "& nr=" + nr.val();
9 }});
```

Listing 7.14: “Add contact” button event handler (JS)

```
1 add(name, nr) =
2  PRE
3   name : STRING &
4   name /: dom(db) &
5   nr : NATURAL &
6   lock = FALSE
7  THEN
8   db := db \/ {name |-> nr}
9 END;
```

Listing 7.15: Phonebook *add* Operation (classical-B)

We can also make use of observers to determine the appearance of the UI based on state variables. Consider the formula observer in Listing 7.16 which observes the *lock* variable stating the database is locked (the user can not add new entries or remove existing entries, i.e. $lock = true$) or unlocked (all UI functions are made available to the user, i.e. $lock = false$). The goal of the observer is to deactivate the input fields based on the lock checkbox graphical element (see upper right side of Fig. 7.2). To do this, the observer is registered for the name and number input fields (#name and #nr) and for the “Add contact” button (#btAdd). In lines 5 to 7 of Listing 7.16 we define a trigger function which sets the *disabled* property of the graphical elements to the value of the lock state variable ($values[0]$) causing a deactivation of the input fields and the button whenever the *lock* variable is set to true (the input fields and the button are unusable and unclickable). Otherwise, the input fields and the button are activated (*lock* is set to false). Note that we have set the *translate* property of the formula observer to *true* (see line 4). This enables us to use the state value of the *lock* variable directly in the context of the *prop* function (see line 6).



Name: "Person3"

Phone: 9876

add(name = "Person3" & nr = 9876)

toggle(name="Person3")

Name	Number	Active
Person1	1234	false
Person2	5678	true
Person3	9876	true
Person4	2456	false

Name	Value
phonebook2	
variables	
db	{("Person1"↦1234), ("Person2"↦5678), ("Person3"↦9876), ("Person4"↦2456)}
active	{"Person2","Person3"}
lock	FALSE

```
erson3")
rson2")
add("Person4",2456)
add("Person3",9876)
add("Person2",5678)
add("Person1",1234)
INITIALISATION({},{},FALSE)
-- root --
```

Figure 7.2.: Interactive formal prototype of phonebook software user interface

```
bms.observe("formula", {
 selector: "#name, #nr, #btAdd",
 formulas: ["lock"],
 translate: true,
 trigger: function(origin, values) {
  origin.prop("disabled", values[0]);
 }
});
```

Listing 7.16: Formula observer for lock variable (JS)

Visualize dynamic data-structures. In formal specification languages like classical-B, the software is often specified with data-structures like sets and relations. These data-structures typically contain a *dynamic* number of elements. For instance, the database of the phonebook is specified as a relation between persons and numbers, where the number of database entries increases or decreases whenever the user adds or removes a person. In this section, we demonstrate the use of the JavaScript MVC (Model View Controller) framework AngularJS [GS13][2] to connect HTML elements like tables and lists with dynamic data-structures like sets or relations.

[2]We choose AngularJS because BMotionWeb is also based on AngularJS. However, we could also use other JavaScript MVC libraries as well.

AngularJS provides *controllers* and *directives*. A controller defines the data and behavior of HTML templates and a directive can attach a specified behavior to an existing HTML element. Consider the controller *phonebookController* in Listing 7.18. In lines 4 to 10 we register a new formula observer which observers the two state variables *db* and *active* in the phonebook application. The values of the variables are assigned to the *scope* of the controller and updated whenever a state change occurred in the animated formal specification (line 8 and 9). A scope can be made available to an HTML template using the *ngController* directive as demonstrated in line 1 in Listing 7.17. Once the scope has been attached to the template, the values of the two variables *db* and *active* can be used within the template. In line 7 we assign an *ngRepeat* directive which creates a table row (lines 7 to 13) in each element in the *db* relation. Each row gets its own scope, where the current element (*el*) of the *db* relation is set to the row's scope. Thus, we can access the name ({{*el[0]*}}), the number ({{*el[1]*}}) and the status (activated or deactivated) of each element ({{*isActive(el[0]*}})) and show them in the respective columns of the row (lines 8 to 12). The *isActive* function is defined in lines 12 to 14 and takes a person as its parameter. It returns true whenever the person is in the *active* set. Otherwise, it returns false.

We can also define custom directives as demonstrated in lines 17 to 30 of Listing 7.18, where we define a new directive called *executeEvent*. The purpose of the directive is to attach an execute event handler to an HTML element in context of the defined *name* and *predicate* attributes of the HTML element (line 24 and 25 respectively). We have assigned this directive to each "Active" column of the HTML table as demonstrated in line 10 of Listing 7.17. For each row, the directive creates a new execute event handler with *toggle* as the operation's name and *name="*{{*el[0]*}}*"* as the operation's predicate.

The lower left side of Fig. 7.2 shows the HTML table during the simulation of the interactive formal prototype. Based on the *db* variable, the rows of the table and the execute event handlers for the "Active" columns are created dynamically. As can be seen in the figure, the table contains four rows representing the four entries of the *db* variable. If we add a new entry or remove an existing entry from the *db* variable, the table will adapt and reflect the new state dynamically.

```html
<table ng-controller="phonebookController">
 <tr>
  <th>Name</th>
  <th>Number</th>
  <th>Active</th>
 </tr>
 <tr ng-repeat="el in db">
  <td>{{el[0]}}</td>
  <td>{{el[1]}}</td>
  <td execute-event name="toggle" predicate='name="{{el[0]}}"'>
   {{isActive(el[0])}}
  </td>
 </tr>
</table>
```

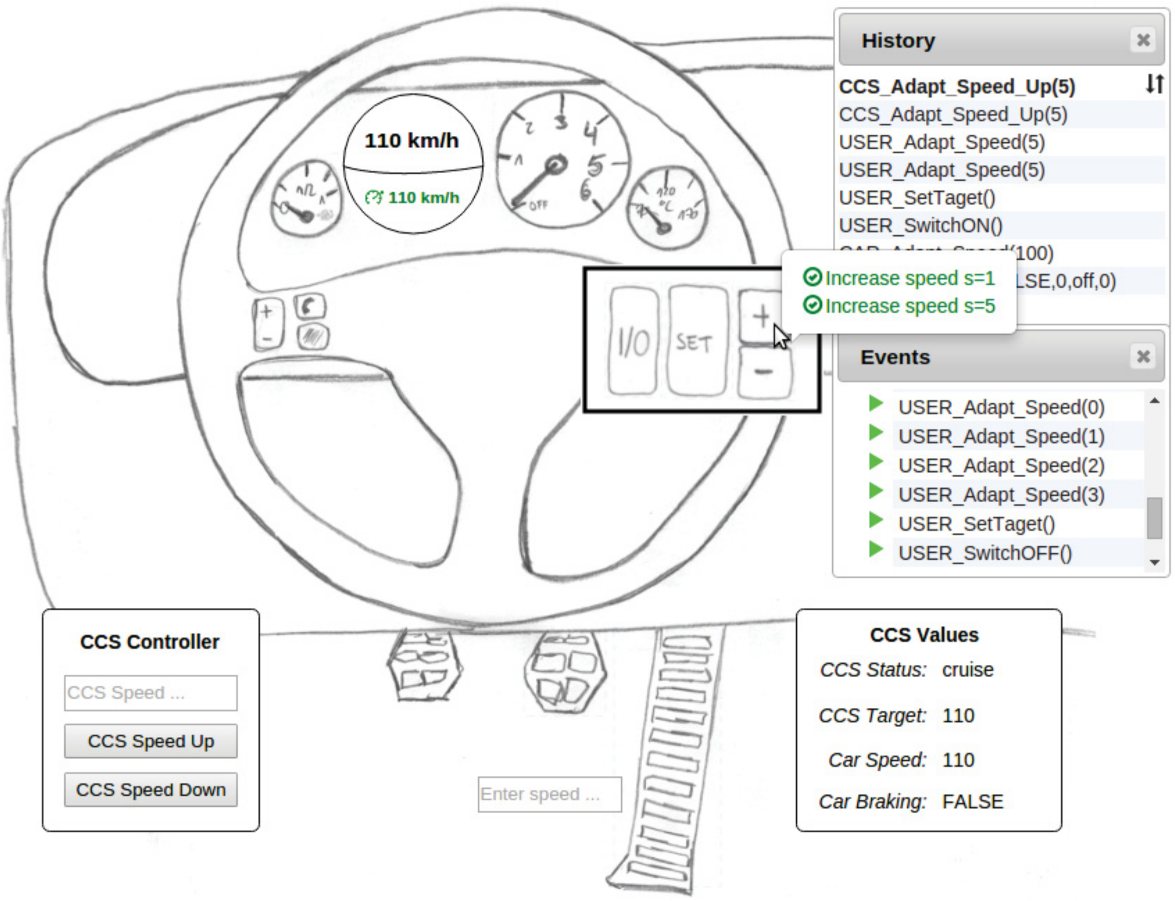
Listing 7.17: Template for database table (HTML)

```js
angular.module("phonebook", [])
.controller("phonebookController", function() {

 bms.observe("formula", {
  formulas: ["db", "active"],
  translate: true,
  trigger: function(values) {
   $scope.db = values[0];
   $scope.act = values[1];
 }});

 $scope.isActive = function(n) {
  return $scope.act.indexOf(n) > -1;
 }

})
.directive("executeEvent", function() {
 return {
  replace: false,
  link: function($scope, $element, attr) {

   bms.handler("executeEvent", {
    element: $element,
    name: attr["name"],
    predicate: attr["predicate"]
   });

  }
 }
});
```

Listing 7.18: Controller for database table (JS)

7.2.2. Cruise Control Device

A common way to develop mockups is to create graphical sketches using the classical paper-and-pencil approach. In this section we demonstrate the application of BMotion-Web for reusing such graphical sketches for the creation of interactive formal prototypes. For this purpose, we use an Event-B specification of a cruise control system (CCS) and a graphical sketch of a car cockpit with a CCS device (see Fig. 7.3) as a case-study. A CCS is an automotive system implemented in software which automatically controls the speed of a car. The CCS device provides buttons to switch the CCS system on/off, to set the current speed of the car as the target speed of the CCS system, and to increase and decrease the target speed. In addition, a speedometer provides information about the state of the CCS system and about the target speed which depends on the current car speed.



History

CCS_Adapt_Speed_Up(5)
CCS_Adapt_Speed_Up(5)
USER_Adapt_Speed(5)
USER_Adapt_Speed(5)
USER_SetTaget()
USER_SwitchON()
(100)
LSE,0,off,0)

Events

USER_Adapt_Speed(0)
USER_Adapt_Speed(1)
USER_Adapt_Speed(2)
USER_Adapt_Speed(3)
USER_SetTaget()
USER_SwitchOFF()

CCS Values

CCS Status:	cruise
CCS Target:	110
Car Speed:	110
Car Braking:	FALSE

Figure 7.3.: Interactive formal prototype of cruise control device

Using the visual editor in BMotionWeb, UI engineers can select a picture (e.g. a graphical sketch or a photograph) of a device or a UI as a starting point for creating an interactive formal prototype. Once a picture is selected it can be extended with additional graphical elements. BMotionWeb makes it possible to add an *interactive area* graphical element which can be placed over the picture. An interactive area allows UI engineers to bind an execute event handler to a specific area in the picture. As an example, in Fig. 7.3 the user hovers over the "+" button (#btSpeedUp) on the graphical sketch of the CCS device. The interactive area overlays the button and setups an execute event handler (see Listing 7.19) that wires the event *USER_Adapt_Speed* shown in Listing 7.20 with the predicates *s=1* and *s=2*. Similar graphical elements wired with corresponding execute event handlers have been created for the other functions in the CCS device.

```
bms.executeEvent({
 selector: "#btSpeedUp",
 events: [{ name: "USER_Adapt_Speed",
          predicate: "s=1" },
         { name: "USER_Adapt_Speed",
          predicate: "s=5" }],
 label: function(evt) {
  return "Increase speed " +
           evt.predicate;
 }
});
```

Listing 7.19: Execute event handler "increase target speed" (JS)

```
event USER_Adapt_Speed
 any
  s
 where
  @g1 s : -maxspeed .. maxspeed
  @g2 ccs_status = cruise
  @g3 ccs_target + s <= maxspeed
  @g4 ccs_target + s >= 0
 then
  @a1 ccs_target := ccs_target + s
end
```

Listing 7.20: CCS "increase target speed" event (Event-B)

7.3. Industrial Application

In this section we demonstrate the application of BMotionWeb to the development of interactive formal prototypes for two industrial case studies: a landing gear system from the aviation domain (Section 7.3.1) and a hemodialysis machine from the medical domain (Section 7.3.2). For each case study, we give a brief overview and description of the case study and the formal specification (both case studies have been specified in Event-B) and present the development of the interactive formal prototype including the visualization and the gluing code (observers and event handlers). Finally, in Section 7.3.3, we demonstrate the application of the interactive formal prototypes to support the validation of the corresponding formal specifications. The purpose of both case studies is to describe in detail how to develop an interactive formal prototype and to show how BMotionWeb supports the validation of industrial case studies.

We have created the interactive formal prototypes based on the last refinement levels of the formal specifications. However, we claim that the development of an interactive formal prototype should happen in parallel to the development of the formal specification. Our approach for creating an interactive formal prototype is as follows: for each aspect of the system which is represented by a state variable in the Event-B specification, we

Table 7.1.: Overview of refinement levels of the Event-B landing gear system

Refinement	**Description**
R0_Gear	The top-level abstraction with one single abstract gear.
R1_GearDoor	Introduction of an abstract door.
R2_GearDoorHandle	Introduction of the pilot handle.
R3_GearsDoorsHandle	Triplication of gear and door (data refinement).
R4_...Valves	Introduction of hydraulic elements such as the valves.
R5_...Controller	Connecting electric orders in the digital part to the valves in the hydraulic part.
R6_...Switch	Introduction of an analog switch to prevent the hydraulic part from behaving abnormally.
R7_...Lights	Introduction of cockpit lights.
R8_...Sensors	Introduction of input sensors into the digital part.

have created a graphical representation and the gluing code incrementally. In order to verify that a graphical element represents all possible states of the system properly, we have generated a projection diagram for each graphical element illustrating all possible states of the element in the system.

7.3.1. Landing Gear System

The first industrial case study is a *landing gear system*.[3] The case study has its origin in the aviation domain and was originally specified as a challenge for the ABZ'14 conference [BW14]. The landing gear system is composed of three parts: a digital part, including the control software, a pilot user interface, and a mechanical part which contains the doors and gears. The system is in charge of controlling the retraction and extension sequence of the gears with respect to the doors and the pilot handle. The latter serves as the input to the system. The extension and retraction sequence can be interrupted by a counter-order of the handle at any time. The full specification and a detailed description of the case study is available at [BW14].

The landing gear system has been specified in Event-B. Table 7.1 gives an overview of the refinement levels of the Event-B specification with a short description of the key aspects of the system. A detailed description of the formal specification and its validation is available at [Lad+15] and [Han+14].

In this section we demonstrate the development of an interactive formal prototype of the landing gear system. The interactive formal prototype consists of two domain specific

[3]This section presents a joint work with Dominik Hansen, Harald Wiegard, Jens Bendisposto, and Michael Leuschel and is described in [Lad+15; Han+14].

visualizations: a visualization of the architecture of the digital and hydraulic part and an additional view that visualizes the physical environment (the doors and gears). Both visualizations are inspired by pictures from the original case study specification [BW14]. Figure 7.4 shows the interactive formal prototype in action. The visualizations are composed from simple SVG graphical elements like images, shapes and lines. For instance, the electric orders to the valves and the input sensors are represented as lines, and the signals are represented as circle shapes (see upper left side of the figure). On the other hand, the visualization of the physical environment contains an image for each gear and door representing the current state of the gear and door respectively (see upper right side of the figure).

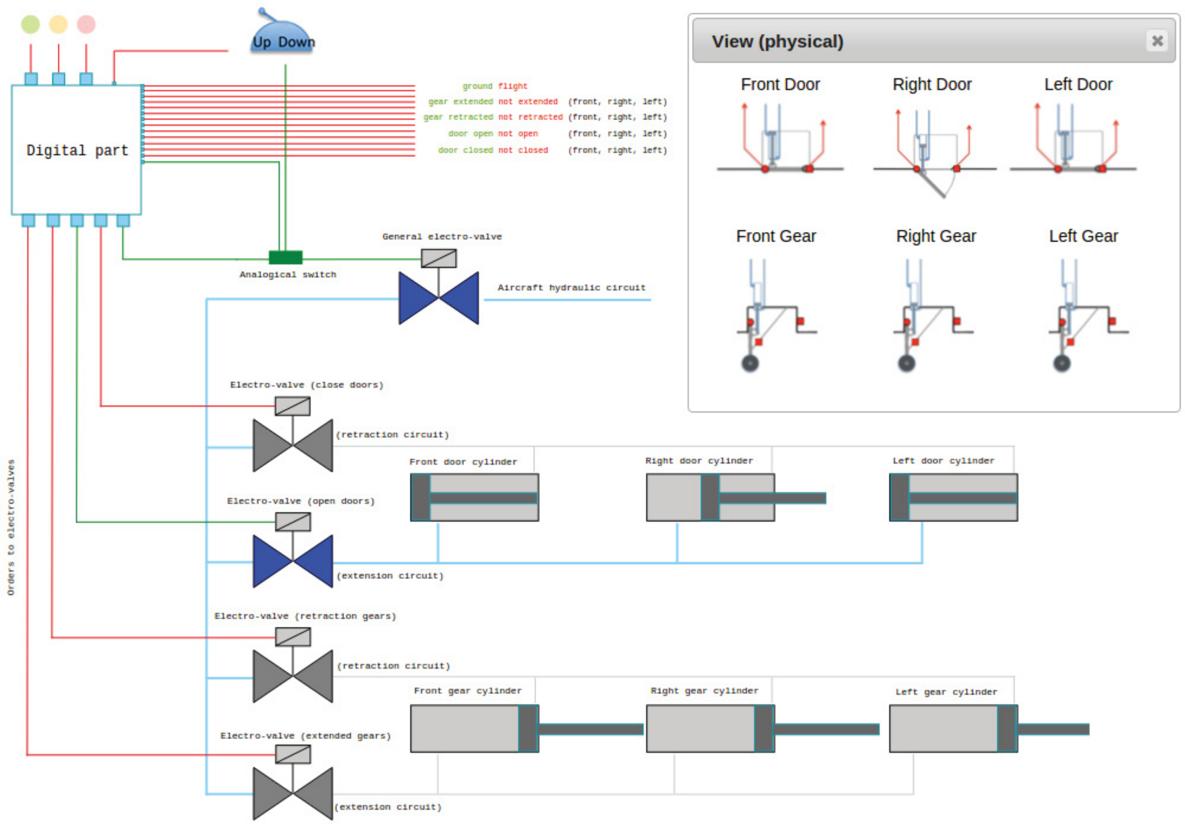


Figure 7.4.: Interactive formal prototype of the landing gear system

In the following subsections we give a detailed presentation of developing the two domain specific visualizations and the gluing code. Later in Section 7.3.3, we will also demonstrate how we have used the interactive formal prototype for validating the formal specification of the landing gear system.

Visualizing the Digital and Hydraulic Architecture

In this section we demonstrate the domain specific visualization and the gluing code for the architecture of the digital and hydraulic part of the landing gear system.

Door and gear cylinders. In the architecture visualization, the gears and doors (the abstract gear and door and the triplicated gears and doors) are represented as cylinders as illustrated in the projection diagram shown in Fig. 7.5 (right side). The projection shows the different states of the front door cylinder (*closed*, *door_moving* and *open*). As can be seen in the projection the movement of the piston in the cylinder corresponds to the state of the front door (*doors*(*front*)). In order to move the piston each cylinder is bound to a formula observer that observes the respective gear or door state variable. As an example, the left side of Fig. 7.5 shows the formula observer for the front door cylinder.

```
1  bms.observe("formula", {
2   selector: "#front_door_cylinder",
3   formulas: ["doors(front)"],
4   trigger: function (el, val) {
5    var move;
6    switch (val[0]) {
7     case "closed":
8      move = "translate(0,0)";
9      break;
10    case "door_moving":
11     move = "translate(45,0)";
12     break;
13    case "open":
14     move = "translate(90,0)";
15     break;
16    }
17    el.find("#front_door_cylinder_piston")
18      .attr("transform", move);
19  }});
```

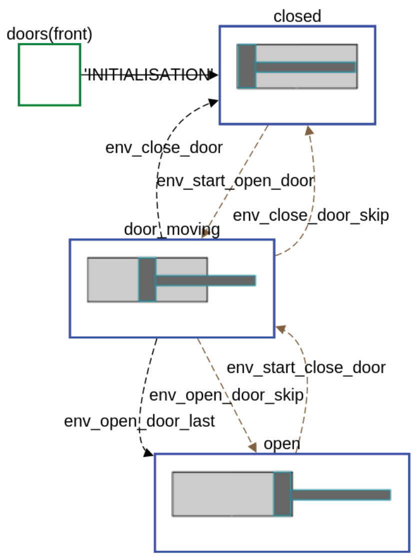


Figure 7.5.: Front door cylinder formula observer (left) and projection (right)

In line 1 and 2 we register a new formula observer for the graphical element that matches the selector #front_door_cylinder. The observer observes the formula *doors*(*front*) (line 3) and moves the piston of the front door cylinder (#front_door_cylinder_piston) based on the state value of the formula. To do this, the *transform* attribute of the piston is changed (lines 17 and 18) and the value of the attribute is determined by a simple switch case statement (lines 5 to 16). For instance, the piston is moved 45 pixels to right by setting the transform attribute to the value *translate(45, 0)* whenever the state value of the formula *doors*(*front*) is *door_moving*. We have defined the observers for the other door and gear cylinders in a similar fashion.

At the moment, the abstract gear and door and their refinement (the triplication of the abstract gear and door) introduced in refinement level *R3GearsDoorsHandle* are all displayed in the visualization. However, depending on the refinement level that is being animated, either only the abstract door and gear or the triplicated gears and doors should be displayed. In order to show this data refinement in the interactive formal prototype, we use the refinement observer. The basic idea of the observer is to hide the abstract gear and door and display the triplicated gears and doors whenever the *R3GearsDoorsHandle* is part of the animated formal specification. Otherwise, the observer should display the abstract gear and door and hide the triplicated gears and doors. To do this, we have grouped the graphical elements: there is SVG group #gear_door_abstract that groups the abstract gear and door and the SVG group #gear_door_refined that groups the triplicated gears and doors. Listing 7.22 demonstrates the refinement observer for #gear_door_refined. In line 3 we define the refinement which introduces the data refinement, namely *R3GearsDoorsHandle*. The observer triggers the enable function (see lines 4 to 6) whenever the animated machine (or one of the corresponding abstracted machines) is the defined refinement in the *refinement* option (line 3). In particular, the enable function sets the *visibility* of the group to the value *visible* (displaying the group). Otherwise, the observer triggers the disable function (see lines 7 to 9) that sets the *visibility* of the group to the value *hidden* (hiding the group). The refinement observer for the abstract gear and door is shown in Listing 7.21.

```
1  bms.observe("refinement", {
2    selector: "#gear_door_abstract",
3    refinement: "R3GearsDoorsHandle",
4    enable: function(el) {
5      el.css("visibility", "hidden");
6    },
7    disable: function(origin) {
8      el.css("visibility", "visible");
9    }
10 });
```

Listing 7.21: Abstract gear and door refinement observer (JS)

```
1  bms.observe("refinement", {
2    selector: "#gear_door_refined",
3    refinement: "R3GearsDoorsHandle",
4    enable: function(el) {
5      el.css("visibility", "visible");
6    },
7    disable: function(origin) {
8      el.css("visibility", "hidden");
9    }
10 });
```

Listing 7.22: Triplicated gear and door refinement observer (JS)

Pilot handle. The pilot handle is represented by an image element and two different image files: one for the *up* state (*handle_up.png*) and one for the *down* state (*handle_down.png*). The left side of Figure 7.6 shows the formula observer for the pilot handle that switches the path for the image of the image element (#handle) to match with the current state of the *handle* variable (lines 4 to 7). The right side of Fig. 7.6 shows the projection onto the pilot handle illustrating the two possible states of the handle in the system (*up* and *down*).

```
bms.observe("formula", {
 selector: "#handle",
 formulas: ["handle"],
 trigger: function (el, val) {
  el.attr("xlink:href",
  "img/handle_" + val[0] + ".png");
 }
});
```

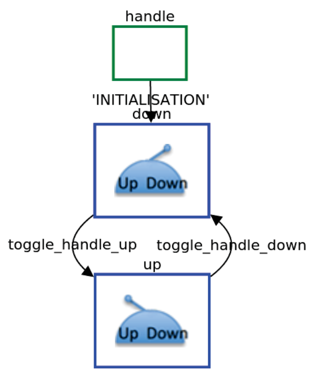


Figure 7.6.: Pilot handle formula observer (left) and projection (right)

Electric orders and valves. The electric orders from the digital part to the electro valves in the hydraulic part are represented as lines in the visualization. In order to represent the current state of the electric orders, we setup an observer for each electric order that changes the color of the lines according to the current state of the order (green when active and red when not active). For example, Listing 7.23 demonstrates the observer for the electric order to stimulate the close door electro valve (#eo_close_doors).

```
1 bms.observe("formula", {
2  selector: "#eo_close_doors",
3  formulas: ["close_EV"],
4  translate: true,
5  trigger: function (el, val) {
6   el.find(".order").attr("stroke",
7    val[0] ? "green" : "red");
8  }});
```

Listing 7.23: Close door electric order formula observer (JS)

```
bms.observe("formula", {
 selector: "#ev_close_doors",
 formulas: ["valve_close_door"],
 trigger: function(el, val) {
 el.find(".valve").attr("fill",
  val[0] === "valve_open" ?
  "blue" : "gray");
 }});
```

Listing 7.24: Close door valve formula observer (JS)

The observer observes the variable *close_EV* (determining the state of the close door electric order). In line 4 we set the *translate* flag of the formula observer to *true* determining that the value of the variable *close_EV* should be translated into a JavaScript object. For instance, whenever the value of the variable *close_EV* is “TRUE”, the value is translated into the JavaScript object *true*. The translated value is used in a conditional statement as demonstrated in lines 6 to 7, where the color is defined according to the state of the variable. In particular, we set the color of the child graphical elements which have the class *order* (the line elements that represent the electric orders) to “green” whenever the value of the variable is *true* or otherwise to “red”. Listing 7.24 shows the formula observer for the electro valve for closing the door. The observer sets the *fill* attribute of the graphical element that represents the valve (#ev_close_doors) to *blue* whenever the current state of the observed variable *valve_close_door* is set to *valve_open*. Otherwise it sets the *fill* color to *gray*. In a similar fashion we have defined

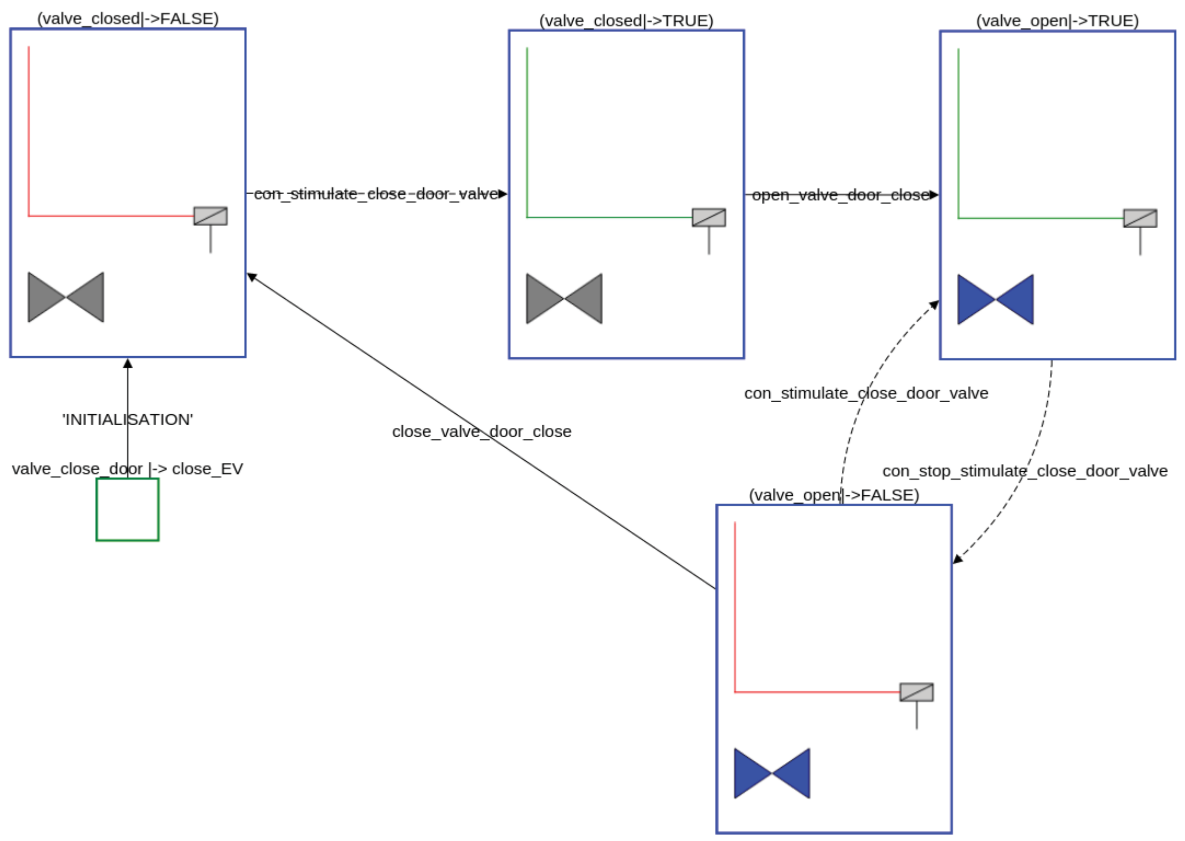


Figure 7.7.: Projection onto the close door electric order and valve (#eo_close_doors, #ev_close_doors)

the observers for the other electric orders and valves.

Figure 7.7 shows the projection onto the electric order and valve graphical elements for closing the door (using the selector "#eo_close_doors, #ev_close_doors"). The diagram confirms that the observers represent the electric orders and valves in the overall system properly.

Analog switch and general electro valve. In refinement *R6_...Switch* the analog switch and the general electro valve are introduced. Figure 7.8 shows the graphical representation of the analog switch and the electric order to the general electro valve as well as their possibles states in the system. The formula observer for the analog switch is shown in Listing 7.25. The observer observes the two variables *analog_switch* and *general_EV*. The second variable is needed to determine the state of the electric orders related to the analog switch (see lines 10 and 11). The rest of the observer is responsible for determining the state of the analog switch. To do this, two graphical elements have been created: one that represents the opened analog switch, and a second that represents the closed analog switch (see Fig. 7.8). Depending on the value of the *analog_switch* variable, the observer hides or shows the correct graphical element (lines 6 to 9).

The formula observers for the general valve and the electro order have been created by adapting the formula observers shown in Listing 7.23 and Listing 7.24. We only need to change the selector (selecting the graphical element that represents the general valve and the electro order) and the observed variable (*general_valve* and *general_EV*).

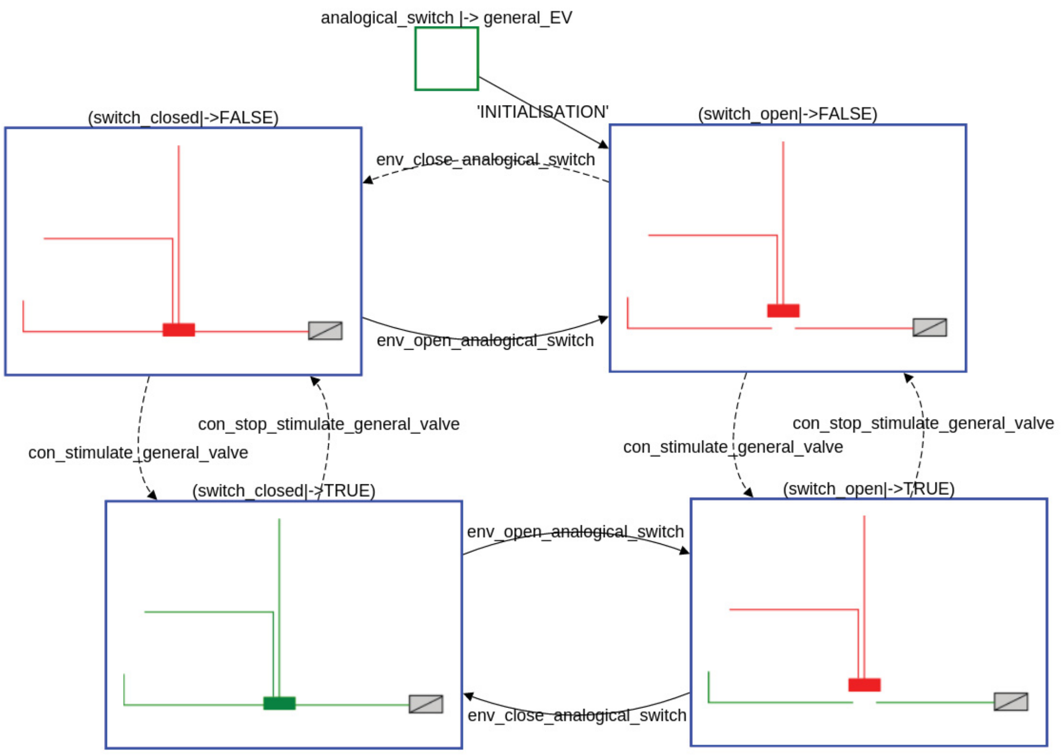

analogical_switch |-> general_EV
'INITIALISATION'
(switch_closed|->FALSE)
(switch_open|->FALSE)
env_close_analogical_switch
env_open_analogical_switch
con_stop_stimulate_general_valve
con_stimulate_general_valve
con_stimulate_general_valve
con_stop_stimulate_general_valve
(switch_closed|->TRUE)
(switch_open|->TRUE)
env_open_analogical_switch
env_close_analogical_switch

Figure 7.8.: Projection onto the analog switch of landing gear system (#analog_switch)

```
1  bms.observe("formula", {
2   selector: "#analog_switch",
3   formulas: ["analog_switch", "general_EV"],
4   translate: true,
5   trigger: function(el, val) {
6    el.find("#analog_switch_open")
7     .css("visibility", val[0] == "switch_open" ? "visible" : "hidden");
8    el.find("#analog_switch_closed")
9     .css("visibility", val[0] == "switch_closed" ? "visible" : "hidden");
10   var color = val[1] ? "green" : "red";
11   el.find(".order").attr({"fill": color, "stroke": color});
12  }});
```

Listing 7.25: Analog switch formula observer (JS)

Cockpit lights and controller sensors. Beside the orders to the hydraulic part, the controller produces three further output signals to the cockpit: (1) a green light signalizing that all gears are locked down, (2) an orange light signalizing that the gears are maneuvering and (3) a red light signalizing an anomaly in the system. We have created a circle shape element for each signal (see upper left side of Fig. 7.4) and added an observer that changes the color of the signal based on the current state (depending of the signal is switched on or off). The graphical representation of the input sensors for the digital part have been visualized in the same way as the electric orders, e.g. as line elements with observers that switch the color of the lines (depending on the current state of the sensor).

Interactive visualization. Up to this point, the visualization can only be driven by performing a state change using ProB (e.g. by clicking on an enabled event in the events view). We have used the execute event handler feature in BMotionWeb to enhance our visualization with interactive features. As an example, Fig. 7.9 shows the execute event handler that wires the events *toggle_handle_down* and *toggle_handle_up* with the graphical element that represents the pilot handle (see left side of the figure). The right side of the figure shows the execute event handler in the interactive formal prototype when the user hovers over the graphical element. A tooltip will be shown with the events and their status in the current state (enabled or disabled). We have created similar execute event handlers for the other graphical elements.

```
bms.executeEvent({
 selector: "#ev_handle",
 events: [
  { name: "toggle_handle_down" },
  { name: "toggle_handle_up" }
 ]
});
```

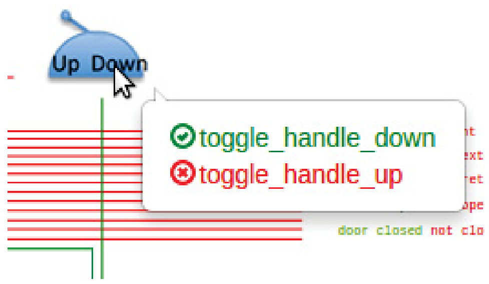


Figure 7.9.: Execute event handler for pilot handle

Visualizing the Physical Environment

The domain specific visualization of the physical environment is composed of eight image elements representing the abstract gear and door and the triplicated gears and doors (front, right and left). Each image element binds a formula observer that observes the state variable of the gear or door. For instance, the left side of Fig. 7.10 shows the formula observer for the image element that represents the physical door (#front_door). The observer switches the file path of the image element based on the current state of the *doors*(*front*) formula (lines 4 to 8). The result is illustrated in the projection diagram shown on the right side of Fig. 7.10. The diagram shows the three possible states of the front door (*closed*, *door_moving* and *open*) in the system. In order to represent the data refinement of the doors and gears, we have adapted the refinement observers shown in Listing 7.21 and Listing 7.22.

```
bms.observe("formula", {
 selector: "#front_door",
 formulas: ["doors(front)"],
 trigger: function(el, val) {
  el.find("image")
  .attr("xlink:href",
  "img/door_"+val[0]+".png");
 }
});
```

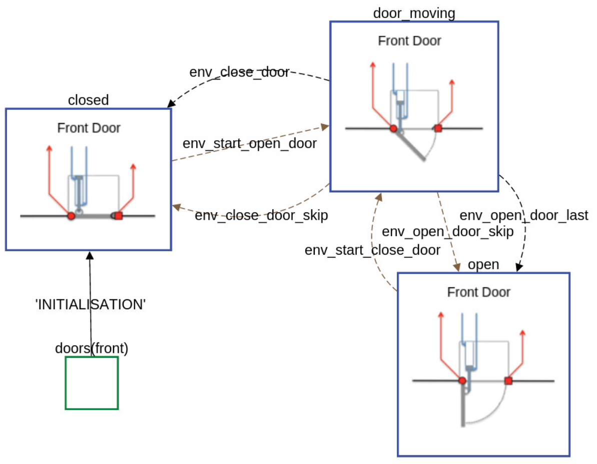


Figure 7.10.: Formula observer (left) and projection diagram (right) for physical front door

Visualizing Superposition Refinements

We already have demonstrated how a data refinement can be visualized in an interactive formal prototype while visualizing the doors and gears. In addition to data refinement, Event-B provides refinement called *superposition* refinement. A superposition refinement introduces more complexity and details into the formal specification rather than refining some abstract data structures. In order to represent a superposition refinement in the interactive formal prototype, we have subdivided the visualization into SVG groups, where each group reflects a refinement level in the specification. A group only contains the graphical elements that corresponds to a specific refinement and is identified by the class *refinementGroup*. A group is only displayed if the refinement is part of the animated formal specification. This is achieved by means of the refinement observer shown in Listing 7.26.

```
 1 bms.observe("refinement", {
 2   selector: ".refinementGroup",
 3   refinement: function(origin) {
 4     return origin.attr("data-custom");
 5   },
 6   enable: function(origin) {
 7     origin.css("visibility", "visible");
 8   },
 9   disable: function(origin) {
10     origin.css("visibility", "hidden");
11   }
12 });
```

Listing 7.26: Generic superposition refinement observer for landing gear system (JS)

In lines 1 and 2 we register a new refinement observer on the graphical element that matches the selector “.refinementGroup” (all SVG elements that have the class *refinementGroup*). In lines 3 to 5 we define the name of the *refinement* based on the *data-custom* attribute of the respective SVG group. The *data-custom* attribute defines the name of the refinement to which the SVG group belongs to. For instance, the SVG group representing the refinement level that introduces the analog switch and the general valve sets the *data-custom* attribute to *R6GearsDoorsHandleValvesControllerSwitch*. The observer triggers the enable function (see lines 6 to 8) and sets the *visibility* of the SVG group to the value *visible* (displaying the SVG group) whenever the specified refinement is part of the refinement hierarchy being animated. Otherwise, the observer triggers the disable function (see lines 9 to 11) and sets the *visibility* of the group element to the value *hidden* (hiding the SVG group).

The refinement observer is generic: whenever the formal specification is refined (e.g. with superposition refinement), we only need to group the graphical elements that belong to the new refinement level and set the class to *refinementGroup* and the *data-custom* attribute to the new refinement level (the machine name) of the group.

7.3.2. Hemodialysis Machine

The second industrial case study considered in this thesis is a *hemodialysis machine* (HD machine) that has its origin in the medical domain.[4] Similar to the landing gear case study, the HD machine was originally given as a challenge for the ABZ'16 conference [Mas15]. The task of the HD machine is to remove excess fluid, minerals, and waste from the blood of a patient with kidney failure. A detailed description of the case study is available at [Mas15]. The case study has been specificed in Event-B using the iUML-B tool [Sno14] which provides a diagram based modeling notation for Event-B inspired by classical UML state machines and class diagrams. Table 7.2 gives an overview

[4]This section presents a joint work with Thai Son Hoang, Colin Snook, and Michael Butler and is described in [Hoa+16].

Table 7.2.: Overview of refinement levels of the Event-B hemodialysis machine

Ref.	Description
m0	The main system state (control system top-level) with three main phases: PREP, INIT, END (together with the STANDBY State).
m1	The sub-processes within each main phases for the top-level control system.
m2	The low-level control system for automatic testing of control functions.
m3	The actual physical result of testing the HD machine's control functions.
m4	Message passing communication between the low-level control system and the HD machine for testing control functions.
m5	Introduction of set of signals.
m6	Signal for indication of control function testing result.
m7	Connection of concentrate to the HD machine.
m8	Setting rinsing parameters.
m9	Connecting patient sequence.
m910	Connecting patient (physically).
m911	Introduction of pressure monitors and system normal/abnormal states.
m912	Introduction of various abnormal blood-side pressures.
m913	Introduction of blood pump, actual blood flow and abnormal situations when monitoring the blood flow.
m914	Introduction of arterial bolus.
m915	Introduction of heparin bolus.

of the refinement levels of the Event-B specification. A more detailed description of the specification is available at [Hoa+16].

In the following section, we present the development of an interactive formal prototype for the HD machine based on the last refinement level *m915* of the corresponding Event-B specification. The interactive formal prototype has been developed in collaboration with the university medical centre of Düsseldorf which also treats patients with kidney failures using HD machines. They provided us with domain knowledge about HD systems and gave us instruction documents for their HD machines.

Visualizing the User Interface

The interactive formal prototype of the HD machine contains a user interface (UI) that provides communication between the operating user and the HD machine. The UI consists of two panels as shown in Fig. 7.11: a panel for visualizing all relevant information about the therapy, such as the dialysis parameters and the alarm conditions (see lower right side of figure) and a panel for entering the different rinsing parameters while preparing the therapy (see upper left side of figure). The user can switch between the two panels by pressing the corresponding buttons at the bottom of the UI.

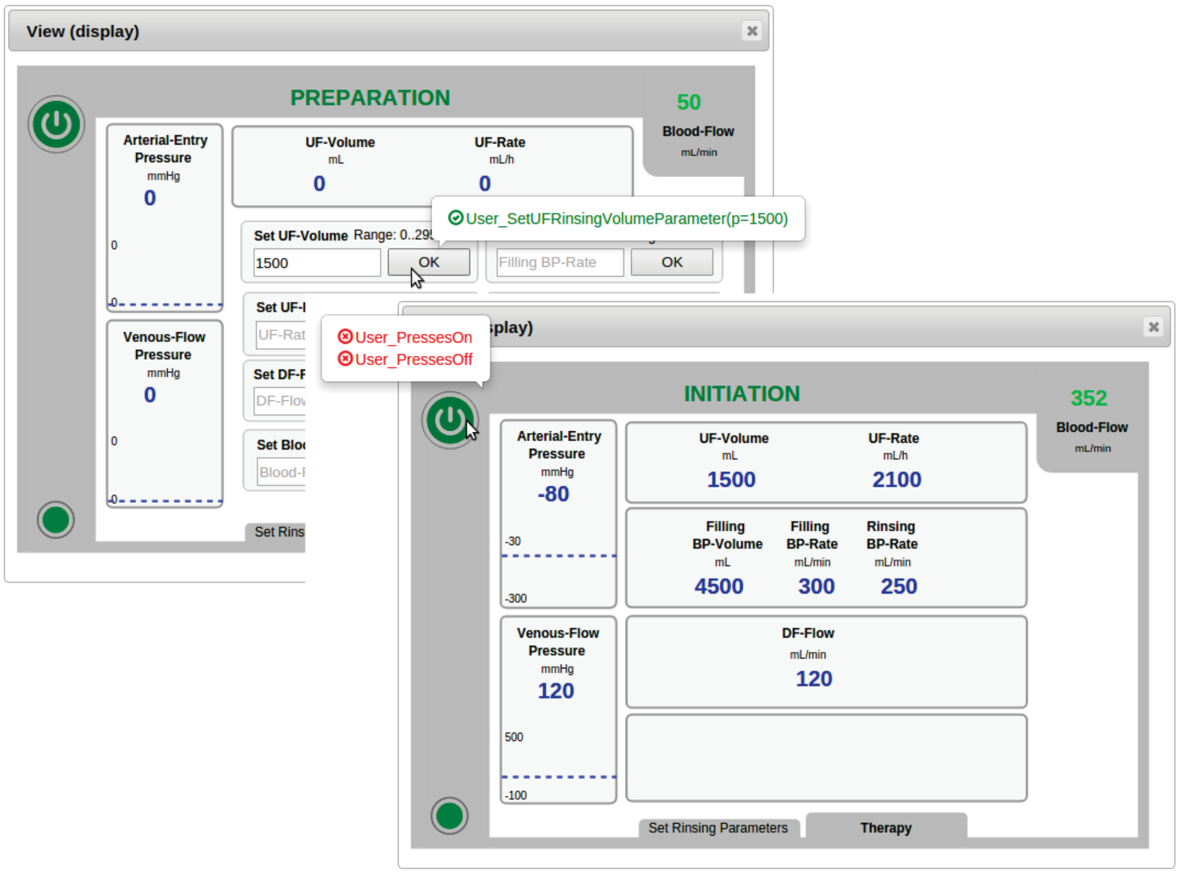


Figure 7.11.: Domain specific visualization of the HD machine's UI

Therapy panel. In the therapy panel, each dialysis parameter is represented using simple graphical elements to display its description, unit and current value, and binds a formula observer that observes the respective state variable for the parameter. In addition, for pressure parameters (arterial entry pressure and venous flow pressure), the width and thresholds of the limits window are shown with the current value being represented by a horizontal dashed line. As an example, Listing 7.27 shows the formula observer for visualizing the arterial entry pressure (AP). The observer is linked to the graphical element that represents the AP pressure (#AP_display) and observes the formulas $pressure(AP)$ (the current AP pressure state value), $limit\_high(AP)$ (the upper arterial limit) and $limit\_low(AP)$ (the lower arterial limit). Based on the values of the formulas we set the position of the horizontal dashed line to represent the current AP pressure within the limits window as demonstrated in lines 5 to 11. Moreover, the current AP pressure value and the limits are visualized as numerical values (see lines 12 to 14).

```
bms.observe("formula", {
 selector: "#AP_display",
 formulas: ["pressure(AP)", "limit_high(AP)", "limit_low(AP)"],
 trigger: function(origin, values) {
  var range = values[1] - values[2];
  if (range > 0) {
   var unity = 50 / range;
   var offset = values[2] < 0 ? (values[2] * unity) * -1 : 0;
   var value = (unity * values[0]) + offset;
   origin.find("#AP_line").attr("transform", "translate(0,-" + value + ")");
  }
  origin.find("#AP_value").text(values[0]);
  origin.find("#AP_limit_high").text(values[1]);
  origin.find("#AP_limit_low").text(values[2]);
 }
});
```

Listing 7.27: Formula observer for arterial entry pressure display of the HD machine's UI (JS)

The UI also contains graphical elements and observers for the automated self test signal lamp (see lower left side of the panels in Fig. 7.11) and for the alarm signal and the HD machine on/off button (see upper left side of the panels in Fig. 7.11). The observer for the automated self test signal lamp is responsible for indicating whether the automated self test has been successfully completed (shown in green) or not (shown in red) based on the observed formula $signal\_status(CF\_TESTING\_SIGNAL)$. The alarm signal flashes red whenever the top-level control system raises an alarm (e.g. due to abnormal behaviour). On the other hand, the HD machine on/off button wires the events *User_PressesOn* and *User_PressesOff* using an execute event handler.

Set rinsing parameters panel. In the therapy preparation phase ($CS\_TopLevel = PREPARATION$), the user is requested to enter the rinsing parameters. To do this, the UI contains a second panel shown in the upper left side of Fig. 7.11. The panel provides several input fields and buttons which wire execute event handlers for setting the value of the respective parameter. For instance, Listing 7.28 shows the execute event handler for the set ultra filtration (UF) volume button (#bt_setUFVolume). The handler wires the *User_SetUFRinsingVolumeParameter* event and determines the event's predicate based on the value entered into the UF volume input field (#input_UFVolume). Figure 7.11 shows the UI where the user hovers the UF volume button. We have created similar handlers for the other input fields.

```
bms.handler("executeEvent", {
  selector: "#bt_setUFVolume",
  events: [{
    name: "User_SetUFRinsingVolumeParameter",
    predicate: function(origin, container) {
      var val = container.find("#input_UFVolume").val();
      return "p=" + val;
    }
  }]
});
```

Listing 7.28: Execute event handler for the UF volume button (JS)

Visualizing the Environment

The second view of the interactive formal prototype contains a domain specific visualization of the environment for the HD machine (see Fig. 7.12). We have reused an existing SVG image of a schematic illustration of a hemodialysis circuit[5] rather than creating a completely new SVG image. The purpose of the visualization is to show how the different parts of the system are connected together. For instance, the visualization shows the connection from the patient to the HD machine, the connected concentrate, and contains graphical elements that represent the dialysis pressure parameters (arterial, venous, and blood entry pressure) and their connection to the environment.

Similar to the interactive formal prototype of the landing gear system (Section 7.3.1), the visualization of the HD machine environment is subdivided into SVG groups, where each group represents a different refinement level. In order to display and hide the graphical elements with respect to the refinement levels, we have adapted the generic superposition refinement observer introduced in Section 7.3.1 (see Listing 7.26).

7.3.3. Validation of the Case Studies

In this section, we demonstrate the application of the interactive formal prototypes to support the validation of the landing gear system and the HD machine Event-B specifications. In a first step, we describe the general application of the interactive formal prototypes. Then we describe some specific applications for validating the landing gear system and the HD machine.

[5]The SVG image has been taken from the English Wikipedia article about Hemodialysis (see `https://en.wikipedia.org/wiki/Hemodialysis`).

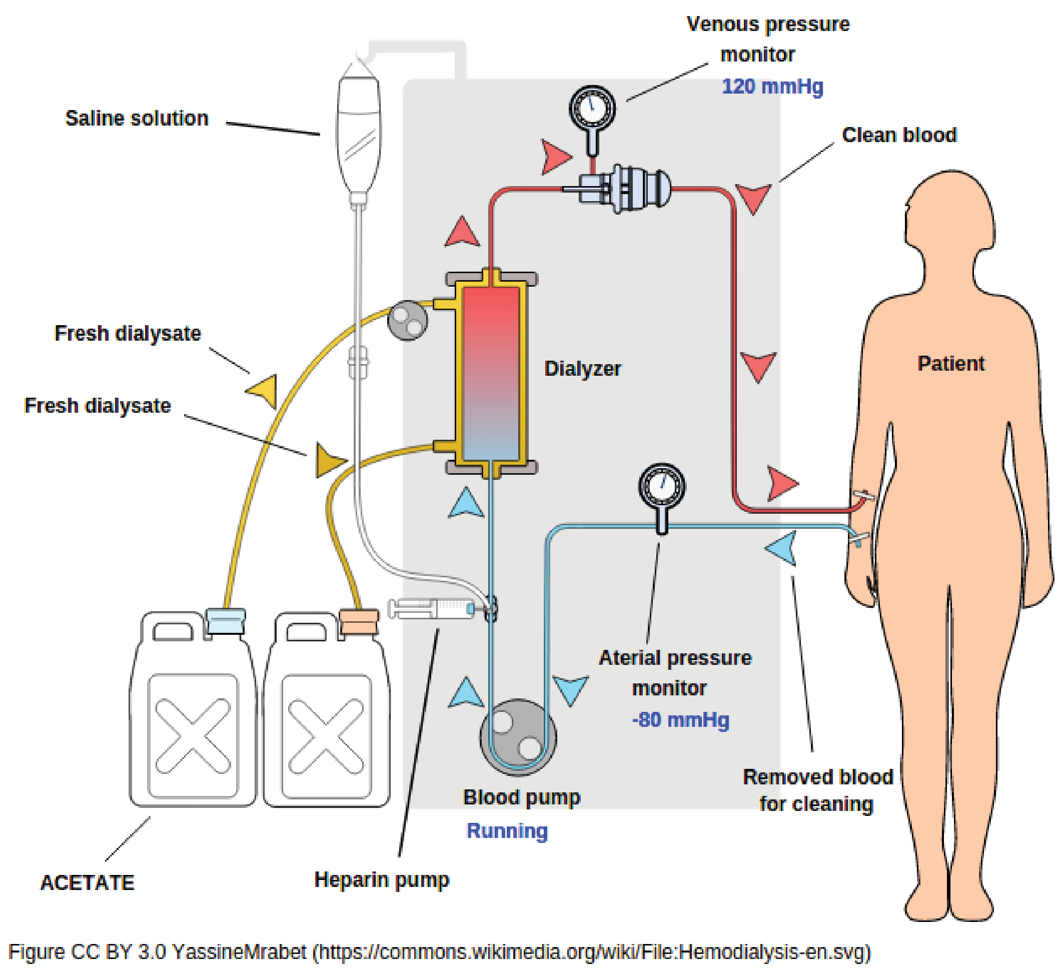
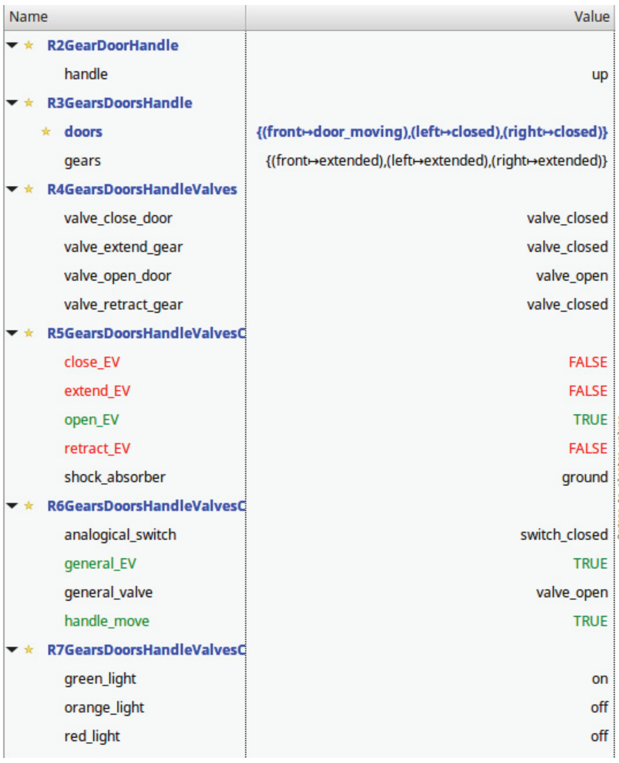


Figure 7.12.: Domain specific visualization of the HD machine environment

General Application

In general, the interactive formal prototypes helped us to reach a common understanding about the formal specifications, to analyze a specific state of the system, and to identify faulty behavior and errors during development. This is particularly valuable when the formal specification becomes complex and large in later refinement levels. The amount of details (e.g. the number of variables) of the formal specification increases with the introduction of new refinement levels. This can make the examination of a specific state difficult when only using an animation tool. For instance, Fig. 7.13 demonstrates two different representations of the same state in the landing gear system. The left side shows the variable configuration of the state in the state view of the ProB animator, while the right side shows the same state in the interactive formal prototype. It can be very hard for the user to get an overview of the current state while examining only the textual representation. A graphical representation of the state may help to avoid this problem. For instance, based on the graphical representation we can see at a glance (without knowledge about the variables in the underlying formal specification) that the electric order to the open door valve is stimulated and the front door cylinder is moving.



Name	Value
R2GearDoorHandle	
handle	up
R3GearsDoorsHandle	
doors	{(front↦door_moving),(left↦closed),(right↦closed)}
gears	{(front↦extended),(left↦extended),(right↦extended)}
R4GearsDoorsHandleValves	
valve_close_door	valve_closed
valve_extend_gear	valve_closed
valve_open_door	valve_open
valve_retract_gear	valve_closed
R5GearsDoorsHandleValvesC	
close_EV	FALSE
extend_EV	FALSE
open_EV	TRUE
retract_EV	FALSE
shock_absorber	ground
R6GearsDoorsHandleValvesC	
analogical_switch	switch_closed
general_EV	TRUE
general_valve	valve_open
handle_move	TRUE
R7GearsDoorsHandleValvesC	
green_light	on
orange_light	off
red_light	off

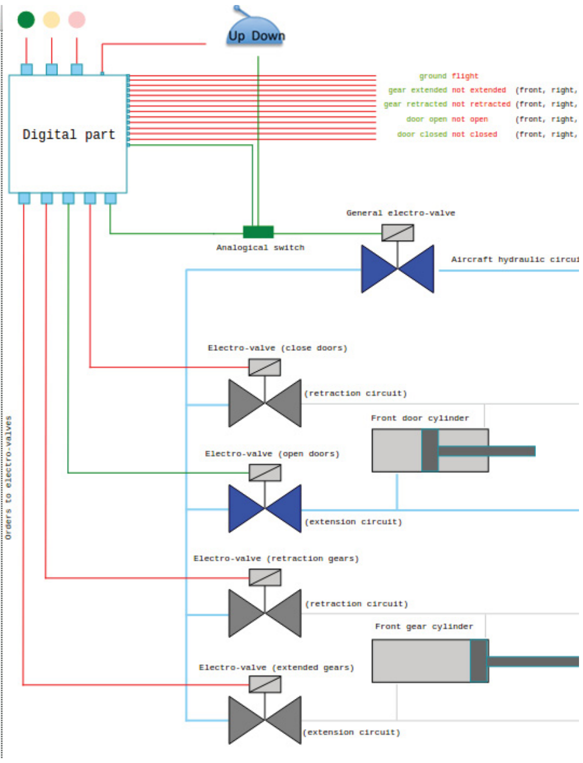

Up
Down
Digital part
ground flight
gear extended not extended (front, right,
gear retracted not retracted (front, right,
door open not open (front, right,
door closed not closed (front, right,
General electro-valve
Analogical switch
Aircraft hydraulic circui
Electro-valve (close doors)
(retraction circuit)
Front door cylinder
Electro-valve (open doors)
(extension circuit)
Orders to electro-valves
Electro-valve (retraction gears)
(retraction circuit)
Front gear cylinder
Electro-valve (extended gears)
(extension circuit)

Figure 7.13.: Textual (left) versus graphical representation (right) of a state

The interactive formal prototypes also helped us to discover problems with the specifications during their development, especially liveness problems or deadlocks, where the system cannot make any progress. We will also demonstrate the use of projection diagrams to discover incorrect behavior in the specification of the landing gear system.

The strict separation between the visualization part and the gluing code (observers and event handlers) makes the visualization reusable for other Event-B or even classical-B specifications of the landing gear system. Indeed, to adapt the visualizations for a different formal specification one simply has to change or adapt the gluing code without modifying the visualization. This has been done for the Event-B specification of the landing gear system by Su and Abrial described in [SA14]. The proecess is described in [Lad+15] and it was shown that it is feasible to adapt an existing visualization for a different formal specification within a reasonable time (the authors claim that it took them about four hours to adapt the visualization).

Landing Gear System

We have used the external method call feature of BMotionWeb (see Section 4.9) to check whether the two basic scenarios: the extension sequence and the retraction sequence are realized accurately in our formal specification. Consider the Groovy script file in Appendix D.1. In lines 1 to 40 we define a sequence of events (including the name and the predicate of the event) for simulating the extension and the retractions sequence. In lines 42 to 70, we register a new external method called *replay*. The basic idea for the method is first to bring the animated formal specification back to the root state and then execute the events of the user defined trace gradually. In order to execute the method, a button has been added to the visualization of the hydraulic part that is wired with the execute method handler shown in Listing 7.29. Once the button is pressed the registered *replay* method on the server side is called and the user can observe the extension and retraction sequence within the interactive formal prototype.

```
bms.handler("method", {
  selector: "#replayButton",
  name: "replay"
});
```

Listing 7.29: Replay extension and retraction sequence method handler (JS)

The handle and gears of the landing gear system are closely related since the extension and retraction of the gears can always be interrupted by a counter order from the handle. A projection on both aspects of the specification (the gears and the pilot handle) helped us to inspect their behaviour in the process of specifying the landing gear system. Consider the projection diagram shown in Fig. 7.14. The diagram was produced with BMotionWeb and demonstrates the projection of the fourth refinement level of an earlier version of the landing gear system specification on the graphical elements that represent the pilot handle (#ev_handle) and the front gear (#front_gear) of the physical environment. The projection function $p(s) = eval(E, s)$, with $E = gears(front) \mapsto handle$ is automatically derived from the observers of the graphical elements.

The diagram confirms that in every state the handle can be toggled (the corresponding transitions are definite) and that the only event which can modify the handle is *env_toggle_handle*. We can also see that the gear do not jump directly from *retracted* to *extended* or vice versa. The transitions for changing the gear state are not definite: this is to be expected, as the doors have to put into the correct position first. This again confirms our intuition about the specified system. However, two transitions were initially surprising: it seems like the gear can start retracting on its own when the gear is extended and the handle is down. Similarly, it seems like the gear can start extending on its own when the gear is retracted and the handle is up. This is the case when the handle was reversed right after the doors had opened and the retract gear valve had been stimulated.

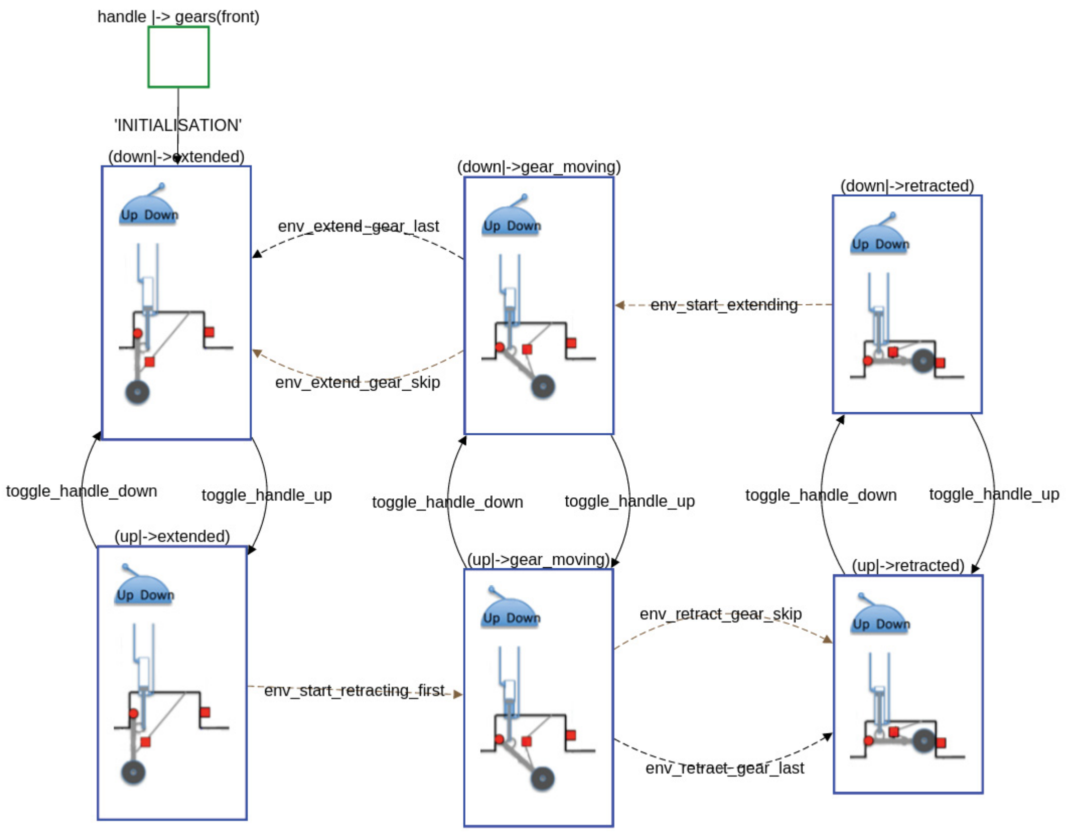


Figure 7.14.: Projection of the fourth refinement level on handle and front gear

Another example application of the projection diagram feature is demonstrated in Fig. 7.15. The diagram illustrates the possible values of the gear and door in the state space of an earlier, faulty version of the fourth refinement of the landing gear system. One can clearly see that something is amiss. For instance, both the door and the gear can be moving at the same time. Figure 7.16 contains a similar projection for the final corrected version of the specification. Here we can clearly see that the retraction/extension and opening/closing sequences happen in the right order.

Hemodialysis Machine

In the case of the HD machine, requirements related to the AP and VP pressures such as **R-5**–**R-8** are modeled by whether or not the iUML-B transitions (ultimately events) are enabled. Such requirements are cumbersome to formulate as a proof obligations in Event-B but can be readily demonstrated via an interactive formal prototype. In order to validate whether the requirements have been adequately taken into account, we have created an additional view for the HD machine interactive formal prototype for entering controller test values for the treatment parameters (AP and VP pressures and their limits). By means of the view the user can test different combinations of the treatment

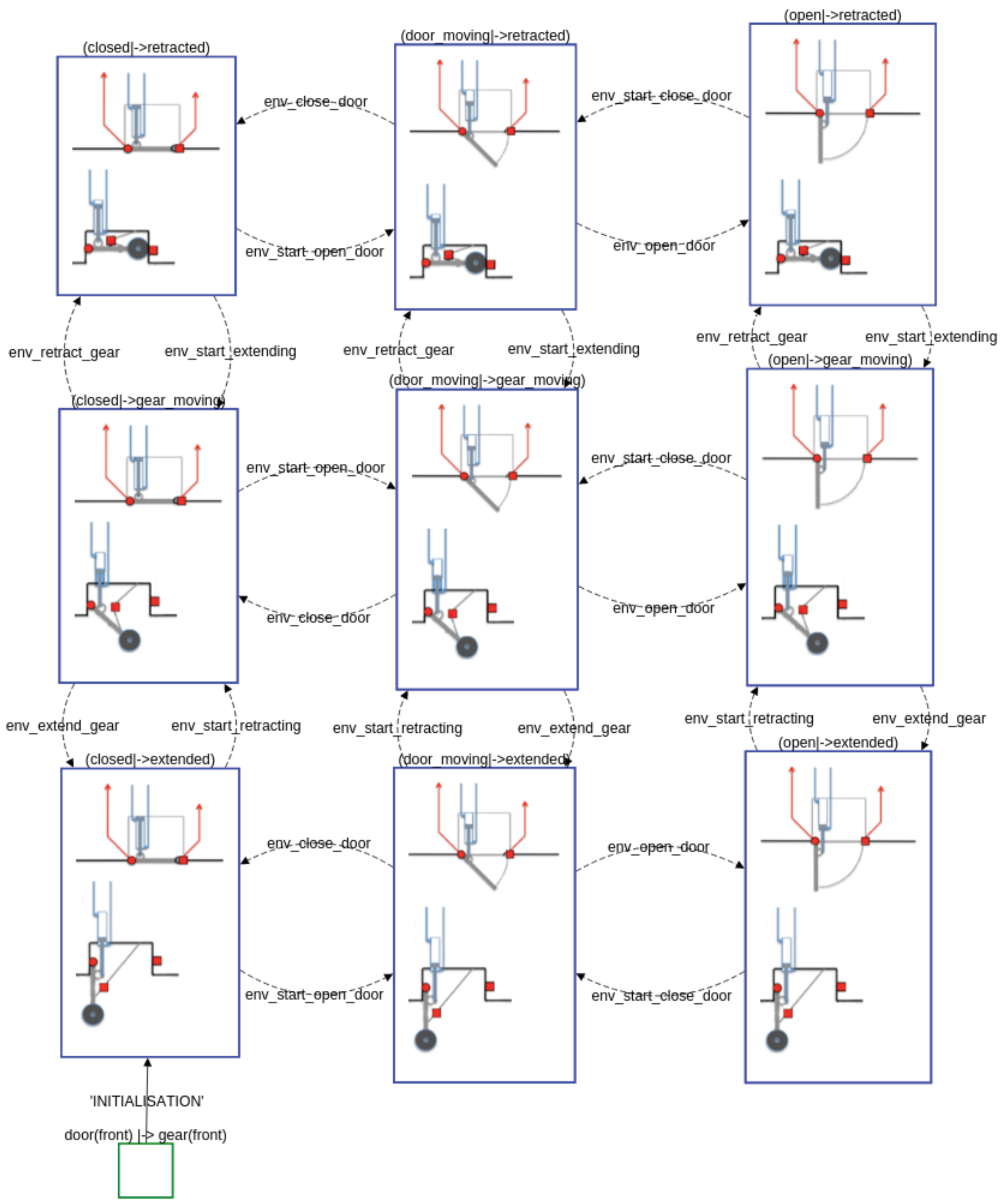


Figure 7.15.: Projection of an erroneous version of the fourth refinement level

parameters while running the interactive formal prototype and observe the effect on the system. As an example, we have simulated different values for the AP and VP to check whether the corresponding alarm signals are executed if the software detects that the AP or VP exceed their upper limit or fall below their lower limit (**R-5**–**R-8**). Figure 7.17 demonstrates the view, where the user enters test values for the AP pressure and its limits.

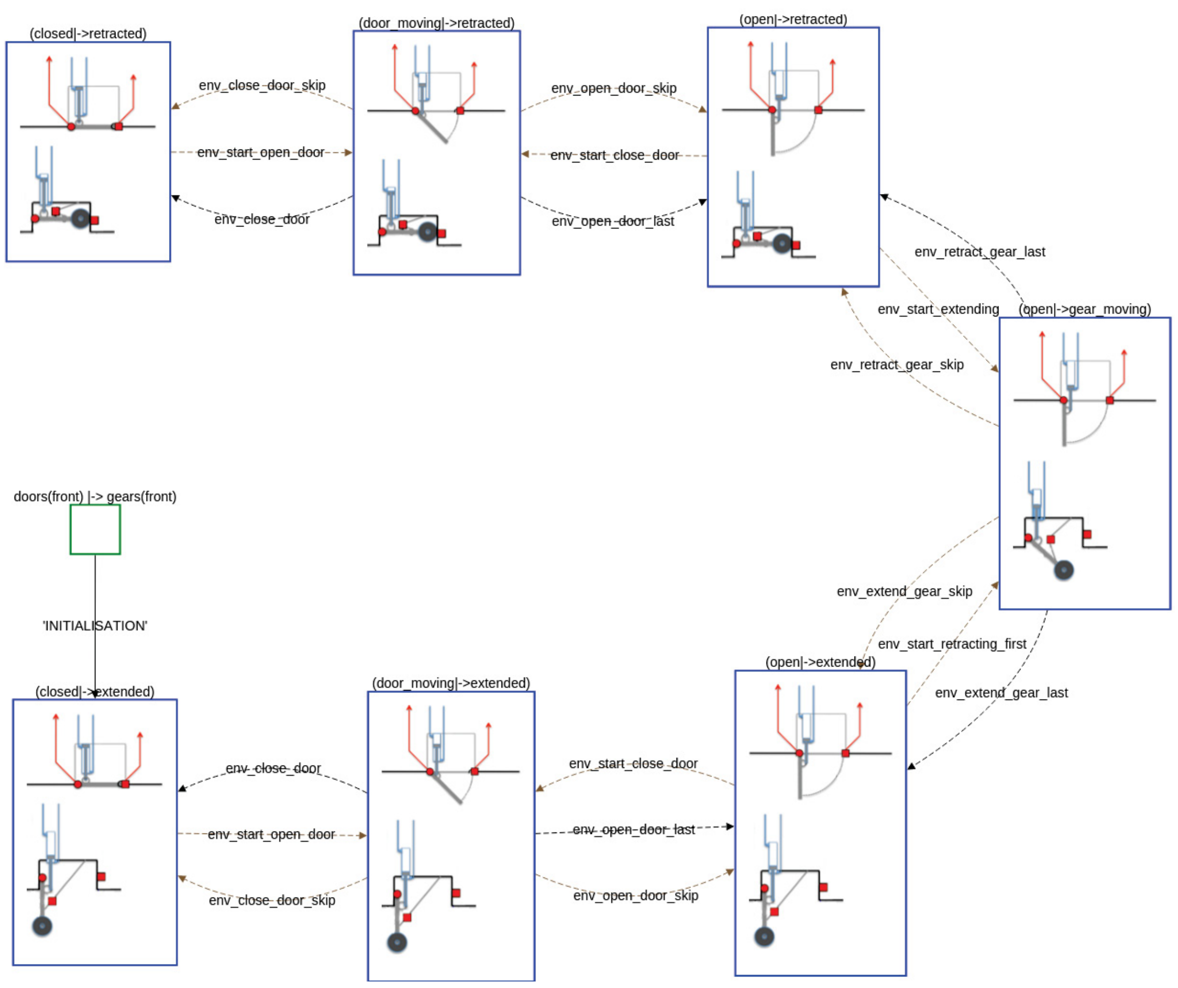
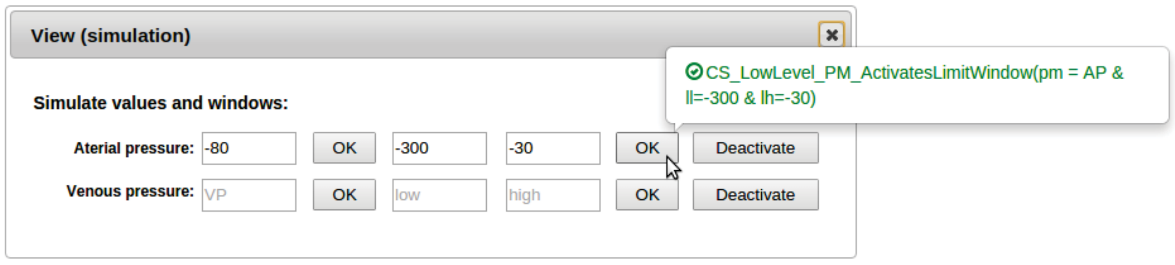


Figure 7.16.: Projection on corrected version of the fourth refinement level


View (simulation)

CS_LowLevel_PM_ActivatesLimitWindow(pm = AP & ll=-300 & lh=-30)

Simulate values and windows:

Aterial pressure:	-80	OK	-300	-30	OK	Deactivate
Venous pressure:	VP	OK	low	high	OK	Deactivate

Figure 7.17.: Simulation view of HD machine

We have also recorded the trace produced the simulation of the AP and VP test controller values to have the opportunity to reproduce the simulation in a later stage of the development. In order to reproduce the trace, we have used the recorded trace together with the replay trace approach introduced in Section 7.3.3. This can be particularly useful to revalidate the simulation together with a domain expert or to provide evidence that the simulation realizes the requirements adequately. As an example, consider the trace defined in Appendix D.2. The trace defines the recorded events to simulate **R-5** "*During initiation, if the software detects that the pressure at the VP transducer exceeds the upper limit, then the software shall stop the BP and execute an alarm signal*". Once, the trace has been started, the user can observe the scenario within the interactive formal prototype and go back in the history to analyze a specific state in more detail.

7.4. Success Stories from other Industrial Applications

In this section we give some brief success stories of applying BMotionStudio and BMotionWeb to other industrial case studies. Section 7.4.1 reports how BMotionWeb was used for two case studies provided by the Advance FP7 EU project and Section 7.4.2 presents how BMotionStudio was used in the railway domain according to employees at the Thales company.

7.4.1. BMotionWeb in the Advance FP7 EU Project

Advance is an FP7 EU project [Adv] with the overall objective to develop a unified tool-based framework for automated formal verification and simulation-based validation of cyber-physical systems. The Advance project provided two industrial case studies: a smart grid from the smart energy domain and an interlocking system from the railway domain. The case studies are described in [BR13] (smart grid) and [Mej13] (interlocking system).

Smart Grid

A smart grid is an electrical grid with a two-way communication between the consumer and supplier. The main objective of a smart grid is to control the production and distribution of electricity efficiently. The purpose of the case study in the Advance project was to verify the security and reliability of the communication infrastructure of the smart grid system using formal methods. To do this, an Event-B specification of the smart grid was developed and used together with tools like Rodin for the verification and ProB for the validation of the specification. Moreover, a domain specific visualization was developed using the BMotionStudio tool. During the project, the visualization was then migrated to an interactive formal prototype created with BMotionWeb. The reason for the migration was that BMotionWeb provides a lot more flexibility and scope in terms of how domain specific visualizations are created and executed (e.g. the reuse of existing SVG images and JavaScript libraries, and scripting support). Section 6.3 of the final case study report [BK14] describes the original visualization, as well as the migration to an interactive formal prototype.

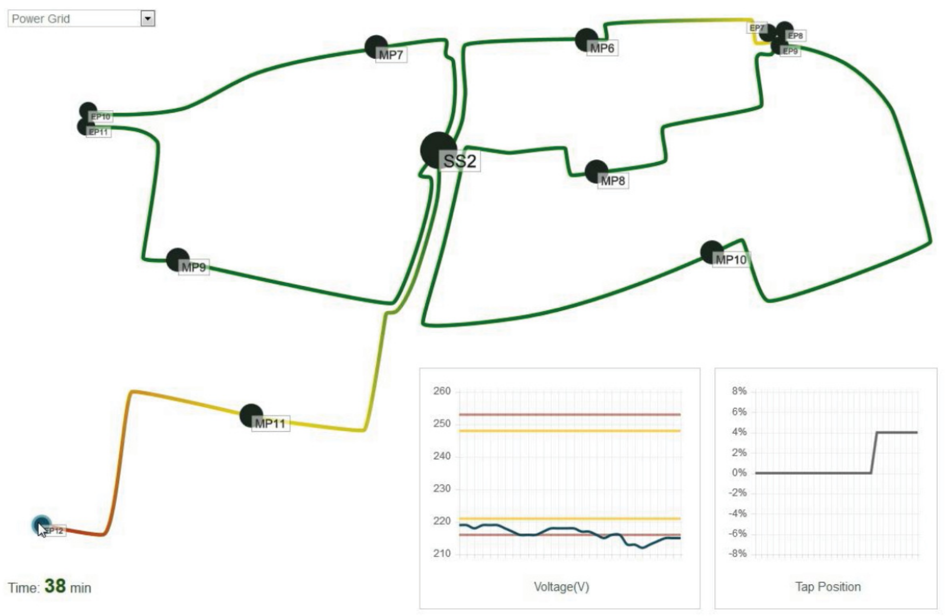


Figure 7.18.: Screenshot of the interactive formal prototype including a visualization of the Advance smart grid low voltage network taken from [BK14]

The smart grid interactive formal prototype contains two views: a view visualizing the communication network and a view including a visualization of the low voltage network as demonstrated in Fig. 7.18. The communication network illustrates the physical topology of the smart grid, whereas the purpose of the low voltage network visualization is to validate how the algorithm was reacting to changes in the voltage network.

The developers of the smart grid interactive formal prototype report in [BK14] that "*a major success of the Advance programme has been the BMotion tool*". Furthermore, they confirm that our thesis goal to complement the benefits of animation using visualization techniques was successful: "*Visualization was found to be essential in comprehending the results of the simulation, and understanding why any unexpected or non-optimal behaviour occurred*". The developers have also used the interactive formal prototype to "*communicate the technical details of the models to domain engineers, without the requirement for them to understand the underlying mathematical formalisms*". The smart grid case study and especially the success story reported in [BK14] confirm the success of our thesis goal and the power of visualization techniques to support the validation process of formal specifications.

Interlocking System

An interlocking system manages signal and switch states on a railway track with the goal of preventing conflicting movements of trains. In the Advance project, a classical-B specification of an interlocking dynamic controller (IXL-DC) was developed with the overall objective of demonstrating the feasibility of formal method techniques in the railway domain [MKS14].

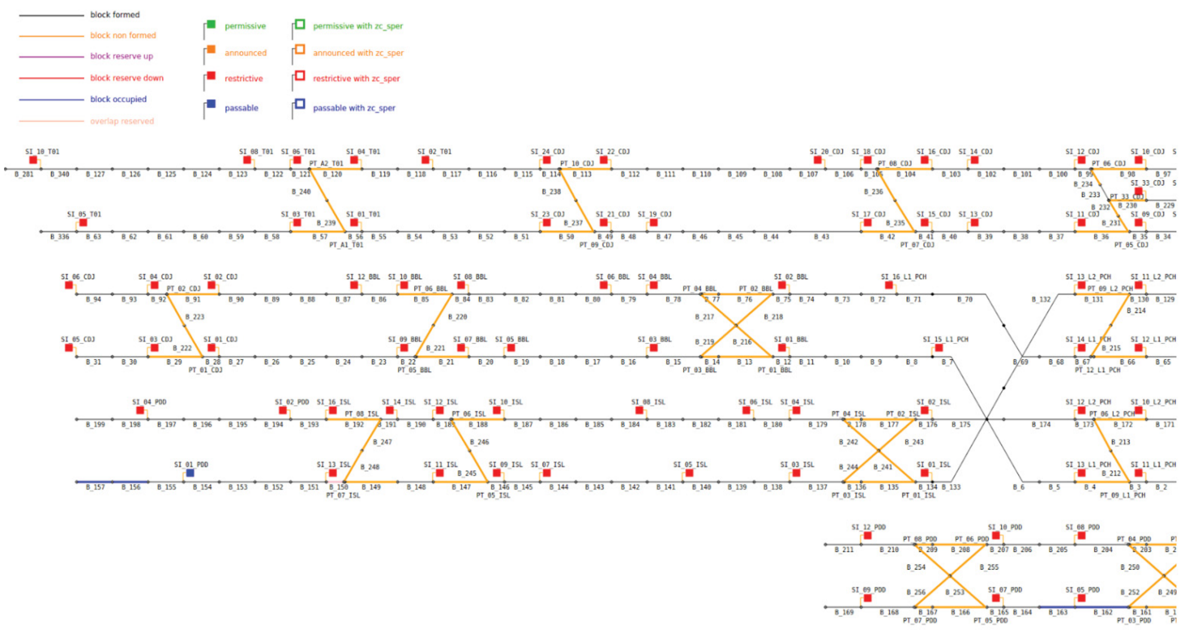


Figure 7.19.: Screenshot of the Advance interlocking system interactive formal prototype taken from [Jen14]

To support the validation of the specification, the authors of the case study created an interactive formal prototype for the case study as demonstrated in Fig. 7.19. In an intermediate report of the case study [MK13], the authors report that “Graphical animation helped us to clarify and develop the behaviour of the IXL-DC”. Moreover, they report in [MKS14] that “*automatic animation (including visualization) was extremely useful to develop the model. It disclosed defects that cannot be disclosed by proof because they reflect incorrect comprehension of the functions of the IXL and of its dynamics*”. This again confirms the success of our thesis goal and the benefits of applying animation together with visualization.

7.4.2. BMotionStudio in the Railway Domain

The authors in [RFT16] (who are employees of the Thales company) present the use of BMotionStudio to support validation in the railway domain. The Event-B specification and the domain specific visualization can be accessed and downloaded from Github.[6] Although developing a domain specific visualization with BMotionStudio required extra effort, the benefits of the visualization were considerable. For example, the authors report in [RFT16] that "*it is essential to support the domain experts so that they can perform their tasks effectively. In this process, domain specific modeling, model visualization and animation are of major assistance*". However, the visualization was created for only one railway station with a specific track layout and topology. In future, the authors plan to migrate the visualization to BMotionWeb in order to visualize different track layouts and topologies.

7.5. Teaching Formal Methods

The BMotionWeb and BMotionStudio tools are also used for teaching formal methods and for public engagement. We have created several visualizations of case studies from formal methods teaching books (e.g. the interlocking system and the location access controller from [Abr10]), as well as various visualizations of puzzles and logic games (see Appendix B.1). These visualizations are used in our master degree courses at the University of Düsseldorf to support the teaching of formal method techniques. As an example, we use visualization to provide a concrete example of an abstract formal specification. Visualization can also help students understand the meaning and behavior of a formal specification. We also allow our students in our formal methods course to develop their own visualizations. The feedback we get from the students is very positive. Creating visualizations can help students learn formal methods.

Another example of an application of BMotionStudio was created by Dana Dghaym, Asieh Salehi, and Colin Snook. For the annual "Science Day" in Southampton they used Rodin and Event-B to demonstrate aspects of science related to their research, e.g. by demonstrating aspects about how mathematics can help to analyze problems. The particular idea of the event was to solve a simple safety related problem based on parking two cars in two parking spaces crossing protected by a signal. The problem was specified in Event-B. Since, many visitors are children, the demonstration had to be designed and adapted by them. To overcome this challenge, they used BMotionStudio to provide a cartoon style visualization and a simple Venn diagram visualization of the Event-B specification rather than showing some complex mathematical formulas.

Dana Dghaym, Asieh Salehi and Colin Snook report that "*the exercise was very popular throughout the day and at times children queued to try their selections. In all, two hundred children performed the exercise. Several teachers commented on how useful they thought the exercise was for the children.*". More information about the case study and the experiences gained by applying it is available at the website of the Rodin Workshop 2016.[7]

[6]`https://github.com/klar42/railground`.

[7]`http://wiki.event-b.org/index.php/Rodin_Workshop_2016`.

8 Future Work, Conclusion and Discussion

8.1. Future Work

Before we draw the final conclusions in Section 8.2, we will discuss some ideas that should be addressed in the future. We first discuss future work which concerns the improvement of BMotionWeb and the creation of interactive formal prototypes. Then we present further ideas for the projection diagram approach and for combining interactive formal prototypes with other visualization approaches.

Improve the use and creation of interactive formal prototypes. In general we plan to improve the use and creation of interactive formal prototypes. This especially concerns the further development of the built-in visual editor (see Section 4.10). Some further ideas for improvements are:

- Display information about observers and interactive handlers within interactive formal prototypes.
- Add the execute event handler directive used in the interactive system case studies (see Section 7.2) as a default feature in BMotionWeb and develop further directives (e.g. for easily binding relations or sets to HTML tables).
- Add the *replay trace* feature used in the industrial case studies (see Section 7.3) as a default feature in BMotionWeb.

Address a wider range of safety-critical systems. BMotionWeb supports the state-based formal methods Event-B and classical-B and the event-based formal method CSPM. This supports the creation of interactive formal prototypes for software and reactive systems (classical-B and Event-B) and for concurrent systems (CSPM). In the future, we plan to integrate other animation tools with BMotionWeb to address a wider range of safety-critical systems. First experiments towards supporting the CoreASM animator [FGG07] have already been made.

Lightweight validation of interactive systems. In Chapter 7 we have shown the use of BMotionWeb for supporting the validation of *interactive* systems. We also plan to develop more features for the lightweight validation of interactive systems, such as A/B testing to compare two variants of a system user interface or device.

Link to requirements. The development of a formal specification is typically based on a set of requirements. In order to trace back to the affected requirements whenever incorrect behavior or a problem has been detected in the formal specification it is useful to maintain a traceability between the requirements and the elements in a formal specification (e.g. invariants, guards or variables). Traceability may also help to identify the affected elements in the formal specification whenever the original requirements document changes. [HJL13] presents an approach for tracing requirements within formal specifications. We plan to extend this approach to also trace the requirements within interactive formal prototypes (via the formal specification) and to display the requirements in the context of the domain specific visualization. For instance, it might be interesting to understand why a button which is wired to an Event-B event is disabled in a specific state (e.g. because a guard is false).

Combine interactive formal prototypes with other visualization approaches. For future work, we plan to combine interactive formal prototypes with other visualization approaches. As an example, we plan to combine LTL counterexample visualizations [Tol11] with interactive formal prototypes. We also plan to enhance the presented trace diagram visualizations with interactive features. Particularly when visualizing counterexamples as trace diagrams, we plan to add more interactive features inspired by the FDR3 tool [Gib+14].

Improve projection diagram approach. A good layout for the nodes and the edges of a projection diagram is crucial for its readability and accessibility [HMM00]. A next step would be also to adapt the underlying layout algorithm of the projection diagrams so that the nodes (the equivalence classes) are ordered based on the defined projection function. This was already done manually in the diagram shown in Fig. 5.10. We ordered the nodes so that the lower side of the diagram contains the equivalence classes where the cabin door is closed, and the upper side of the figure contains the equivalence classes where the cabin door is open.

We also plan to enhance the projection diagram with interactive features. For instance, it would be desirable to "jump" into an equivalence class and to inspect the individual states which have been merged into it. This could be in particular useful when taking a closer look at equivalence classes that have unexpected outgoing edges, e.g. if the user expected a definite edge, but the equivalence class has a non-definite edge instead. One could jump into the affected class and inspect the states in which an event is not enabled.

Finally, we plan to symbolically construct a projection diagram statically using the built-in constraint solver in ProB [Leu+14] rather than first having to (exhaustively) explore the full state space using the model-checking feature in ProB. This is related to proof-based approaches in [BPS05] and [BL11].

8.2. Conclusion and Discussion

Animation, data visualization, and interactivity are established techniques that can support a user in accomplishing specific objectives. One the one hand, animation is often used in the field of formal methods with the objective of supporting the user in validating formal specifications. On the other hand, data visualization is a well know technique that supports human understanding by providing pictures or diagrams rather than by a substantial amount of numerical or textual data. Moreover, interactivity supports data visualization since users can interact with the visualization, e.g. by changing parameters and seeing the effect. At the beginning of this thesis we defined the following research question:

> "How can animation benefit from interactive data visualization to support the validation process of formal specifications and to make animation techniques more accessible to non-formal method experts?"

In order to answer this question, the starting point of this thesis was to examine the state-of-the-art research for existing visualization techniques in the field of formal methods. Based on the result of the state-of-the-art research and the problems and limitations that were identified, we presented a new graphical environment called BMotionWebfor the rapid creation of *interactive formal prototypes*. An interactive formal prototype combines animation with interactive domain specific visualization. We described the concept of BMotionWeb and gave a detailed description of its implementation and features for the state-based formal methods classical-B and Event-B and the process based formal method CSPM. To illustrate the tool, we integrated it with the ProB animator. However, BMotionWeb can also be integrated with other animation engines. BMotion-Web uses web-technologies for developing its frontend and the visualization template of an interactive formal prototype. This design decision has several advantages. Creating an interactive formal prototype is flexible since external resources, such as SVG images and third party JavaScript libraries, can be reused. The JavaScript language enables the developer to create complex and generic interactive formal prototypes (e.g. with numerous or repeated graphical elements). Finally, an interactive formal prototype can be deployed online and thus be accessed from other devices, such as tablets and mobile phones, and be shared with other stakeholders (e.g. during an online project meeting).

We also presented a visualization approach based on the state space of a formal specification called *projection diagrams* with the main goal of considerably reducing the complexity of state space visualizations and supporting human analysis of the system by highlighting relevant aspects of a formal specification. Although the produced projection diagrams may not be equivalent to the original state space (as far as the sequence of the events are concerned), the projection may reduce the size of the state space while still preserving beneficial information. In particular, the categorization of the edges and equivalence classes proved to be very useful to support the inspection of the diagram and to understand useful properties within the respective formal specification.

The approach is also flexible since the user may adjust the underlying projection function. The possibility of defining an individual projection function enables the user to *query* the full state space and to obtain only the information that the user is interested in (comparable with defining queries for a database). Moreover, we believe that inspecting multiple small projection diagrams (with a manageable number of nodes and transitions) representing different aspects of the specification can be more helpful than inspecting one big state space visualization. One reason for this is that the user can concentrate on a specific aspect of the specification (e.g. on certain variables) or on checking a specific behavior while hiding non-relevant information from the diagram. We also believe that a projection diagram may help to verify properties of the specification which are hard to express as invariants.

In addition, we showed how projection diagrams and other state space visualization approaches such as trace diagrams can be combined with interactive formal prototypes. The direct linking of observers to graphical elements and the strict separation between *query* (query the needed state data from the animation) and *update* (update the graphical element according to some state data) functions is beneficial for this purpose: we only need to determine the observers of the graphical elements to be combined and feed the animation data coming from other tools or approaches (e.g. the state data of a node in a trace diagram or the projection function value of an equivalence class in the projection diagram) into the *update* function of the observers. We believe that combining a projection or trace diagram with an interactive formal prototype affords further advantages. For example, the graphical representation of a specific aspect or behavior of the specification can be helpful for discussing the specification with non-formal method experts and for the further development of the specification. A non-formal method expert can even use the trace and projection diagram features without any knowledge about the notation used in formal methods since the diagrams are produced based on graphical elements.

Finally, we demonstrated the application of BMotionWeb based on various case studies, including interactive systems, industrial case studies and case studies for teaching formal methods. A crucial additional question had come up while we have been working on BMotionWeb and the different case studies: “How can we guarantee that an interactive formal prototype reflects the formal specification correctly?”.

In general, an interactive formal prototype is developed by a human, and humans can make mistakes. For instance, a developer can create a wrong or misleading visual representation or hide an issue in a formal specification in the interactive formal prototype (e.g. faulty behavior). The latter would be a critical mistake since the interactive formal prototype would lead us to believe that the formal specification works as expected although it contains errors in reality. Moreover, a formal specification typically evolves in the development process. Changes made in the formal specification may also affect the interactive formal prototype and it may need to be adapted.

It is hard to guarantee that the developed interactive formal prototype contains no misleading or wrong information and reflects the formal specification correctly. However, what we can do is to support the development of interactive formal prototypes and minimize the mistakes made by humans. As an example, in the course of working on the industrial applications (see Section 7.3), the projection diagram feature proved to be helpful for this purpose. In particular, we have used the projection diagram feature to check if a particular graphical element represents all states of the system properly and to eliminate undesirable behavior in the interactive formal prototype. We also believe that the maintenance of a traceability between the interactive formal prototype and the mathematical elements of a formal specification (e.g. variables or constants) may be helpful when adapting to changes made in the formal specification. Such a traceability could also be generated automatically based on the information coming from observers. For instance, formula observers provide information about which variables, expressions, or predicates are wired to graphical elements. This information could be used to trace graphical elements back to the formal specification. We are also able to determine *implicit traces* discovered via model element relationships [Jas+10]. This includes references to other model elements (e.g. invariants or guards), refinement relationships, or proof obligations.

In summary, we can conclude that we have achieved the thesis goal. The different example applications and the success stories for applying BMotionStudio and BMotionWeb in industry and for teaching formal methods answered our research question and confirmed the benefits of using interactive data visualization in combination with animation. After all, “one picture is worth ten thousand words”.

A State-Of-The-Art Specifications

A.1. Simple Lift System

A.1.1. classical-B

```
MACHINE Lift

DEFINITIONS
 SET_PREF_SHOW_EVENTB_ANY_VALUES==TRUE;
 ANIMATION_FUNCTION == ({ r,c,i | r : groundf..topf &
                         ((c=0 & i=0) or (c=1 & i=3)) } <+
                          ({ r,c,i | r : groundf..topf & Rconv : request &
                             c=1 & i=4 } \/
                           { r,c,i | r : groundf..topf & Rconv=floor & c=0 &
                             ((door = open & i=1) or (door = closed & i=2)) } ) );
    ANIMATION_IMG0 == "LiftEmpty.gif";
    ANIMATION_IMG1 == "LiftOpen.gif";
    ANIMATION_IMG2 == "LiftClosed.gif";
    ANIMATION_IMG3 == "CallButtonOff.gif";
    ANIMATION_IMG4 == "CallButtonOn.gif"

SETS
 Move = {up, down, idle}; Door = {open, closed}

CONSTANTS groundf, topf

PROPERTIES
  topf : INTEGER & groundf : INTEGER &
  groundf = -1 & topf = 1 &  groundf < topf

VARIABLES
  floor, move, door, request

INVARIANT
  floor : groundf..topf & door : Door & move : Move &
  (move : {up,down} => door = closed) & request : POW(groundf .. topf)

INITIALISATION floor := 0 || move := idle || door := closed || request := {}

```

```
OPERATIONS

  move_up =
  PRE floor < topf & move = up
  THEN floor := floor + 1
  END;

  move_down =
  PRE floor > groundf &  move = down
  THEN floor := floor - 1
  END;

  switch_move_up =
  PRE move = idle & door = closed & floor < topf & floor /: request
  THEN move := up
  END;

  switch_move_down =
  PRE move = idle & door = closed & floor > groundf & floor /: request
  THEN move := down
  END;

  switch_move_stop =
  PRE move = up or move = down
  THEN move := idle
  END;

  door_open =
  PRE move = idle & door = closed & floor : request
  THEN door := open
  END;

  door_close =
  PRE move = idle & door = open
  THEN door := closed || request := request \ {floor}
  END;

  send_request =
  ANY f
  WHERE f : groundf .. topf & f /: request
  THEN request := request \/ {f}
  END

END
```

A.1.2. Event-B

Specification

```
context c0

sets Door

constants groundf topf open closed

axioms
  @axm1 groundf : INT
  @axm2 topf : INT
  @axm3 groundf = -1
  @axm4 topf = 1
  @axm5 groundf < topf
  @axm6 partition(Door, {open}, {closed})
end
```

```
context c1 extends c0

sets Move

constants up down idle

axioms
  @axm1 partition(Move, {up}, {down}, {idle})
end
```

```
machine m0 sees c0

variables floor door

invariants
  @inv1 floor : groundf..topf
  @inv2 door : Door

events
  event INITIALISATION
    then
      @act1 floor := 0
      @act2 door := closed
  end

  event move_up
    where
      @grd1 floor < topf
    then
      @act1 floor := floor + 1
  end

  event move_down
    where
      @grd1 floor > groundf
    then
      @act1 floor := floor - 1
  end

  event door_open
    where
      @grd1 door = closed
    then
      @act1 door := open
  end

  event door_close
    where
      @grd1 door = open
    then
      @act1 door := closed
  end
end
```

```
machine m1 refines m0 sees c1

variables floor door move

invariants
  @inv1 move : Move
  @inv2 move : {up,down} => door = closed

events
  event INITIALISATION extends INITIALISATION
    then
      @act3 move := idle
  end

  event move_up extends move_up
    where
      @grd2 move = up
  end

  event move_down extends move_down
    where
      @grd2 move = down
  end

  event door_open extends door_open
    where
      @grd2 move = idle
  end

  event door_close extends door_close
    where
      @grd2 move = idle
  end

  event switch_move_up
    where
      @grd1 move = idle
      @grd2 door = closed
      @grd3 floor < topf
    then
      @act1 move := up
  end

  event switch_move_down
    where
      @grd1 move = idle
      @grd2 door = closed
      @grd3 floor > groundf
    then
      @act1 move := down
  end

  event switch_move_stop
    then
      @act1 move := idle
  end
end
```

```
machine m2 refines m1 sees c1

variables floor door move request

invariants
  @inv1 request : POW(groundf..topf)

events
  event INITIALISATION extends INITIALISATION
    then
      @act4 request := {}
  end

  event move_up extends move_up
  end

  event move_down extends move_down
  end

  event door_open extends door_open
    where
      @grd3 floor : request
  end

  event door_close extends door_close
    then
      @act2 request := request \ {floor}
  end

  event switch_move_up extends switch_move_up
    where
      @grd4 floor /: request
  end

  event switch_move_down extends switch_move_down
    where
      @grd4 floor /: request
  end

  event switch_move_stop extends switch_move_stop
  end

  event send_request
    any f
    where
      @grd1 f : groundf..topf
      @grd2 f /: request
    then
      @act1 request := request \/ {f}
  end
end
```

JeB Graphical Visualization (HTML5 / JavaScript)

```
jeb.animator.init = function() {

  $anim.canvas.width = 500;
  $anim.canvas.height = 650;
  $anim.canvas.style.display = '';

  background.src = 'images/background.jpg';

  background.onload = function() {

    doorOpen.src = 'images/door_open.jpg';
    doorClosed.src = 'images/door_active.jpg';
    requestButtonNotPressed.src = 'images/call_button.gif';
    requestButtonPressed.src = 'images/call_button_pressed.gif';

    $anim.drawImage(background, 0, 0);

    // Draw initial state of door
    doorClosed.onload = function() {
      $anim.drawImage(doorClosed, 192, 228);
    };

    // Draw initial state of request buttons
    requestButtonNotPressed.onload = function() {
      $anim.drawImage(requestButtonNotPressed, 128, 95); // 1
      $anim.drawImage(requestButtonNotPressed, 128, 314); // E
      $anim.drawImage(requestButtonNotPressed, 128, 527); // U1
    };

  };

};

jeb.animator.draw = function() {

  var floor = $var.floor.value;
  var door = $var.door.value;
  var request = $var.request.value;

  // Clear canvas
  $anim.clearRect(0, 0, $anim.canvas.width, $anim.canvas.height);

  // Redraw background
  $anim.drawImage(background, 0, 0);

  // Determine whether the door is
  // closed or open in the current state
  var newDoor;
  if (door === 'open') {
    newDoor = doorOpen;
  } else {
    newDoor = doorClosed;
  }

  // Draw lift cabin state (position and door)
  if (floor.equal(BigInteger(1))) {
    $anim.drawImage(newDoor, 192, 12);
  } else if (floor.equal(BigInteger(0))) {
    $anim.drawImage(newDoor, 192, 228);
  } else if (floor.equal(BigInteger(-1))) {
    $anim.drawImage(newDoor, 192, 444);
  }

  $anim.drawImage(requestButtonNotPressed, 128, 95); // 1
  $anim.drawImage(requestButtonNotPressed, 128, 314); // E
  $anim.drawImage(requestButtonNotPressed, 128, 527); // U1

  // Draw request button states
  for (i = 0; i < request.length; i++) {
    if (request[i].equal(BigInteger(1))) {
      $anim.drawImage(requestButtonPressed, 128, 95); // 1
    } else if (request[i].equal(BigInteger(0))) {
      $anim.drawImage(requestButtonPressed, 128, 314); // E
    } else if (request[i].equal(BigInteger(-1))) {
      $anim.drawImage(requestButtonPressed, 128, 527); // U1
    }
  }

}
```

A.1.3. VDM-SL

```
module Lift

imports from gui_Graphics all

exports all

definitions

values
  topf : int = 1;
  groundf : int = -1;

types
  Door = <open> | <closed>;
  Move = <up> | <down> | <idle>;
  floors = int
    inv floors == floors >= groundf and floors <= topf

state lift of
  position : floors
  door : Door
  move : Move
  request : set of floors
inv mk_lift(position,door,move,request) ==
    move in set {<up>,<down>} => door = <closed> and
    position in set {groundf, ..., topf}
init
  s == s = mk_lift(0,<closed>,<idle>,{})
end

operations

  initialize: () ==> ()
  initialize() ==
  gui_Graphics`initialize();

  move_up: () ==> ()
  move_up() ==
  (position := position + 1;
  gui_Graphics`updateLiftPosition(position))
  pre position < topf and move = <up>;

  move_down: () ==> ()
  move_down() ==
  (position := position - 1;
  gui_Graphics`updateLiftPosition(position))
  pre position > groundf and move = <down>;

  switch_move_up() ==
  move := <up>
  pre move = <idle> and door = <closed> and position < topf and position not in
      set request
  post move = <up>;

  switch_move_down() ==
  move := <down>
  pre move = <idle> and door = <closed> and position > groundf and position not
      in set request
  post move = <down>;

  switch_move_stop() ==
  move := <idle>
  pre move = <up> or move = <down>
  post move = <idle>;

  door_open() ==
  (door := <open>;
  request := request \ {position};
  gui_Graphics`updateDoorState(door = <open>);
  gui_Graphics`removeRequestButton(position))
  pre move = <idle> and door = <closed> and position in set request
  post door = <open>;

  door_close() ==
  (door := <closed>;
  gui_Graphics`updateDoorState(door = <open>))
  pre move = <idle> and door = <open>
  post door = <closed>;

  send_request: int ==> ()
  send_request(f) ==
  (request := request union {f};
  gui_Graphics`addRequestButton(f))
  pre f not in set request;

end Lift
```

```
module gui_Graphics

imports from Lift all

exports all

definitions
```

```

operations

  initialize: () ==> ()
  initialize() == is not yet specified;

  updateLiftPosition: int ==> ()
  updateLiftPosition(newLiftPosition) == is not yet specified;

  updateDoorState: bool ==> ()
  updateDoorState(newDoorState) == is not yet specified;

  addRequestButton: int ==> ()
  addRequestButton(requestButton) == is not yet specified;

  removeRequestButton: int ==> ()
  removeRequestButton(requestButton) == is not yet specified;

end gui_Graphics
```

B
Case Studies, Examples and Applications

B.1. Overview of Examples and Applications

The following two tables give an overview of the case studies and applications implemented with BMotionWeb (Appendix B.1.1) and BMotionStudio (Appendix B.1.2). The tables show the name and a short description of the case study, the formalism used for the corresponding animated formal specification, and if it is available in the examples GitHub repository.

B.1.1. BMotionWeb Case Studies

Name	Formalism	GitHub*	Note
Landing Gear System	Event-B	✓	Challenge from ABZ'14 and described in [Han+14; Lad+15] and in this thesis (see Section 7.3.1).
Hemodialysis Machine	Event-B	✓	Challenge from ABZ'16 and described in [Hoa+16] and in this thesis (see Section 7.3.2).
Advance Smart Grid	Event-B	✓	Developed during the Advance FP7 Project and described in [BK14].
Advance Interlocking System	Event-B	✓	Developed during the Advance FP7 Project and described in [MKS14].
Crossing Bridge	Event-B	✓	-
Cruise Control Device	Event-B	✓	Described in [LL16] and this thesis (see Section 7.2.2).
Simple Lift System	Event-B	✓	Described in this thesis (see Appendix A.1).
Pacman	Event-B	✓	Developed by the student Christoph Heinzen at the University of Düsseldorf.
Lightbot	Event-B	✓	Used as a graduation project and for teaching in a formal methods course at the university of Düsseldorf.
Route Reservation	Event-B	✗	Dominik Hansen used BMotionWeb as part of his research for creating a graphical simulation of train movement and route reservation on a railway topology.
Phonebook Application	classical-B	✓	Described in [LL16] and this thesis (see Section 7.2.1).
Frog Puzzle	classical-B	✓	-
Chess Engine	classical-B	✓	Developed by the student Philip Hoefges at the University of Düsseldorf. Described in [Hoe16].
Five Philosophers Problem	CSPM	✓	-
Bully Algorithm	CSPM	✓	Described in [LDL14].
Level Crossing Gate	CSPM	✓	

*The GitHub Case Study repository is available at
`https://github.com/ladenberger/bmotion-prob-examples`.

B.1.2. BMotionStudio Case Studies

Name	GitHub*	Note
Location Access Controller	✓	Case Study from [Abr10]. Developed by Ivaylo Dobrikov for a course at the University of Düsseldorf.
Interlocking	✓	Case Study from [Abr10]. Described in [Lad10] and used for teaching formal methods at the University of Düsseldorf.
Interlocking	✓	Case Study from the book [Abr10]. Used for teaching formal methods at the University of Düsseldorf.
Mechanical Press	✓	Case Study from the book [Abr10]. Described in [Lad10].
Config Chooser	✓	-
Farmer Puzzle	✓	-
Hanoi	✓	-
Simple Lift System	✓	Described in [LBL09] and [Lad09].
Poker	✓	Developed by the student Ivan Merlin at the University of Düsseldorf.
Postal Puzzle	✓	-
8 Puzzle Game	✓	Described in [Lad09].
Rush Hour	✓	-
Russian Postal	✓	-
S21	✓	Visualization of the German Stuttgart 21 project. Developed by Harald Wiegard at the University of Düsseldorf.
SET	✓	-
Waterboiler	✓	-
LACE	✗	The authors in [Tik+13] and [Bou13] use BMotionStudio to visualize a LACE specification, a DSL developed and used within ASML (`http://www.asml.com`) for controlling lithography machines.
Run-Time Management	✗	The authors in [FSB14] use BMotionStudio as a part of a process for development of a run-time management system.
Public Engagement	✗	Developed by Dana Dghaym, Asieh Salehi, and Colin Snook for the annual "Science Day" in Southampton for public engagement. More information available at `http://wiki.event-b.org/index.php/Rodin_Workshop_2016`.
Railground	✗	Developed by Klaus Reichl and Thomas Fischer for demonstrating the use of Rodin in the field of railway system engineering. Available at `https://github.com/klar42/railground` and described in [RFT16].

*The GitHub Case Study repository is available at `https://github.com/ladenberger/bmotion-prob-examples`.

B.2. Simple Lift System Interactive Formal Prototype

This section presents the interactive formal prototype of a simple lift system. The lift system allows movement of a single lift cabin between a finite number of floors and the opening and closing of the lift cabin door. The user can request the lift on a specific floor by pressing a request button that is installed on each floor. Appendix B.2 presents the visualization template of the simple lift system. The Event-B specification of the simple lift system is shown in Appendix A.1.2.

Visualization Template

```
{
  "id": "lift",
  "name": "Simple lift system",
  "template": "lift.html",
  "model": "model/m2.bcm",
  "autoOpen": [
    "CurrentTrace",
    "Events"
  ]
}
```

Listing B.30: Simple lift system manifest (bmotion.json)

```
<!DOCTYPE html>
<html>
<head>
  <title>Simple lift system</title>
</head>
<body>
  <script src="bms.api.js"></script>
  <script src="lift.js"></script>
  <div bms-svg="lift.svg"></div>
</body>
</html>
```

Listing B.31: Simple lift system HTML template (lift.html)

```
<svg width="220" height="340"
  xmlns="http://www.w3.org/2000/svg">
 <g id="lift_system">
  <g id="lift">
   <rect fill="white" stroke="black"
     height="330" width="100" y="5" x="50"/>
   <rect id="door" fill="gray" stroke="black"
     height="80" width="70" y="245" x="65" />
```

```
    <text fill="black" y="58" x="165">Floor 1</text>
    <text fill="black" y="182" x="165">Floor 0</text>
    <text fill="black" y="290" x="165">Floor -1</text>
   </g>
   <g id="request_buttons">
    <ellipse id="bt_1" data-floor="1"
      ry="11" rx="11" cy="54" cx="22" fill="gray"/>
    <ellipse id="bt_0" data-floor="0"
      ry="11" rx="11" cy="177" cx="22" fill="gray"/>
    <ellipse id="bt_-1" data-floor="-1"
      ry="11" rx="11" cy="285" cx="22" fill="gray"/>
   </g>
  </g>
</svg>
```

Listing B.32: Simple lift system SVG visualization (lift.svg)

```
bms.observe("formula", {
  selector: "#lift",
  formulas: ["floor"],
  translate: true,
  trigger: function(origin, values) {
    var door = origin.find("#door");
    switch (values[0]) {
      case 1:
        door.attr("y", "20");
        break
      case 0:
        door.attr("y", "140");
        break
      case -1:
        door.attr("y", "250");
        break
    }

  }
});

bms.observe("formula", {
  selector: "#door",
  formulas: ["door"],
  trigger: function(origin, values) {
    if (values[0] === 'open') {
      origin.attr("fill", "white");
    } else if (values[0] === 'closed') {
      origin.attr("fill", "gray");
    }
  }
});

bms.observe("set", {
  selector: "#request_buttons",
  set: "request",
  convert: function(element) {
    return "#bt_" + element;
  },
  actions: [
    { attr: "fill", value: "green" }
  ]
});

bms.handler("executeEvent", {
  selector: "#door",
  events: [
    { name: "door_open" },
    { name: "door_close" }
  ]
});

bms.handler("executeEvent", {
  selector: "ellipse[data-floor]",
  events: [{
    name: "send_request",
    predicate: function(origin) {
      return "f=" + origin.attr("data-floor")
    }
  }],
  label: function(origin, event) {
    return "Push button " + event.predicate;
  }
});
```

Listing B.33: Simple lift system observers and event handlers (lift.js)

C
BMotionWeb Groovy Scripting API

```
 package de.bmotion.core;

import java.util.Map;

import groovy.lang.Closure;

public interface IBMotionApi {
 /**
  *
  * Logs the given message on the client side. An arbitrary object can be
  * passed as a message with the assumption that the object is serializable.
  *
  * @param message
  * An arbitrary serializable message object
  */
 public void log(Object message);

 /**
  *
  * Executes an event for the given name.
  *
  * @param name
  * The name of the event that should be executed
  * @return The return value of the event (e.g. classical-B operations may
  * have return values)
  * @throws BMotionException
  */
 public Object executeEvent(String name) throws BMotionException;

 /**
  *
  * Executes an event for the given name and options.
  *
  * @param name
  * The name of the event that should be executed
  * @param options
  * The options for the event (e.g. an additional predicate)
  * @return The return value of the event (e.g. classical-B operations may
  * have return values)
  * @throws BMotionException
  */
 public Object executeEvent(String name, Map<String, String> options) throws
     BMotionException;

 /**
  *
  * Evaluates the given formula in the current state and returns the value.
  *
  * @param formula
  * The formula that should be evaluated in the current state
  * @return The result of the formula
  * @throws BMotionException
  */
 public Object eval(String formula) throws BMotionException;

 /**
  *
  * Evaluates the given formula with options in the current state and returns
  * the value.
  *
  * @param formula
  * The formula that should be evaluated in the current state
  * @param options
  * The options for the evaluation (e.g. translate flag)
  * @return The result of the formula
  * @throws BMotionException
  */
 public Object eval(String formula, Map<String, Object> options) throws
     BMotionException;

 /**
  *
  * Registers a method on the server side.
  *
  * @param name
  * The name of the method.
  * @param func
  * The functional body of the method as a {@link Closure}
  */
 public void registerMethod(String name, Closure<?> func);

 /**
  *
  * Calls a registered method on the server side.
  *
  * @param name
  * The name of the method.
  * @param args
  * The arguments for the method
  * @return The return value of the method
  * @throws BMotionException
  */
 public Object callMethod(String name, Object... args) throws BMotionException;

 /**
  *
  * Returns a list of registered server side methods.
  *
  * @return A list of registered server side methods
  */
 public Map<String, Closure<?>> getMethods();

 /**
  *
  * Returns session related data.
  *
  * @return Session related data
  */
 public Map<String, Object> getSessionData();

 /**
  *
  * Returns tool related data.
  *
  * @return Tool related data
  */
 public Map<String, Object> getToolData();

}
```

Listing C.34: BMotionWeb Groovy Scripting API

```
package de.bmotion.prob;

import de.bmotion.core.IBMotionApi;
import de.prob.model.representation.AbstractModel;
import de.prob.statespace.AnimationSelector;
import de.prob.statespace.Trace;

public interface IProBVisualizationApi extends IBMotionApi {

 /**
  *
  * Returns the ProB representation of the loaded formal specification.
  *
  * @return The formal specification as {@link AbstractModel}
  */
 public AbstractModel getModel();

 /**
  *
  * Returns the current {@link Trace} of the animation.
  *
  * @return The current {@link Trace} of the animation
  */
 public Trace getTrace();

 /**
  *
  * Returns the {@link AnimationSelector} which is the entry point to the
  * ProB GUI.
  *
  * @return The {@link AnimationSelector} which is the entry point to the
  * ProB GUI
  */
 public AnimationSelector getAnimationSelector();

}
```

Listing C.35: BMotionWeb for ProB specific Groovy Scripting API

D Additional Resources

D.1. Groovy Script File for for Landing Gear System

```
def replayTrace = [
  [name: "begin_flying"],
  [name: "toggle_handle_up"],
  [name: "con_stimulate_general_valve"],
  [name: "env_close_analogical_switch"],
  [name: "evn_open_general_valve"],
  [name: "con_stimulate_open_door_valve"],
  [name: "open_valve_door_open"],
  [name: "env_start_open_door", pred: "gr=front"],
  [name: "env_open_door_skip", pred: "gr=front"],
  [name: "env_start_open_door", pred: "gr=right"],
  [name: "env_open_door_skip", pred: "gr=right"],
  [name: "env_start_open_door", pred: "gr=left"],
  [name: "env_open_door_last", pred: "gr=left"],
  [name: "con_stimulate_retract_gear_valve"],
  [name: "open_valve_retract_gear"],
  [name: "env_start_retracting_first", pred: "gr=front"],
  [name: "env_retract_gear_skip", pred: "gr=front"],
  [name: "env_start_retracting_first", pred: "gr=right"],
  [name: "env_retract_gear_skip", pred: "gr=right"],
  [name: "env_start_retracting_first", pred: "gr=left"],
  [name: "env_retract_gear_last", pred: "gr=left"],
  [name: "con_stop_stimulate_retract_gear_valve"],
  [name: "close_valve_retract_gear"],
  [name: "con_stop_stimulate_open_door_valve"],
  [name: "close_valve_door_open"],
  [name: "con_stimulate_close_door_valve"],
  [name: "open_valve_door_close"],
  [name: "env_start_close_door", pred: "gr=front"],
  [name: "env_close_door_skip", pred: "gr=front"],
  [name: "env_start_close_door", pred: "gr=right"],
  [name: "env_close_door_skip", pred: "gr=right"],
  [name: "env_start_close_door", pred: "gr=left"],
  [name: "env_close_door", pred: "gr=left"],
  [name: "con_stop_stimulate_close_door_valve"],
  [name: "close_valve_door_close"],
  [name: "con_stop_stimulate_general_valve"],
  [name: "evn_close_general_valve"],
  [name: "env_open_analogical_switch"]
]

bms.registerMethod("replay", {

  def animationSelector = bms.getAnimationSelector()
  def currentTrace = bms.getTrace()
  def trace = currentTrace

  // Go back to root state
  while(trace.canGoBack()) {
    trace = trace.back()
  }
  animationSelector.traceChange(trace)
  trace = trace.anyEvent() // setup constants
  trace = trace.anyEvent() // initialize machine
  animationSelector.traceChange(trace)

  sleep 1000

  // Replay extension and retraction sequence
  Thread.start {

    replayTrace.each {
      trace = trace.execute(it.name, it.pred ?: "TRUE=TRUE");
      animationSelector.traceChange(trace)
      sleep 1000
    }

  }

});
```

Listing D.36: Groovy script file for landing gear system (script.groovy)

D.2. Groovy Script File for the HD Machine

```
def replayTrace = [
  [name: "User_PressesOn"],
  [name: "CS_LowLevel_StartsTestingCF"],
  [name: "HDMachine_CFTests"],
  [name: "CS_LowLevel_CFTestsOK"],
  [name: "CS_TopLevel_CFTestingSignal2Green", pred: "sgn=CF_TESTING_SIGNAL"],
  [name: "CS_TopLevel_StartsConnectingConcentrate"],
  [name: "User_ConnectsConcentrate"],
  [name: "HDSystem_StartsSettingRP"],
  [name: "HDSystem_StartsPreparingTS"],
  [name: "HDSystem_StartsPreparingHP"],
  [name: "HDSystem_StartsSettingTP"],
  [name: "HDSystem_StartsRinsingDialyzer"],
  [name: "HDSystem_StartsConnectingArterially"],
  [name: "Patient_ConnectsArterially"],
  [name: "HDSystem_PatientConnecting_StartsBFInitiating"],
  [name: "HDSystem_StartsConnectingVenously"],
  [name: "Patient_ConnectsVenously"],
  [name: "HDSystem_PatientConnecting_RestartsBP"],
  [name: "CS_LowLevel_PM_ActivatesLimitWindow", pred: "pm=VP & ll=-100 & lh=500"],
  [name: "CS_LowLevel_BP_On_PatientConnecting_BP_Restarting"],
  [name: "PM_SetsPressure", pred: "pm=VP & prs=550"],
  [name: "CS_LowLevel_PM_PatientConnecting_VP_High"],
  [name: "CS_LowLevel_PM_PatientConnecting_VP_High_Abnormal"],
  [name: "CS_TopLevel_RaisesAlarm"]
]

bms.registerMethod("replay", {

  def animationSelector = bms.getAnimationSelector()
  def currentTrace = bms.getTrace()
  def trace = currentTrace

  // Go back to root state
  while(trace.canGoBack()) {
    trace = trace.back()
  }
  animationSelector.traceChange(trace)
  trace = trace.anyEvent() // setup constants
  trace = trace.anyEvent() // initialize machine
  animationSelector.traceChange(trace)

  sleep 1000

  // Replay extension and retraction sequence
  Thread.start {

    replayTrace.each {
      trace = trace.execute(it.name, it.pred ?: "TRUE=TRUE");
      animationSelector.traceChange(trace)
      sleep 1000
    }

  }

});
```

Listing D.37: Replay trace to validate requirement R5 of the HD machine (script.groovy)

D.3. Structure of Source Code Repository

Name	**Repository***	**Description**
BMotionWeb Common Server	bmotion	General BMotionWeb server module. Includes the animation engine interface for integrating other animation engines.
BMotionWeb ProB	bmotion-prob	ProB specific implementation of animation engine interface of BMotionWeb.
BMotionWeb Front-end	bmotion-frontend	General front-end for BMotionWeb. Includes build scripts for desktop application and online version of BMotionWeb.
BMotionWeb ProB standalone	bmotion-prob-standalone	Build scripts for BMotionWeb for ProB online standalone server.
BMotionWeb ProB examples	bmotion-prob-examples	Examples repository for BMotionWeb for ProB.

*URL of the repository: https://github.com/ladenberger/[repository name].

Bibliography

[Abr+10] Jean-Raymond Abrial et al. “Rodin: an open toolset for modelling and reasoning in Event-B”. In: *STTT* 12.6 (2010), pp. 447–466.

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, June 2010.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996. ISBN: 0-521-49619-5.

[AD94] Rajeev Alur and David L Dill. “A theory of timed automata”. In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[Adv] Advance. *Advanced Design and Verification Environment for Cyber-physical System Engineering*. `http://www.advance-ict.eu/`. Accessed: 2015-12-01.

[Alm+11] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. “An overview of formal methods tools and techniques”. In: *Rigorous Software Development*. Springer, 2011, pp. 15–44.

[Ans73] Francis J Anscombe. “Graphs in statistical analysis”. In: *The American Statistician* 27.1 (1973), pp. 17–21.

[ASU86] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[Ben06a] Jens Bendisposto. “Integration of the ProB model checker into Eclipse”. Bachelor Thesis. Heinrich-Heine-University of Düsseldorf, July 2006.

[Ben06b] Jens Marco Bendisposto. *Integration of the ProB model checker into Eclipse*. Bachelor Thesis. June 2006.

[Ben15] J Bendisposto. “Directed and Distributed Model Checking of B-Specifications”. PhD thesis. Dissertation, University of Düsseldorf, 2015.

[BG11] Radhakisan Baheti and Helen Gill. “Cyber-physical systems”. In: *The impact of control technology* 12 (2011), pp. 161–166.

[Bic+97] Juan Bicarregui, Jeremy Dick, Brian Matthews, and Eoin Woods. “Making the most of formal specification through animation, testing and proof”. In: *Science of Computer Programming* 29.1–2 (1997). {COST} 247, Verification and validation methods for formal descriptions, pp. 53–78. ISSN: 0167-6423.

[BK14] Brett Bicknell and Karim Kanso. *ADVANCE Deliverables: D.2.4 Full Application in the Smart Engery Domain*. 2014.

[BL07] Jens Bendisposto and Michael Leuschel. “A Generic Flash-Based Animation Engine for ProB”. In: *Proceedings of B 2007*. Ed. by Jacques Julliand and Olga Kouchnarenko. Vol. 4355. Lecture Notes in Computer Science. Springer, 2007, pp. 266–269. ISBN: 3-540-68760-2.

[BL11] Jens Bendisposto and Michael Leuschel. “Automatic Flow Analysis for Event-B”. In: *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2011*. Ed. by Dimitra Giannakopoulou and Fernando Orejas. Vol. 6603. Lecture Notes in Computer Science. Springer, 2011, pp. 50–64. ISBN: 3642198104.

[BN98] Milica Barjaktarovic and Michael Nassiff. “The State-of-the-art in Formal Methods”. In: *AFOSR Summer Research technical report for Rome Research Site, AFRL/IFGB. http://www. wetstonetech. com/fm_quest. html* (1998).

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. “D^3 data-driven documents”. In: *Visualization and Computer Graphics, IEEE Transactions on* 17.12 (2011), pp. 2301–2309.

[Bou13] R.C. Boudewijns. “Graphical simulation of the execution of DSL models”. Master Thesis. Technische Universiteit Eindhoven, Oct. 2013.

[BPS05] Didier Bert, Marie-Laure Potet, and Nicolas Stouls. “GeneSyst: A Tool to Reason About Behavioral Aspects of B Event Specifications. Application to Security Properties.” In: *ZB 2005*. 2005, pp. 299–318.

[BR13] Brett Bicknell and José Reis. *ADVANCE Deliverables: D.2.1 Smart Energy Grid Case Study Definition*. 2013.

[BS12] Egon Börger and Robert Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Science & Business Media, 2012.

[BW14] Frédéric Boniol and Virginie Wiels. “The Landing Gear System Case Study”. In: *ABZ 2014: The Landing Gear Case Study: Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014. Proceedings*. Ed. by Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe. Cham: Springer International Publishing, 2014, pp. 1–18. ISBN: 978-3-319-07512-9.

[Cal+08] Francesco Calzolai, Rocco De Nicola, Michele Loreti, and Francesco Tiezzi. “TAPAs: A tool for the analysis of process algebras”. In: *Transactions on Petri Nets and Other Models of Concurrency I*. Springer, 2008, pp. 54–70.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[Cla13] Joy Clark. *Data Visualization in ProB*. Bachelor Thesis. June 2013.

[Cle09] ClearSy. *User Manual of Atelier B 4.0*. English. `tools.clearsy.com/wp-content/uploads/sites/8/resources/User_uk.pdf`. 2009.

[CW96] Edmund M Clarke and Jeannette M Wing. “Formal methods: State of the art and future directions”. In: *ACM Computing Surveys (CSUR)* 28.4 (1996), pp. 626–643.

[Dal04] Nicholas Daley. *Gaffe: Graphical Front-ends for C Animation*. 2004.

[DES] DESTECS. *DESTECS (Design Support and Tooling for Embedded Control Software)*. `http://www.destecs.org/`. Accessed: 2015-12-01.

[Die09] Lukas Diekmann. “Eine neue graphische Benutzeroberfläche für ProB in Eclipse”. Bachelor Thesis. Heinrich-Heine-University of Düsseldorf, 2009.

[Dix+03] Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2003. ISBN: 0130461091.

[DKS09] Nicholas J Dingle, William J Knottenbelt, and Tamas Suto. "PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets". In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (2009), pp. 34–39.

[Dow97] Mark Dowson. "The Ariane 5 Software Failure". In: *SIGSOFT Softw. Eng. Notes* 22.2 (Mar. 1997), pp. 84–. ISSN: 0163-5948.

[ELW98] Robert Eckstein, Marc Loy, and Dave Wood. *Java swing*. O'Reilly & Associates, Inc., 1998.

[Enga] ClearSy System Engineering. *COPP – Platform Screen Doors at the Paris-Chatillon Station*. `http://www.copp.fr/en/`. Accessed: 2015-12-01.

[Engb] ClearSy System Engineering. *DOF1 Animation with Brama*. `http://www.brama.fr/php/demos-brama-dof1-en.php`. Accessed: 2015-12-01.

[Engc] ClearSy System Engineering. *Dof1 - System to Open and Close the Platform Doors on Line 1, SIL4*. `http://www.dof1.eu/en/`. Accessed: 2015-12-01.

[FGG07] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. "CoreASM: An extensible ASM execution engine". In: *Fundamenta Informaticae* 77.1-2 (2007), pp. 71–104.

[Flu04] Bundesstelle für Flugunfalluntersuchung. *Untersuchungbericht*. AX001-1-2/02. May 2004.

[FM11] Ian Fette and Alexey Melnikov. *The websocket protocol*. 2011.

[For] Formal Systems (Europe) Ltd. *Process Behaviour Explorer (ProBE User Manual, version 1.30)*. Available at `http://www.fsel.com/probe_manual.html`.

[FSB14] Asieh Salehi Fathabadi, Colin Snook, and Michael Butler. "Applying an Integrated Modelling Process to Run-time Management of Many-Core Systems". In: *International Conference on Integrated Formal Methods*. Springer. 2014, pp. 120–135.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[Gib+14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. "FDR3 — A Modern Model Checker for CSP". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. 2014, pp. 187–201.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[GS13] Brad Green and Shyam Seshadri. *AngularJS*. "O'Reilly Media, Inc.", 2013.

[Han+14] Dominik Hansen, Lukas Ladenberger, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. "Validation of the ABZ Landing Gear System using ProB". In: *ABZ 2014: The Landing Gear Case Study*. 2014.

[HJL13] Stefan Hallerstede, Michael Jastram, and Lukas Ladenberger. “A Method and Tool for Tracing Requirements into Specifications”. Science of Computer Programming. 2013.

[HL12] Dominik Hansen and Michael Leuschel. “Translating TLA+ to B for Validation with ProB”. In: *Proceedings iFM’2012*. LNCS 7321. Springer, 2012, pp. 24–38.

[HLP13] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. “Validation of Formal Models by Refinement Animation”. In: *Science of Computer Programming* 78.3 (2013), pp. 272–292. ISSN: 0167-6423.

[HMM00] I. Herman, G. Melancon, and M. S. Marshall. “Graph visualization and navigation in information visualization: A survey”. In: *IEEE Transactions on Visualization and Computer Graphics* 6.1 (Jan. 2000), pp. 24–43. ISSN: 1077-2626.

[Hoa+16] Thai Son Hoang, Colin Snook, Lukas Ladenberger, and Michael Butler. “Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*. Ed. by Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro. Cham: Springer International Publishing, 2016, pp. 360–375. ISBN: 978-3-319-33600-8.

[Hoa83] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Commun. ACM* 26.1 (1983), pp. 100–106. ISSN: 0001-0782.

[Hoe16] Philip Hoeges. “A formal-based chess engine for ProB”. Master Thesis. Heinrich-Heine-University of Düsseldorf, Nov. 2016.

[Hop79] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[IL06] Akram Idani and Yves Ledru. “Dynamic graphical {UML} views from formal B specifications”. In: *Information and Software Technology* 48.3 (2006), pp. 154–169. ISSN: 0950-5849.

[IS15] Akram Idani and Nicolas Stouls. “When a Formal Model Rhymes with a Graphical Notation”. English. In: *Software Engineering and Formal Methods*. Ed. by Carlos Canal and Akram Idani. Vol. 8938. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 54–68. ISBN: 978-3-319-15200-4.

[Jas+10] Michael Jastram, Stefan Hallerstede, Michael Leuschel, and Aryldo G Russo Jr. “An Approach of Requirements Tracing in Formal Refinement”. In: *VSTTE*. Vol. 6217. Lecture Notes in Computer Science. Springer, 2010, pp. 97–111. ISBN: 978-3-642-15056-2.

[Jen14] et al. Jens Bendisposto. *ADVANCE Deliverables: D.4.4 Method and tools for simulation and testing III*. 2014.

[Jon86] Cliff B Jones. *Systematic software development using VDM*. Vol. 2. Prentice-Hall Englewood Cliffs, NJ, 1986.

[Koe+07] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in action*. Vol. 1. Manning, 2007.

[Lad+15] Lukas Ladenberger, Dominik Hansen, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. “Validation of the ABZ landing gear system using ProB”. English. In: *International Journal on Software Tools for Technology Transfer* (2015), pp. 1–17. ISSN: 1433-2779.

[Lad09] Lukas Ladenberger. “A Visual Editor for B-Animations”. Bachelor Thesis. Heinrich-Heine-University of Düsseldorf, Jan. 2009.

[Lad10] Lukas Ladenberger. “Industrial Applications of BMotionStudio”. Master Thesis. Heinrich-Heine-University of Düsseldorf, Sept. 2010.

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[LB03] Michael Leuschel and Michael Butler. “ProB: A Model Checker for B”. In: *FME*. Ed. by Araki Keijiro, Stefania Gnesi, and Mandrio Dino. Vol. 2805. Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 855–874. ISBN: 3-540-40828-2.

[LB08] Michael Leuschel and Michael Butler. “ProB: An Automated Analysis Toolset for the B Method”. In: *STTT* 10.2 (2008), pp. 185–203.

[LBL09] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. “Visualising Event-B Models with B-Motion Studio”. In: *Formal Methods for Industrial Critical Systems: 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*. Ed. by María Alpuente, Byron Cook, and Christophe Joubert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 202–204. ISBN: 978-3-642-04570-7.

[LDL14] Lukas Ladenberger, Ivaylo Dobrikov, and Michael Leuschel. “An Approach for Creating Domain Specific Visualisations of CSP Models”. In: *HOFM 2014*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. LNCS. 2014.

[Leu+01] Michael Leuschel, Laksono Adhianto, Michael Butler, Carla Ferreira, and Leonid Mikhailov. “Animation and Model Checking of CSP and B using Prolog Technology”. In: *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic*. Ed. by Michael Leuschel, Andreas Podelski, C.R. Ramakrishnan, and Ulrich Ultes-Nitsche. Sept. 2001, pp. 97–109.

[Leu+08] Michael Leuschel, Mireille Samia, Jens Bendisposto, and Li Luo. “Easy Graphical Animation and Formula Viewing for Teaching B”. In: *The B Method: from Research to Teaching*. Ed. by C. Attiogbé and H. Habrias. Lina, 2008, pp. 17–32.

[Leu+14] Michael Leuschel, Jens Bendisposto, Ivaylo Dobrikov, Sebastian Krings, and Daniel Plagge. “From Animation to Data Validation: The ProB Constraint Solver 10 Years On”. In: *Formal Methods Applied to Complex Systems: Implementation of the B Method*. Ed. by Jean-Louis Boulanger. Hoboken, NJ: Wiley ISTE, 2014. Chap. Chapter 14, pp. 427–446.

[LF08] Michael Leuschel and Marc Fontaine. “Probing the Depths of CSP-M: A new FDR-compliant Validation Tool”. In: *ICFEM 2008* (2008), pp. 278–297.

[LL15] Lukas Ladenberger and Michael Leuschel. “Mastering the Visualization of Larger State Spaces with Projection Diagrams”. In: *Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*. Ed. by Michael Butler, Sylvain Conchon, and Fatiha Zaïdi. Cham: Springer International Publishing, 2015, pp. 153–169. ISBN: 978-3-319-25423-4.

[LL16] Lukas Ladenberger and Michael Leuschel. “BMotionWeb: A Tool for Rapid Creation of Formal Prototypes”. In: *Proceedings SEFM’16*. Vol. 9763. LNCS. Springer, 2016.

[LPU02] B. Legeard, F. Peureux, and Mark Utting. “Automated Boundary Testing from Z and B”. In: *Proceedings FME’02*. Ed. by L.-H. Eriksson and P. Lindsay. LNCS 2391. Springer-Verlag, 2002, pp. 21–40.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. “UPPAAL in a nutshell”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 1.1 (1997), pp. 134–152.

[LT05] Michael Leuschel and Edd Turner. “Visualising Larger State Spaces in ProB”. In: *ZB*. Ed. by Helen Treharne, Steve King, Martin Henson, and Steve Schneider. Vol. 3455. Lecture Notes in Computer Science. Springer-Verlag, Nov. 2005, pp. 6–23. ISBN: 3-540-25559-1.

[Mas+14] Paolo Masci, Patrick Oladimeji, Paul Curzon, and Harold Thimbleby. “Tool demo: Using PVSio-web to demonstrate software issues in medical user interfaces”. In: *4th International Symposium on Foundations of Healthcare Information Engineering and Systems (FHIES2014)*. 2014.

[Mas+15a] Paolo Masci, Piergiuseppe Mallozzi, Francesco Luca De Angelis, Giovanna Di Marzo Serugendo, and Paul Curzon. “Using PVSio-web and SAPERE for rapid prototyping of user interfaces in Integrated Clinical Environments”. In: *submitted to Verisure2015, Workshop on Verification and Assurance, co-located with CAV2015*. 2015.

[Mas+15b] Paolo Masci, Patrick Oladimeji, Paul Curzon, and Harold Thimbleby. “PVSio-web 2.0: Joining PVS to Human-Computer Interaction”. In: *27th International Conference on Computer Aided Verification (CAV2015)*. Tool and application examples available at http://www.pvsioweb.org. Springer, 2015.

[Mas15] A. Mashkoor. “The Hemodialysis Machine Case Study”. `http://www.cdcc.faw.jku.at/ABZ2016/HD-CaseStudy.pdf`. 2015.

[Mej13] Fernando Mejia. *ADVANCE Deliverables: D.1.1 Railway Case Study Definition*. 2013.

[Mét] C. Métayer. *AnimB Homepage*. `http://www.animb.org/`. Accessed: 2015-12-01.

[Mil89] Robin Milner. *Communication and concurrency*. Vol. 84. Prentice hall New York etc., 1989.

[MK13] Fernando Mejia and Minh-Thang Khuu. *ADVANCE Deliverables: D.1.3 Intermediate Report on Application on RailWay Domain*. 2013.

[MKS14] Fernando Mejia, Minh-Thang Khuu, and Asieh Salehi. *ADVANCE Deliverables: D.1.5 Final Report on Application in Railway Domain*. 2014.

[Mun05] César A Munoz. “PVSio Reference Manual”. In: *National Institute of Aerospace (NIA), Formal Methods Group* 100 (2005).

[Mun14] Tamara Munzner. *Visualization Analysis and Design*. CRC Press, 2014.

[NLL12] ClausBallegaard Nielsen, Kenneth Lausdahl, and PeterGorm Larsen. “Combining VDM with Executable Code”. English. In: *Abstract State Machines, Alloy, B, VDM, and Z*. Ed. by John Derrick et al. Vol. 7316. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 266–279. ISBN: 978-3-642-30884-0.

[Oda+15] Tomohiro Oda, Yasuhiro Yamamoto, Kumiyo Nakakoji, Keijiro Araki, and Peter Gorm Larsen. “VDM Animation for a Wider Range of Stakeholders”. In: *GRACE TECHNICAL REPORTS* (2015), p. 18.

[Ola+13] Patrick Oladimeji, Paolo Masci, Paul Curzon, and Harold Thimbleby. “PVSio-web: a tool for rapid prototyping device user interfaces in PVS”. In: *FMIS2013* (2013).

[ORS92] Sam Owre, John M Rushby, and Natarajan Shankar. “PVS: A prototype verification system”. In: *Automated Deduction—CADE-11*. Springer, 1992, pp. 748–752.

[Pau15] Ian Paul. *So long, Flash! YouTube now defaults to HTML5 on the web*. `http://www.pcworld.com/article/2876307/so-long-flash-youtube-now-defaults-to-html5-on-the-web.html`. Accessed: 2015-12-01. Jan. 2015.

[Pel08] Radek Pelánek. “Fighting state space explosion: Review and evaluation”. In: *Formal Methods for Industrial Critical Systems*. Springer, 2008, pp. 37–52.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981. ISBN: 0136619835.

[PL07] Daniel Plagge and Michael Leuschel. “Validating Z Specifications using the ProB Animator and Model Checker”. In: *Integrated Formal Methods*. Ed. by J. Davies and J. Gibbons. Vol. 4591. Lecture Notes in Computer Science. Springer-Verlag, 2007, pp. 480–500.

[PL10] Daniel Plagge and Michael Leuschel. “Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more”. In: *Software Tools for Technology Transfer (STTT)* 12.1 (Feb. 2010), pp. 9–21. ISSN: 1433-2779.

[Pnu77] Amir Pnueli. “The temporal logic of programs”. In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 46–57.

[Pre08] A.J. Pretorius. *Visualization of State Transition Graphs*. 2008.

[QSu] W3Techs.com. Q-Success. *Usage of Flash for websites*. `http://w3techs.com/technologies/history_overview/client_side_language/all`. Accessed: 2015-12-01.

[RFT16] Klaus Reichl, Tomas Fischer, and Peter Tummeltshammer. "Using Formal Methods for Verification and Validation in Railway". In: *Tests and Proofs: 10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings*. Ed. by K. Bernhard Aichernig and A. Carlo Furia. Cham: Springer International Publishing, 2016, pp. 3–13. ISBN: 978-3-319-41135-4.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN: 0136744095.

[Ros10] A.W. Roscoe. *Understanding Concurrent Systems*. 1st. New York, NY, USA: Springer-Verlag New York, Inc., 2010. ISBN: 9781848822573.

[SA11] Bryan Scattergood and Philip Armstrong. *CSP-M: A Reference Manual*. Jan. 2011.

[SA14] Wen Su and Jean-Raymond Abrial. "Aircraft Landing Gear System: Approaches with Event-B to the Modeling of an Industrial System". In: *ABZ 2014: The Landing Gear Case Study*. Springer, 2014, pp. 19–35.

[SA92] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[Sch01] Steve Schneider. *The B-method: An introduction*. Palgrave Oxford, 2001.

[Ser06] Thierry Servat. "Brama: A new graphic animation tool for B models". In: *B 2007: Formal Specification and Development in B*. Springer, 2006, pp. 274–276.

[Sno14] C. Snook. "iUML-B Statemachines". In: *Proceedings of the Rodin Workshop 2014*. `http://eprints.soton.ac.uk/365301/`. Toulouse, France, 2014.

[Som06] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321313798.

[Spe04a] P. van der Spek. *The Overture Project: Towards an open source toolset*. Tech. rep. Delft University of Technology, Jan. 2004, p. 122.

[Spe04b] P. van der Spek. "The Overture Project: Designing an Open Source Tool Set". MA thesis. Delft University of Technology, Aug. 2004, p. 239.

[STU] STUPS. *ProB Java API*. `http://www.stups.hhu.de/ProB/ProB_Java_API`. Accessed: 2016-08-04.

[Sun+09a] Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. "Integrating Specification and Programs for System Modeling and Verification". In: *Proceedings TASE '09*. Ed. by Wei-Ngan Chin and Shengchao Qin. IEEE Computer Society, 2009, pp. 127–135.

[Sun+09b] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. "PAT: Towards Flexible Verification under Fairness". In: *Computer Aided Verification*. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 709–714.

[TG83] Edward R Tufte and PR Graves-Morris. *The visual display of quantitative information*. Vol. 2. 9. Graphics press Cheshire, CT, 1983.

[Tik+13] Ulyana Tikhonova, Maarten Manders, Mark van den Brand, Suzana Andova, and Tom Verhoeff. "Applying Model Transformation and Event-B for Specifying an Industrial DSL." In: *MoDeVVa@ MoDELS*. 2013, pp. 41–50.

[Tol11] Andriy Tolstoy. "Visualisierung von LTL-Gegenbeispielen". Master Thesis. Heinrich-Heine-University of Düsseldorf, Dec. 2011.

[Val98] Antti Valmari. "The state explosion problem". In: *Lectures on Petri nets I: Basic models*. Springer, 1998, pp. 429–528.

[Vau01] Steven J. Vaughan-Nichols. *Flash is dead. Long live HTML5.* `http://www.zdnet.com/article/flash-is-dead-long-live-html5`. Accessed: 2015-12-01. Nov. 2001.

[W3C11] W3C SVG Working Group. *Scalable Vector Graphics (SVG) 1.1 (Second Edition).* http://www.w3.org/TR/SVG11/. Aug. 2011.

[W3C14] W3C SVG Working Group. *HTML5, A vocabulary and associated APIs for HTML and XHTML.* http://www.w3.org/TR/html5/. Oct. 2014.

[War12] Colin Ware. *Information visualization: perception for design.* Elsevier, 2012.

[Yan13] Faqing Yang. "A Simulation Framework for the Validation of Event-B Specifications". PhD thesis. Université de Lorraine, 2013.

[ZGS14] Simone Zenzaro, Vincenzo Gervasi, and Jacopo Soldani. "WebASM: An Abstract State Machine Execution Environment for the Web". English. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z.* Ed. by Yamine Ait Ameur and Klaus-Dieter Schewe. Vol. 8477. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 216–221. ISBN: 978-3-662-43651-6.

List of Figures

List of Tables

List of Listings

List of Algorithms