University of Magdeburg

Faculty of Computer Science

Bachelor Thesis

Classification of Refinement Mechanisms in Formal Methods

Author:

Benjamin Radau

August 7, 2015

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Workgroup Databases and Software Engineering

M.Sc. Fabian Benduhn
Workgroup Databases and Software Engineering

Radau, Benjamin:
Classification of Refinement Mechanisms in Formal Methods
Bachelor Thesis, University of Magdeburg, 2015.

Abstract

Formal methods are a promising way to create software-models in a mathematically rooted way to keep them formal and analyzable. Stepwise refinement adds the possibility to successively refine an abstract specification to a more concrete one and finally to executable program code. A large amount of different formal methods make it difficult to choose the right one for a given problem. In order to help with this problem and to create a starting point in the selection process, this bachelor thesis is introducing a taxonomy and applies three popular formal methods to it. This taxonomy classifies and evaluates those in different aspects. Three selected formal methods are introduced and a common example is presented to show each one in practical use. In the last step it is evaluated and rated how well these formal methods and their refinement mechanisms fit in the presented taxonomy.

Formale Methoden sind eine vielversprechende Möglichkeit Softwaremodelle zu erstellen, die eine mathematische Grundlage haben und dadurch formal und analysierbar bleiben. Schrittweise Verfeinerung gibt die Möglichkeit, eine abstrakte Spezifikation nach und nach zu konkretisieren und letztendlich ausführbaren Programmcode zu erzeugen. Eine große Anzahl an verschiedenen formalen Methoden kann es schwierig machen, eine zu einem Problem passende auszuwählen. Um bei diesem Problem zu helfen und um einen Einstiegspunkt bei diesem Auswahlprozess zu geben, stellt diese Bachelorarbeit eine Taxonomie vor, welche auf drei populäre formale Methoden angewandt wird. Diese Taxonomie klassifiziert und bewertet diese anhand mehrerer Aspekte. Drei ausgewählte formale Methoden werden vorgestellt und ein gemeinsames Beispiel zeigt die praktische Anwendung dieser. Es wird untersucht und bewertet, wie gut die vorgestellten formalen Methoden und ihre Verfeinerungsmechanismen in die vorgestellte Taxonomie passen.

Contents

List of Figures

List of Tables

1. Introduction

Computer software affects almost every aspect in our daily life and thus software engineering has become a more and more prominent exercise. As Foster states, “Software engineering is the process by which software systems are investigated, planned, modeled, developed, implemented and managed [...]. The ultimate objective is the provision or improvement of desirable conveniences and the enhancement of productivity within the related problem domain” [2].
Software engineering has been considered an explicit theoretical process by many, while a rather technical exercise to others. The former point of view focuses on mathematical semantics and provability, while the latter develops informal models like Unified Modeling Language diagrams or workflow diagrams to describe their software. Both have their benefits and drawbacks. Informal models can generally not be proven on correctness and some software problems cannot be described in purely mathematical terms.
One approach to close this gap between the two views is the usage of formal methods. Those try to give provability to their models as well as a good readability for everybody involved in the software engineering process. According to Alagar, “Formal specification methods use languages with mathematically defined syntax and semantics and offer methods to describe systems and their properties” [3].
Even as early as 1971, Wirth stated that program construction would consist of a sequence of refinement steps and that the degree of modularity obtained in this way would determine the ease or difficulty with which a program can be adapted to changes or extensions [4]. Model refinement allows to build a software model gradually by making the next model more detailed and more precise than its predecessor.
Trying to combine the refinement with formal methods, several model-based refinement methods have been proposed in the literature. These methods apply the idea of (stepwise) refinement to formal specifications: an abstract specification is successively refined to a more concrete specification and finally to executable program code. On each refinement level, the system can be analyzed and potential errors can be found.

The different formal methods differ in many aspects such as their underlying formalism, their notion of refinement and their support for different analysis techniques.
There are also several refinement supporting formal methods out there, which often use different approaches on how the refinement mechanisms work in each corresponding modeling language. Therefore it is not easy to decide which formal method is the most appropriate for a given problem.

The major goal of this thesis is to create a starting point for potential users to choose the right refinement supporting formal method. This thesis makes the following contributions:

- A taxonomy is proposed in which existing formal-based refinement methods can be evaluated into.
- Three selected formal methods and their refinement mechanisms are introduced.
- A common example is provided to give an advanced overview of differences between notions and notations of the formal methods.
- The applicability of the taxonomy on the presented formal methods is evaluated.

Structure of the Thesis

The following Chapter 2 explains the basics required to understand this work. The common concepts for all formal methods will be introduced and the term *refinement* will be explained. Chapter 3 describes the previously mentioned taxonomy of refinement mechanisms and the formal methods Event-B, ASM and Z will be introduced as well. There will be a brief insight into the history of each method, followed by a summary of their key elements and distinguishing features. Furthermore it will be surveyed, whether tool support is available and to what extend it assists the user in practical use. Also parts of the common example will be described in the respective formal method. This chapter finishes with the results and evaluation of the introduced formal methods, regarding their applicability on the proposed taxonomy. Chapter 4 gives an overview of related work, which may have different approaches to compare formal methods or extend the results of this thesis. The last chapter concludes this work.

2. Background

Stepwise refinement in formal methods can be used to create software in a formal and structured manner. To understand this thesis, it is important to understand the idea and the purpose of formal methods and the notion of refinement. Any software project goes through the software life-cycle to distinguish the individual tasks necessary to complete the development process. This life-cycle contains the following steps: *Requirements analysis*, *system specification*, *design*, *implementation*, *test and integration* and *support and enhancement* [5]. The requirements analysis and system specification define the customer needs and requirements of the system such as performance, reliability and most important ***what*** the system does. The design phase explores ways to fulfill all those previously defined requirements and needs and is the crucial bridge between the first two phases and the implementation phase, which is less focused on what to accomplish rather than ***how*** to accomplish the requirements in executable program code. The importance in all of this lies in the monetary aspect. A big bulk of the costs of a software project lies in fixing the errors during the implementation and test phases. However, it has been investigated that a very large percentage of these errors have their origin in false or insufficient specifications, often due to economies in the specification phase [6, 7]. To reduce these cost-inefficient implementation and testing phases it is all the more important to have an error free specification.

2.1 Formal methods

"Formal specifications use mathematical notation to describe in a precise way the properties which an [...] system must have, without unduly constraining the way in which these properties are achieved" [8]. Formal methods can describe *exactly* what a system does without having someone to interpret some program code or naturally written description. A formal specification can be used for all following tasks of the software development. It can be used as a reference for programmers or the program documentation and it gives the advantage of testing the products results with formal proven test scenarios. Not being dependent on the program code, the formal specification can be created in an early stage and it can be used as a central medium of communication for all people involved in the development process. Formal methods use formal notations which distinguish them from less formal models such as data flow or activity diagrams. Formal semantics add an undoubted meaning to a given model with little to none scope for interpretation.

2.2 Refinement

Development often proceeds from an abstract specification to a detailed design. This process is called refinement. Both the specification and the design are models, but the specification is closer to the users' view, while the design is closer to an executable program [9]. If both models are expressed in mathematics, formal reasoning can be used to check that the design faithfully expresses the intent of the specification. It can be mathematically proven that the specification and the design are two models of the same thing. This ability to check the correctness of design steps is one of the distinguishing features of a formal method. A design is correct if it has all the properties of the specification, however it usually has additional properties as well. This relation is expressed precisely by logical implication. The predicate that describes the design must imply the predicate that describes the specification.

Example: The specification requires an increase of a variable x ($x' > x$). The increase is proposed by adding 1 ($x' = x + 1$). This design should imply the specification: $x' = x + 1 => x' > x$ This implication is obviously true and so the refinement is correct. $x' = x + 1$ refines $x' > x$.

2.3 Common Example

To provide a further insight into each formal method, a common example will be used to describe the individual formal methods whenever it is possible. Their general semantics and syntax as well as chosen refinement steps will be explained by using this example. This basic example will contain a simple traffic light with two states as seen in Figure 2.1. In the first state the red light is turned on, while the green light is turned off. The second state is the other way. The states shall be changed by some mechanics. The state changing mechanics may be a simple timer alternating the states but the exact mechanics may be different from method to method.

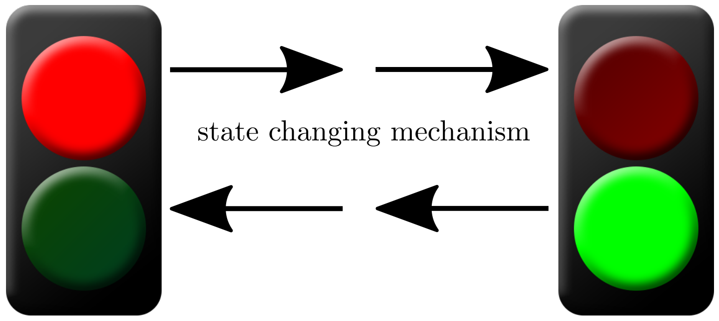


Figure 2.1: Simple traffic light

The refinement on this example will also differ between different formal methods in order to focus on their specific features. The example can be extended or refined by adding another set of lights for an intersection or adding multiple state changing mechanics (e.g. dependency on other lights, emergency vehicles or buses).

3. Survey of Model-Based Refinement Methods

3.1 Taxonomy of refinement mechanisms in formal methods

There are many formal methods that support the possibility of refinement, but not all of them are equally qualified to suffice the practical use in every aspect. In order to choose a formal method which meets the requirements of the desired features, several selection criteria can be taken into account. The factor in learning the notation and the semantics of a language as well as the availability of computer aided tools for a specific language may be critical for the selection process. This issue will be taken care of in the following chapter for each described method. The taxonomy presented here focuses more on the underlying features of the formal methods. Figure 3.1 shows the three main columns expressiveness, strictness and correctness. All formal methods have this three aspects in common yet there are differences in specific characteristics.

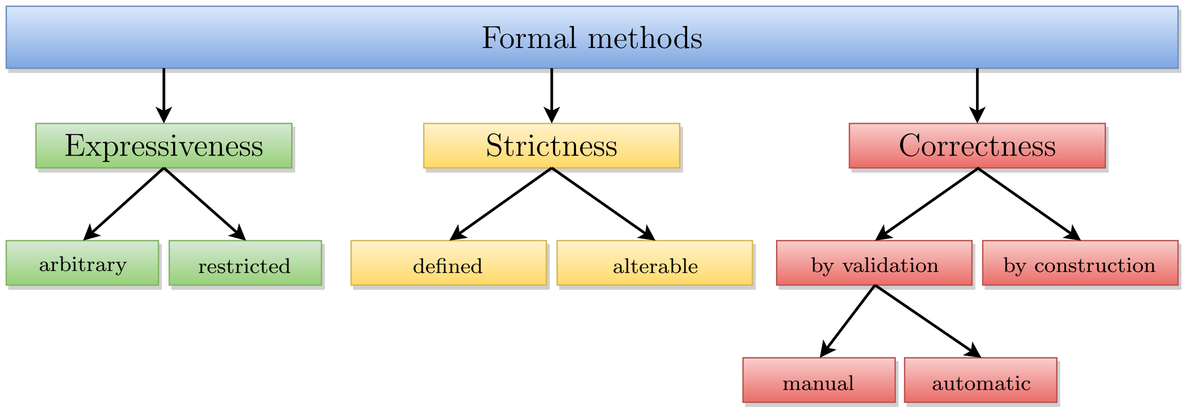


Figure 3.1: A taxonomy for refinement mechanisms in formal methods

3.1.1 Expressiveness

Probably the majority of formal methods have the possibility to design a sequential program, but may not be able to handle concurrent or reactive software. It has to be examined whether some methods have the potential to adapt to unexpected situations, or whether they are restricted to specific software types. In this context, a formal method can be arbitrary or restricted regarding their refinement methods.

3.1.1.1 Restricted

Restricted refinement in formal methods can be described as the inability to express certain model extending intentions, either restricted by possible refinement schemes or the formal methods language by itself. Taking the traffic light as example, there is one simple light alternating between a red and a green state in a given interval t (see Figure 3.2a). The extension or the refinement of the model shall add a button which immediately switches to the red state through an interrupt i when pressed (see Figure 3.2b). Some formal methods may not allow or support this refinement, e.g. because external interrupts are not included in that specific formal method. Other reasons could be, that the model requires certain prerequisites or even alteration in its system in order to make the refinement possible.

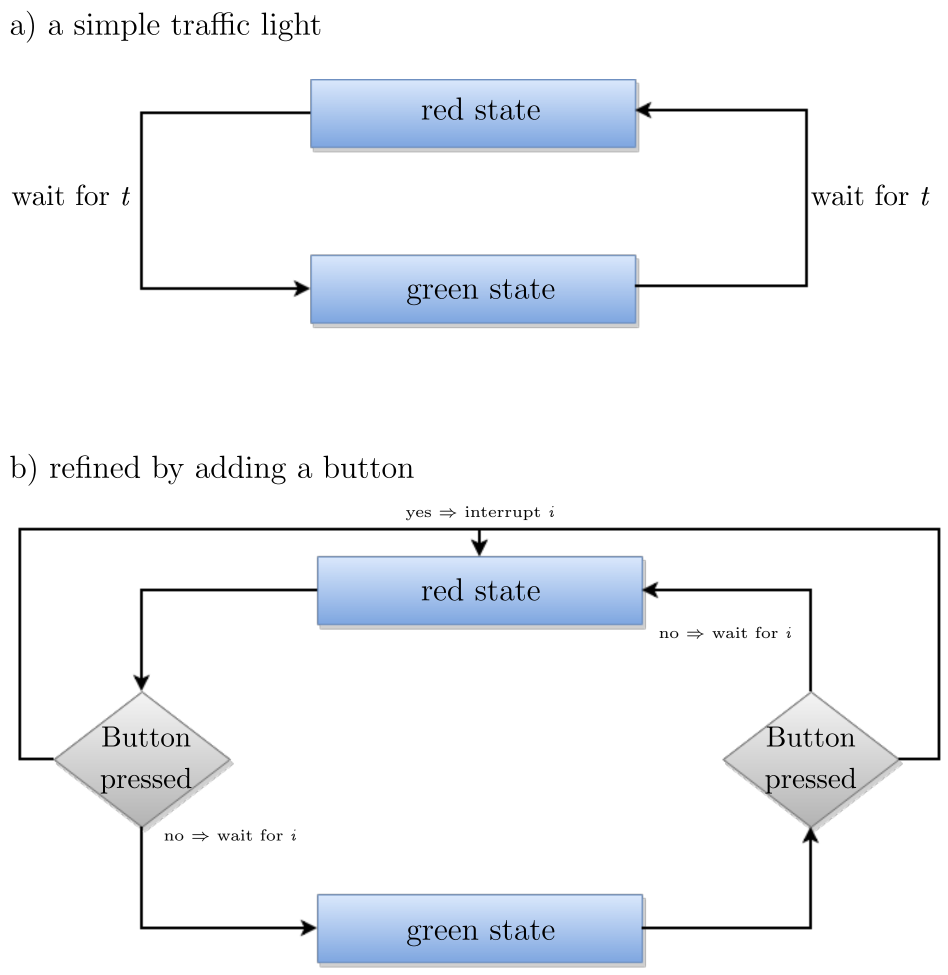


Figure 3.2: Example of a possible refinement of the traffic light

3.1.1.2 Arbitrary

In contrast to the restricted refinement, the arbitrary refinement does not need special prerequisites. A model can be refined whenever it is required to do, so no special sequence has to be taken into account. The example from Figure 3.2 could be extended by an extra set of traffic lights, managing a crossroad. In doing so, the arbitrary refinement would allow to add the second set of lights first and then the button or vice versa.

3.1.2 Strictness

While not directly linked to the classical refinement term, the possibility to adapt to specific needs or circumstances of a specification, which may require altering or extending the given foundation of a formal method, can be seen as a kind of refinement.

Something that previously was not possible to model can be modeled after the expansion. Most formal methods however do not support an alteration of its syntax and semantics. The option to add non-specified elements to a language comes with difficulties. An analogy to this is the usage of a dialect. To some people it is natural to use certain words or phrases in a dialect, whereas other people are not necessarily able to understand them. Same goes for the language of formal methods. The strictness of the usage can be classified in the two simple characteristics: They allow alteration or they do not.

3.1.3 Correctness

A crucial part of using formal methods is the possibility to prove the correctness of a given design. Whereas most formal models can be proven by validation, stepwise refinement adds another possible way of assuring correctness of a design. The refinement can be restricted to a certain amount of pre-validated operations which ensure the correctness of the outcome. The formal methods can establish correctness by validation or correctness by construction.

3.1.3.1 Correctness by validation

Each step of refinement may represent a different level of abstraction or a different view on the design. Therefore it is not always possible to see whether a refined model is truly a representation of the original behavior. There a several approaches to validate a refined model. Validating a refined model can be done manually or (semi-)automatically. Manual validation however has a few drawbacks. It is usually error-prone, time-consuming and expensive. One error in a refinement step can affect all other sequential steps [10].

3.1.3.2 Correctness by construction

Stepwise refinement can be restricted, as mentioned in Section 3.1.1.1. One restriction might be, that only certain specific refinement mechanisms can be applied to a model. This mechanisms may already be proven to be valid after being applied to the more abstract model. If a model is already validated, this mechanisms ensure the consistency for the refined model. For example, there exists a set of colors for the traffic light model, which contains the colors red and green. Adding new colors, which may be needed later (e.g. yellow), is a simple predefined action of set-extension with items of the same type and thus a re-validation can be omitted.

3.1.4 Tool support

The tool support is not directly part of this taxonomy as it overlaps with all previous sections. However, for a potential user of a formal method, tool support is one of the first things that comes to mind whether to choose the formal method or not. There can be tool support helping the user with the construction of the model (e.g. via graphical user interface). It is also possible to ensure strictness with built-in syntax and semantics checks. There are also tools, which help to validate the model or there can be no tool support at all. Since the tool support can be very crucial for the user, the following evaluation of the different formal methods will also take this aspect into consideration wherever it is possible.

3.2 Event-B

Event-B is a formal method for modeling discrete systems explicitly based on the premise of refinement [1]. Both, Event-B and its predecessor, the B Method, have been used to model large scale industrial projects e.g. the VAL shuttle for Roissy Charles de Gaulle airport in France [11]. The main purpose of an Event-B model is reasoning, as this is the key to understand complex models. “Reasoning about complex models should not happen accidentally but needs systematic support within the modelling method. [...] Proof obligations serve to reason about a model and to provide meaning” [12]. This chapter explains the basic formal notation and the fundamental concepts of the Event-B modeling language, its refinement and its place in the proposed taxonomy.

3.2.1 Overview

The B method is a state-based method, which combines set theory, predicate logic and substitution [13]. Models in Event-B are described with two basic constructs: contexts and machines. Contexts describe the static part of a model like sets, axioms and constants, whereas machines describe the dynamic part of an Event-B model. Machines may contain events, variables, theorems, variants and invariants. A context in the example is the set of possible colors the traffic light can show: $\{red, green, yellow\}$. An example of a corresponding machine for the traffic lights can be found below.

Variables $v = v_1, ..., v_a$ define the state of a machine and are constrained by invariants $I(v)$. State changes are described by Events $E(v)$. Each event is composed of a guard $G(t, v)$ and an action $x := S(t, v)$, where $t = t_1, ..., t_b$ are parameters the event may contain and $x = x_1, ..., x_c$ are the variables it may change. The guard states the necessary condition under which the event may occur and the action describes how the state variables evolve when the event occurs. The basic event notion $E(v)$ is one of the following:

```
E(v)   ≙   any t when
               G(t, v)
           then
               x := S(t, v)
           end
```

or

```
E(v)   ≙   begin
               x := S(v)
           end
```

The short form can be used when the event does not have parameters. It is important to note, that all assignments of an action $x := S(t, v)$ occur simultaneously. In order to keep consistency, whenever a variable value changes, invariants are supposed to hold and should be proven for each event - invariant combination.

To be more concrete, the traffic light example, which is similar to the one in the Event-B handbook [14], adds a descriptive view on the theoretical notation. This specific example adds a pedestrian crossing in addition to a simple traffic light. There are two simple Boolean variables: *traffic_flow* and *pedestrians_flow*.

```
set_pedestrians_flow   ≙   when
                               pedestrians_flow = FALSE
                           then
                               traffic_flow = TRUE
                           end
```

A more common notation using the Rodin tool is seen below. The excerpt from a possible traffic light controller in Event-B has two variables *traffic_flow* and *pedestrians_flow*, which each describe whether the cars or the pedestrians are allowed to move. The invariant **inv1** and **inv2** restrict these variables to the Boolean type. There is an event called *set_pedestrians_flow* that changes the value of *pedestrians_flow* to *true* in action **act1** when the guard **grd1** is *TRUE* thus when the flow of cars has stopped ($traffic\_flow = FALSE$).
This machine will be refined in the next section.

```
MACHINE
   $trafficLights0$
VARIABLES
   $traffic\_flow$
   $pedestrians\_flow$
INVARIANTS
   inv1:
     $traffic\_flow \in BOOL$
   inv2:
     $pedestrians\_flow \in BOOL$
EVENTS
   Event
    $set\_pedestrians\_flow \mathrel{\widehat{=}}$
    when
      grd1:
        $traffic\_flow = FALSE$
    then
      act1:
        $pedestrians\_flow = TRUE$
```

3.2.2 Refinement in Event-B

A machine M^* can refine a machine M. M is called an abstract machine and M^* is called the concrete machine. The refinement strengthens invariants and can add details to a model (see Figure 3.3). The state of the abstract machine is linked to the state of the concrete machine by a gluing invariant $J(v, v^*)$ where v are the variables of the abstract machine and v^* are the variables of the concrete machine. This gluing invariant helps to verify the refinement between models [15]. Each abstract event $E(v)$ is refined by a concrete event $E^*(v^*)$.

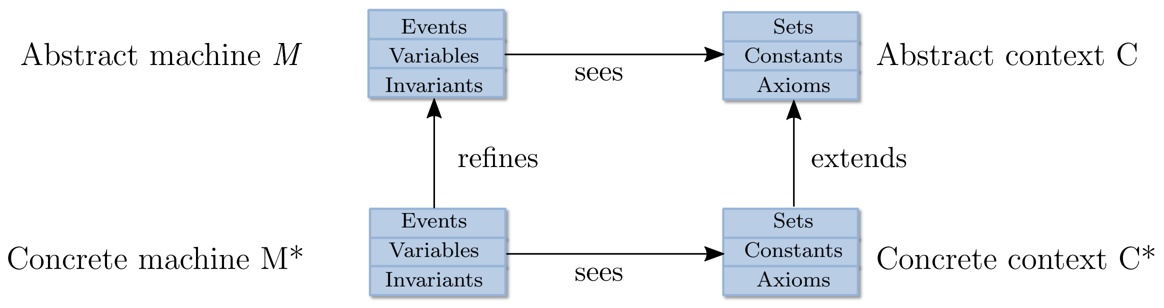


Figure 3.3: Relationship between machines and contexts in Event-B refinement [1]

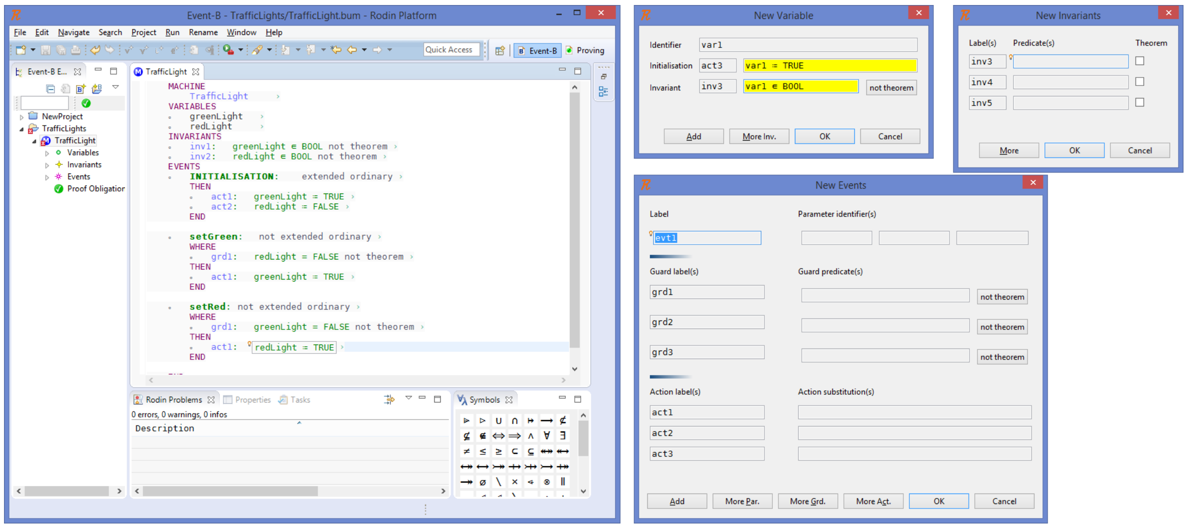
Refining a machine by adding new events is technically possible as it is seen as refinement of a *skip* event which does nothing and can occur at any given time. Assuming the first abstract machine $trafficLights0$ manages to set a variable $traffic\_flow$ to $TRUE$ when the traffic can flow and to $FALSE$ if it has to stop. A refinement could be that the light can be set to green or red corresponding to $traffic\_flow$. The gluing invariant $gluing\_inv$ connects both machines.

MACHINE
$trafficLights1$
REFINES
$trafficLights0$
SEES
$context0$
VARIABLES
$lights\_color$
INVARIANTS
$inv\_new \in \{red, green\}$
$gluing\_inv = traffic\_flow = TRUE \Leftrightarrow lights\_color = green$

As seen above, this refined machine can access the old, more abstract machine. All previously declared variables, invariants, events etc. are still active and functionally.

3.2.3 Tool support

There are two major tools to support modeling in Event-B. The Atelier B [16] environment and the Rodin platform [17] provide facilities for editing machines, contexts, refinements and for proving proof obligations. As seen in Figure 3.4, it has an assistant helping to create new events including variables, invariants etc. As Event-B is developed with refinement in mind, the Rodin tool supports refinement natively, simply by right-clicking an existing machine. The framework for the refinement is then automatically created. The proof obligation is shared by both environments and can be automatic and/or interactive. As the refinement is a progressive procedure, the goal is to distribute the complexity of proofs through the proof-based refinement [13].



```
MACHINE
    TrafficLight
VARIABLES
    greenLight
    redLight
INVARIANTS
    inv1:   greenLight ∈ BOOL not theorem
    inv2:   redLight ∈ BOOL not theorem
EVENTS
    INITIALISATION:    extended ordinary
    THEN
        act1:   greenLight ≔ TRUE
        act2:   redLight ≔ FALSE
    END

    setGreen:   not extended ordinary
    WHERE
        grd1:   redLight = FALSE not theorem
    THEN
        act1:   greenLight ≔ TRUE
    END

    setRed: not extended ordinary
    WHERE
        grd1:   greenLight = FALSE not theorem
    THEN
        act1:   redLight ≔ TRUE
    END
```

0 errors, 0 warnings, 0 infos

Description

Identifier	var1		
Initialisation	act3	var1 ≔ TRUE	
Invariant	inv3	var1 ∈ BOOL	not theorem

Add | More Inv. | OK | Cancel

Label(s)	Predicate(s)	Theorem
inv3		☐
inv4		☐
inv5		☐

More | OK | Cancel

Label	Parameter identifier(s)			
evt1				

Guard label(s)	Guard predicate(s)	
grd1		not theorem
grd2		not theorem
grd3		not theorem

Action label(s)	Action substitution(s)
act1	
act2	
act3	

Add | More Par. | More Grd. | More Act. | OK | Cancel

Figure 3.4: Overview of the basic Rodin Tool assistants

3.2.4 Summary

Event-B has a strict scheme that has to be followed in order to refine a model as it is intended by the method. The refinement process also is also strictly defined and has to be followed to construct a correct refinement of a machine. This developed syntax and semantics can be used to create a fully provable model. Alterations and addition to the modeling language is not supported in Event-B. As for tool support, the RODIN Tool assists the user with all steps of the modeling process including proof obligations.

3.3 ASM

This chapter describes the definitions of the basic Abstract State Machines (ASM), its refinement and tool support. Börger states in his book, that "the ASM Method [...] is a system engineering technique which supports the integration of problem-domain-oriented modeling and analysis into the development cycle. Its goal is to improve industrial system development by accurate high-level modeling which is linked to seamlessly, in a way the practitioner can verify and validate, to executable code" [18].

3.3.1 Overview

An ASM is a finite set of transition rules of the following form:

if *Guard* **then** *Updates*

which can transform abstract states. The *Guard* uses a predicate logic formula, whose output can either be *TRUE* or *FALSE*. *Updates* is a set of (multiple) functions. Its general form is:

$$f(t_1, ..., t_n) := t$$

Applied to the traffic lights example, there can be a transition rule *CHANGELIGHT* which changes the activated lights when the guard *change* is *TRUE* and one of the lights *red* (L) or *green* (L) is *on*.

```
CHANGELIGHT (L) =
  if change and green (L) = on then {green (L) := off, red (L) := on, change (L) := false}
  if change and red (L) = on then {red (L) := off, green (L) := on, change (L) := false}
```

The notion of states in ASMs is the notion of mathematical structures. Like in Event-B, an ASM computation step executes all updates simultaneously. Simultaneous execution of a rule R for each x-satisfying a given condition C can be expressed in the following notation:

```
forall x with C
R
```

Non-determinism can also be expressed by rules of the form:

```
choose x with C
R
```

The meaning of such an ASM-rule is to execute rule R with an arbitrary x chosen among those for which C is true. Common notations like **where**, **let**, **if-then-else** can also be used.

The language of ASM has a predefined syntax and semantics but does not force the user to use it as proposed. Own notations are being encouraged often adds a more understandable meaning to a model. A term like *Light* $(L1) = \{(●,○,○)\}$ or *Light* $(L2) = \{(○,○,●)\}$

can be perfectly fine as substitution for a Boolean assignment like the one in *CHANGELIGHT*.

The traffic light L has the variables $green\,(L)$ and $red\,(L)$ for indicating the lights state, $change\,(L)$ which is $TRUE$ whenever the lights shall be changed and an external event $BUTTONPRESSED\,(L)$ which is $TRUE$ whenever an button is pressed. The two simple transition rules define a traffic light with two colors changing its state whenever the button is pressed.

```
$INIT\,(L) =$
  $red\,(L) := true$
  $green\,(L) := false$


$PRESSBUTTON\,(L) =$
  $if\ BUTTONPRESSED\,(L)\ thenchange\,(L) := true$


$CHANGELIGHT\,(L) =$
$if\,change\,and\,green\,(L) = true\,then\,\{green\,(L) := false,\ red\,(L) := true,\ change\,(L) := false\}$
$if\ change\ and\ red\,(L) = true\ then\ \{red\,(L) := false,\ green\,(L) := true,\ change\,(L) := false\}$
```

3.3.2 Refinement of ASM

As stated in the de facto standard from Börger, “The guiding principle of the ASM refinement method is the problem orientation” [18]. There is no direct rule or principle that has to be followed in order to refine an ASM. There are several mathematical and formalized logic elements that can be used, but as mentioned, an own notation can be created in order to achieve a concrete goal or to adapt the needs of a specific system [19].

The are several steps a designer has to define in order to refine an ASM M to an ASM M^* where the most important ones are the notion of a refined state and the notion of equivalence of data in corresponding states of interest[19].

A refinement of this model could be the addition of a time-based switch of light states. A new variable *lastchange* is needed, which stores the time information about the last change of states. Also the *INIT* and *PRESSBUTTON* transitions have to be expanded in order to adapt to the new timing transition. To do this, a global variable *DELTA* is predefined which stores information about the transition interval.

```
$INIT\,(L) =$
  $lastchange\,(L) := CURRENTTIME$
  $red\,(L) := true$
  $green\,(L) := false$


$TIMER\,(L) =$
  $if\ lastchange\ +\ DELTA\ =\ CURRENTTIME\ then\ change\,(L) := true$


$PRESSBUTTON\,(L) =$
  $if\ BUTTONPRESSED\,(L)\ then\ \{change\,(L) := true,\ lastchange := CURRENTTIME\}$
```

3.3.3 Tool support

There are several tools that support ASM. The Karlsruhe Interactive Verifier (KIV), available as Scala version and as Eclipse plugin [20], has been modified to support the specific needs the ASM method requires [21]. The usage of stepwise refinement allows that problems can be split into smaller, more comprehensible problems. Another approach has been made with the Prototype Verification System, which is especially oriented towards automatic proof checking and mechanized proving of properties [22]. The generic theorem prover Isabelle has also been successfully adapted for ASM [18]. There have been some advances for validation checking [21], but those limit the open extensibility that ASM offers in order to permit certain expressions.

3.3.4 Summary

ASM is a formal method focusing on problem orientation. It supports the usage of own notions and possible corresponding notations. Therefore, the refinement process is less strict than other formal methods, but nevertheless has to be well defined by the user. Since complex proof obligations can be split into smaller ones, the KIV tool is able to verify ASM models.

3.4 Z

Z is a formal method, which relies heavily on the usage of mathematical notation to describe computer systems. Z has been around since the late 1970s and was developed by Jean-Raymond Abrial and the Oxford University. Later, several industrial partners, like IBM, supported further development. The first Z reference manual, published in 1989, has been considered standard by the American National Standards Institute, the British Standard Institute and the International Standards Institute. The last and final standardization for Z was completed in 2002 and is publicly available from the ISO website. The Z notation uses structured, mathematical text. The main construct is called *paragraph*. The most conspicuous kind of Z paragraph is a macro-like abbreviation and naming construct called the *schema*. Z defines a *schema calculus* that can be used to build big schemas from smaller ones. The schema is the feature that most distinguishes Z from other formal notations. The mathematical notation of Z consists of a small core, supplemented by a larger collection of useful objects and operators called the Z *mathematical tool-kit* [7].

3.4.1 Overview

This chapter introduces the basic constructs of the Z notation.

Displaying sets
Often software systems contain different types of things. In Z similar objects are gathered into collections which are treated as an object on its own. These collections are called *sets* and are a central feature of Z.

$$\{red, yellow, green\}$$

Sets can have names to avoid writing them every single time. Some often used sets have already an assigned standard name (e.g. $\mathbb{Z}$ is the set of integers and $\mathbb{N}$ is the set of natural numbers).

Types
In Z sets must contain objects of the same *type*. Types and sets are closely related. Every type has a *carrier set*, which includes all objects of that specific type. Every type is a set, but not every set is a type.

$\{red, green, yellow, blue, orange, ...\}$ is a set
$\{red, green\}$ is a type (and also a set)

Variables and declaration
In order to talk about any object in Z, its name has to be introduced in a declaration. Declarations introduce variables and define which set each variable belongs to.

$i : \mathbb{Z}$	[i is an integer]
$color : COLORS$	[$color$ is one of the values in the $COLORS$ set]

The declarations in Z can usually express more than the declaration used in programming languages, which usually declare objects as a type. Z declarations can be more restricted because they use sets, not just types.

$e : EVEN$	[e is an even number]
$o : ODD$	[o is an odd number]

In most programming languages e and o would normally be declared the same way because they share the same integer type.

Constraining variables, axiomatic definitions
A variable is constrained when it is defined which values it can take. Declarations can constrain a variable. In Z variables are defined in a construct called an *axiomatic definition*, which can include more constraints. Its general form is:

$$\begin{array}{|l} declarations \\ \hline predicates \end{array}$$

Predicates are optional and can further constrain variables introduced in the declaration.

$$\begin{array}{|l} l_1, l_2 : red, green \end{array}$$

An axiomatic definition with a predicate constrains the variables l_1 and l_1 even further by not allowing them to take the same value.

$$\begin{array}{|l} l_1, l_2 : red, green \\ \hline a_1 \neq a_2 \end{array}$$

Variables declared in an axiomatic definitions can be used anywhere in a Z model after their definition and always have the same value. This differs from many programming languages in which you can have multiple variables of the same name but each one can have a different value.

Sometimes it is necessary to distinguish between sets and types. The type is defined in the declaration above the line and additional information about the membership is provided in the predicate below. A definition where types are explicitly spelled out this way is said to be *normalized.* A *signature* is a declaration, which names the type and so is normalized.

Writing in the normalized definition is making clear which variables have the same type. This can be important to know, because variables from different sets can still be used in the same expression as long as they have the same type.

As $\mathbb{Z}$ is the only predefined type in Z, it is necessary to provide the possibility to add new types. There are two possible ways to introduce these. The first is the *free type* definition. A free type can be defined when a finite number of objects is given and all the elements are known. To define a free type, the following syntax is used:

$$COLORS ::= red \mid green$$

The second method is the *basic type* definition, which can be used when not all the elements it might contain are known. A basic type can be defined by its name contained in brackets.

$$[NAME]$$

The basic type NAME can contain names of persons with a growing number of individuals.

Z allows great freedom in choosing names. Z identifiers can also include symbols beside the alphanumeric alphabet.

Schemas
Z is also known for so called schemas. "The schema language is used to structure and compose descriptions: collating pieces of information, encapsulating them, and naming them for re-use" [9]. The schema is using the previously defined declarations and predicates.

A possible schema for the traffic light example could be

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_ LightControl \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
lights : Traffic \rightarrow Light \\
lastchanged : Traffic \rightarrow Time \\
\hline
\forall t_1, t_2 : Traffic \mid t_1 \neq t_2 \bullet lights(t_1) = red \vee lights(t_2) = red \\
\hline
\end{array}$$

where a color and a time is assigned to a direction of traffic. The predicate ensures, that two directions t_1 and t_2 are not assigned the same color and that at least one of them has to be red.

Operations
Operations can be applied to schemas and can use all elements of the related schema. *LightChanger* changes the value of *lastchanged* to the current time whenever any *Traffic* element is changed.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_ LightChanger \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta LightControl \\
t? : Traffic \\
\hline
lastchanged' = lastchanged \oplus t? \rightarrow CurrentTime \\
\hline
\end{array}$$

Example
This particular form of the traffic light example has a traffic flow, which can have the states *Direction1* and *Direction2*, a traffic light which can be either *red* or *green* and a time based state changing mechanism.

$$Light == \{red, green\}$$
$$Traffic == \{Direction1, Direction2\}$$

$$\begin{array}{|l}
\hline
\textit{LightControl} \\
lights : Traffic \rightarrow Light \\
\hline
\forall t_1, t_2 : Traffic \mid t_1 \neq t_2 \bullet lights(t_1) = red \vee lights(t_2) = red \\
\hline
\end{array}$$

The *LightControl* schema restricts the traffic light in a way that one direction always has to get a *red* signal, while the other direction does not necessarily needs to be *red*, but this predicate assures that as soon as a direction has a *green* signal, the light of other direction always has to be *red*.

$$\begin{array}{|l}
\hline
\textit{LightControlInit} \\
LightControl' \\
\hline
\forall t : Traffic \bullet lights(t) = red \\
\hline
\end{array}$$

This *LightControlInit* schema sets all directions to *red* as it ensures a safe way to start the traffic light control. This however requires an other schema that controls the following start sequence and initialization, but due to simplicity, this is not implemented here.

$$\begin{array}{|l}
\hline
\textit{LightChanger} \\
\Delta TrafficControl \\
t? : Traffic \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{ToGreen} \\
LightChanger \\
\hline
lights(t?) = red \\
lights' = lights \oplus t? \rightarrow green \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{ToRed} \\
LightChanger \\
\hline
lights(t?) = green \\
lights' = lights \oplus t? \rightarrow red \\
\hline
\end{array}$$

The schemas *ToGreen* and *ToRed*, each initiated by *LightChanger*, change the color of the traffic light corresponding to a direction. The problem that this description holds is that there is no state changing mechanism implemented yet and the color of the traffic light is alternating continuously.

3.4.2 Refinement of Z

There are a few rules, which must be followed by refining in Z. When a concrete operation O_2 refines an abstract operation O_1, this is noted by $O_1 \subseteq O_2$. The refinement has to show applicability and correctness:

- When O_2 is applicable, so is O_1.
- When O_1 is applicable, but O_2 is applied, the result is consistent with O_1.

While in Z the observable behavior of a specification is preserved, the internal representation, however, can be changed by refinement. The whole addition of timing to change the traffic lights color can be seen as refinement of the example.

$$\begin{array}{l} \mathit{Light} == \{\mathit{red}, \mathit{green}\} \\ \mathit{Traffic} == \{\mathit{Direction1}, \mathit{Direction2}\} \\ [\mathit{Time}] \\ \mathit{CurrentTime} : \mathit{TIME} \end{array}$$

In addition to *Light* and *Traffic* an open type called *Time* is introduced to describe the *CurrentTime*.

$$\begin{array}{|l} \hline \mathit{LightControl} \\ \mathit{lights} : \mathit{Traffic} \rightarrow \mathit{Light} \\ \mathit{lastchanged} : \mathit{Traffic} \rightarrow \mathit{Time} \\ \hline \forall\, t_1, t_2 : \mathit{Traffic} \mid t_1 \neq t_2 \bullet \mathit{lights}(t_1) = \mathit{red} \lor \mathit{lights}(t_2) = \mathit{red} \\ \hline \end{array}$$

A new declaration has been added to assign a time to the last color change to one of the directions.

$$\begin{array}{|l} \hline \mathit{LightControlInit} \\ \mathit{TrafficControl}' \\ \hline \forall\, t_1, t_2 : \mathit{Traffic} \bullet \mathit{lights}(t_1) \neq \mathit{lights}(t_2) \\ \hline \end{array}$$

The *LightControlInit* is changed in a way, that the light of one direction has to be *red* and the other directions light is forced to be *green*.

$$\begin{array}{|l} \hline \mathit{LightChanger} \\ \Delta \mathit{TrafficControl} \\ t? : \mathit{Traffic} \\ \hline \mathit{lastchanged}' = \mathit{lastchanged} \oplus t? \rightarrow \mathit{CurrentTime} \\ \hline \end{array}$$

$ToGreen$

$LightChanger$

$lights(t?) = red$
$CurrentTime - lastchanged(t?) \geq 20$
$lights' = lights \oplus t? \to green$

$ToRed$

$LightChanger$

$lights(t?) = green$
$CurrentTime - lastchanged(t?) \geq 20$
$lights' = lights \oplus t? \to red$

After a predicted amount of time (here 20 units of whatever Time measuring unit *TIME* is having) the assigned color is switched. This method obviously only works with two directions since more than two directions would create an error in the predicate of *LightControl* where *red* has to be assigned at least to one direction.

3.4.3 Tool support

There are several Z tools available, which offer parsing and type checking facilities and some also offer proof facilities [23][24][25]. Most of these tools, however, do not support the Z ISO standard, established in 2002. The Community Z Tools is an open-source community project, initiated in 2001 by Andrew Martin [26]. One of its primary objectives is to encourage interchange between existing Z tools [27]. Recently, even a prototype for refinement checking has been in development [28], but in general, due to the change and variations in Z till 2002, there has been no practicable tool for validating a Z model.

3.4.4 Summary

Z has a scheme which needs to be followed in order to refine a model intended by the method. The refinement process also is strictly defined and is to be followed to construct a correct refinement of a machine. This developed syntax and semantics can be used to create a fully provable model. Like Event-B, alterations and addition to the modeling language are not supported by Z.

3.5 Discussion

The previously introduced formal methods Event-B, ASM and Z are discussed and evaluated in this chapter. Further, it will be discussed how they fit in the taxonomy from Section 3.1. Based on the individual formal methods' refinement mechanisms, they will be rated and classified regarding correctness, strictness and expressiveness.

As Table 3.1 shows, both ASM and Z require manual validation due to lacking tool support. Event-B with its strict construction rules, particularly the usage of gluing invariants, ensure a safe and correct refinement and the strong Rodin-Tool, as all-round development kit, allows the user to automatically validate a model and its refinement as the modeling-process progresses. ASM has several tools with approaches for automated validation. Recently, the possibility for automatic validation for Z refinements is in development. This is particularly interesting due to the fact, that only Z is officially standardized for more than a decade.

Formal method	Correctness by		
	manual validation	automatic validation	construction
Event-B		+	++
ASM	+	(+)	
Z	+	(+)	

Table 3.1: Evaluation of Event-B, ASM and Z regarding their correctness of refinement

From the three exemplary chosen formal methods, solely ASM supports language alteration (see Table 3.2). Event-B and Z do not support own syntax changes. Taking the correctness aspect into account, there seems to be a direct relation between strictness and correctness. As soon as the language of a formal method allows alteration and addition to it, the ability of an automatic validation is much more complex and therefore aggravated. New elements are to be fully defined, which may be easily feasible by construction and visualization tools, but validating an unknown, new format is hardly possible for computer programs. In contrast to ASM, Event-B and Z are predefined languages, which can not be changed and have a fixed set of formalities. Therefore tool support is available for at least constructing, syntax and type checking.

Formal method	fixed rules	alterable rules
Event-B	+	
ASM		+
Z	+	

Table 3.2: Evaluation Event-B, ASM and Z regarding their strictness

While conducting research on the different formal methods and their refinement mechanisms there was no sign of restriction in the refinement mechanisms to certain aspects (see Table 3.3) of the example presented here. All formal methods introduced could express the traffic light example without problems and even different refinements like the addition of a second set of traffic lights was possible. Any form of state changing mechanics could be expressed in all three examined formal methods. Due to the fact, that these formal methods are quite popular, it is reasonable that they support a wide variety of software and partly even hardware systems [18]. It seems unlikely that all formal methods can express every kind of software systems. Further research is necessary to analyze possible restrictions in the expression of refinement of specific software systems.

Formal method	arbitrary	restricted
Event-B	+	
ASM	+	
Z	+	

Table 3.3: Event-B, ASM and Z regarding their expressiveness

Summarizing, the three formal methods each have a specific property showing a prominent feature. ASM has the most flexibility and freedom regarding depiction of desired needs. Theoretically it can model almost everything; the developer just has to define the meaning of his or hers notion. Event-B has a mostly predefined syntax, fixed structures and refinement mechanisms. This may restrict the modeling process at some point, but it has the advantage, that the tool support is at an advanced state, which does not only allow assisted construction of a model, but validation on the fly as well. Z is one of the few officially standardized formal methods, but the tool support is still very limited. Most tools were commercial products, but are now freely available. Since the support for this tools has already been stopped, the available tools do not support the normed version of Z anymore and are therefore outdated.

3.6 Conclusion

Formal methods are a promising way to model system specifications with the intend to refine them gradually and to make them unambiguously and mathematically provable. Due to a wide variety of different formal methods, each development team has the responsibility to find a formal method suitable for their needs and willingness to get involved with a new and maybe complex modeling language. In an approach to characterize and classify refinement supported formal methods, it turned out, there are different kinds of refinement mechanisms. Those can be taken into consideration to help to support the process of finding a suitable formal method. The three pillars of refinement mechanisms presented here are expressiveness, strictness and correctness and its breakdowns respectively. Exemplary, Event-B, ASM and Z have shown that this taxonomy can be applied to different existing formal methods. For further research, other formal methods like VDM or Alloy can be taken into consideration and it can be tried to apply presented taxonomy to them.

4. Related Work

Banach takes a similar approach to investigate several formal methods [29]. The focus lies even more on the notion of formal methods than this thesis. Also differences and similarities are examined in order to discuss the relation between different approaches. This is used to discuss which methodologies could be applied from one language to another. In addition to this, an overview of the possible tool support is presented. With the relation of different formal methods in mind, it is discussed how tool support can be improved and how a greater tool flexibility can be achieved. This paper is a good addition to this thesis as it takes a closer look at the tool support and the general methodologies of refinement.

This thesis focuses on refinement mechanisms of a selected number of formal methods and how applicable they are in practice. The survey by Woodcock et al. [30] summarizes the general use of formal methods in industrial projects. It revisits several older surveys and compares them with their own survey and thus creates an overview of the change of usage of formal methods in the industry over the years. In contrast to this mainly theoretical work with practical approaches, the survey analyzes the effect of the practical use of formal methods on time, cost and quality improvement or worsening. Whereas here, with the traffic light example, one single application is presented, the Woodcock et al. survey shows different domains of application as well as different types of application (e.g. real-time, parallel, graphics etc.).

In 1986 Mili et al. also compared models of stepwise refinement [31], but they followed an other approach than this thesis. They did not restrict stepwise refinement to formal methods, but to programming in general. Instead of surveying different programming languages or paradigms and then evaluating possible different refinement mechanics or models, it was assumed that there *are* three different models of stepwise refinement: Program refinement by assertion decomposition, functional decomposition and relational decomposition. These three models are introduced by a common set of notations and assumptions and are later compared, especially regarding their common features. In addition to that, three different examples showed how those can be used in practice.

A fairly recent publication from the 4th International Conference ABZ 2014 introduced a case study called “ABZ 2014: The Landing Gear Case Study” [32]. This case study was developed specifically as representation of industrial needs. That in mind, it is obviously more detailed and more complex than the extract of a traffic light controller. It also includes the modeling of external systems like mechanical or hydraulic parts. The most interesting part is, that this publications contains 11 papers, which each suggest another approach on how this case study can be implemented. They use different formal methods to develop a model including ASM, Event-B and Z. They also propose different kinds of refinement and verification techniques, like model checking and simulation. This collection of papers is a good opportunity to gain an insight of the use of formal methods in an actual fully specified industrial project.

5. Conclusion

This thesis proposed a taxonomy that can be applied to existing refinement supporting formal methods. This taxonomy consists of three main aspects with possible sub-categories. These main aspects have been:

- Expressiveness has evaluated, whether the refinement is restricted so specific software systems or arbitrary modeling is possible
- Strictness has evaluated, whether the formal method is well defined or modifiable to express own refinement notions
- Correctness has evaluated how the correctness of a refinement can either be achieved by validation or by construction mechanics

With Event-B, ASM and Z, three different formal methods have been introduced by explaining the basic notations and by using a common example to see possible differences on the same model (with slightly changes). The proposed taxonomy has shown, that it can be applied on the introduced formal methods. Further, it is shown, that this three formal methods differ in selected aspects. The notion of how correctness from one refinement step to another can be achieved may be by manual or automatic validation or the construction itself can imply the correctness by the use of a pre-validated set of possible operations.
With this overview and the proposed taxonomy a starting point has been created, which allows interested users to gain an first insight into specific features and the basic notion of these formal methods. Since not all formal methods have been examined, possible future work has a few possibilities to complement this thesis and further research is necessary to spot and evaluate possible limitations to specific software systems for any of these formal methods. The traffic light example can be seen as first step of a case-study to approach that task.

Bibliography

[1] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inf.*, 77(1-2):1–28, 2007. (cited on Page vii, 11, and 14)

[2] Elvis C. Foster. *Software Engineering A Methodical Approach*. Apress, 2014. (cited on Page 1)

[3] V. S. Alagar. *Specification of Software Systems*. Springer-Verlag London Limited, 2011. (cited on Page 1)

[4] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971. (cited on Page 1)

[5] Leszek Maciaszek. *Requirements analysis and system design*. Pearson Education, 2007. (cited on Page 3)

[6] Haneen Hijazi, Shihadeh Alqrainy, Hasan Muaidi, and Thair Khdour. Risk factors in software development phases. *European Scientific Journal*, 10(3):213–232, 2014. (cited on Page 3)

[7] Jonathan Jacky. *The way of Z: practical programming with formal methods*. Press Syndicate of the University of Cambridge, 1997. (cited on Page 3 and 18)

[8] John Michael Spivey. *The Z notation a reference manual*. Prentice-Hall, Inc., c 1992. (cited on Page 3)

[9] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., 1996. (cited on Page 4 and 20)

[10] Yuan Ren, Gerd Gröner, Jens Lemcke, Tirdad Rahmani, Andreas Friesen, Yuting Zhao, Jeff Z. Pan, and Steffen Staab. Process refinement validation and explanation with ontology reasoning. In *Service-Oriented Computing*, pages 515–523. Springer, 2013. (cited on Page 10)

[11] Frédéric Badeau and Arnaud Amelot. Using b as a high level programming language in an industrial project: Roissy val. In *ZB 2005: Formal Specification and Development in Z and B*, volume 3455, pages 334–354. Springer Berlin Heidelberg, 2005. (cited on Page 11)

[12] Stefan Hallerstede. On the purpose of event-b proof obligations. In *Abstract State Machines, B and Z*, volume 5238, pages 125–138. Springer Berlin Heidelberg, 2008. (cited on Page 11)

[13] Neeraj Kumar Singh. *Using Event-B for Critical Device Software Systems*. Springer Science & Business Media, 2013. (cited on Page 11 and 15)

[14] Rodin User's Handbook v.2.8. Website. Available online at http://handbook.event-b.org/; visited on July 5th, 2015. (cited on Page 12)

[15] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Peyton-Jones, et al. Roadmap for enhanced languages and methods to aid verification. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236. ACM, 2006. (cited on Page 13)

[16] ClearSy. Atelier B. Website. Available online at http://www.atelierb.eu/en/; visited on May 24th, 2015. (cited on Page 14)

[17] RODIN - Rigorous open development environment for complex systems. Website. Available online at http://www.event-b.org/; visited on July 4th, 2015. (cited on Page 14)

[18] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., 2003. (cited on Page 16, 17, 18, and 24)

[19] Egon Börger. The asm refinement method. *Formal Aspects of Computing*, 15(2-3):237–257, 2003. (cited on Page 17)

[20] Augsburg University Institute for Software & Systems Engineering. KIV - Karlsruhe Interactive Verifier. Website. Available online at http://www.isse.uni-augsburg.de/en/software/kiv/; visited on May 24th, 2015. (cited on Page 18)

[21] M. Giese, D. Kempe, and A. Schönegge. *KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining-Architektur*. Universität Karlsruhe, Fakultät für Informatik, Bibliothek, 1997. (cited on Page 18)

[22] Angelo Gargantini and Elvinia Riccobene. Encoding abstract state machines in pvs. In *Abstract State Machines - Theory and Applications*, pages 303–322. Springer Berlin Heidelberg, 2000. (cited on Page 18)

[23] Z TYPE CHECKER. Website. Available online at http://se.cs.depaul.edu/fm/ztc.html; visited on June 2nd, 2009. (cited on Page 23)

[24] CADiZ. Website. Available online at http://www.cs.york.ac.uk/hise/cadiz/; visited on May 25th, 2015. (cited on Page 23)

[25] Mike Spivey. The fuzz type-checker for Z. Website. Available online at http://spivey.oriel.ox.ac.uk/mike/fuzz/; visited on May 15th, 2015. (cited on Page 23)

[26] Community Z Tools Initiative (CZT). Website. Available online at http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT; visited on May 26th, 2015. (cited on Page 23)

[27] CZT: Community Z Tools - Tools for developing and reasoning about Z specifications. Website. Available online at http://czt.sourceforge.net/; visited on June 2nd, 2015. (cited on Page 23)

[28] John Derrick, Siobhán North, and Anthony J. H. Simons. Building a refinement checker for Z. In *Proceedings 15th International Refinement Workshop, Refine 2011, Limerick, Ireland, 20th June 2011.*, pages 37–52, 2011. (cited on Page 23)

[29] Richard Banach. Model based refinement and the tools of tomorrow. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z*, pages 42–56. Springer-Verlag, 2008. (cited on Page 27)

[30] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, 2009. (cited on Page 27)

[31] Ali Mili, Jules Desharnais, and Jean Raynomd Gagné. Formal models of stepwise refinements of programs. *ACM Comput. Surv.*, 18(3):231–276, 1986. (cited on Page 27)

[32] F. Boniol, V. Wiels, Y.A. Ameur, and K.D.Schewe. *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.* Springer International Publishing, 2014. (cited on Page 28)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 07. August 2015