Automating abstractions in formal modelling

4th Year Dissertation

Peter Kovacs – MEng Software Engineering

School of Mathematical and Computer Sciences – Heriot-Watt University

Supervisor: Gudmund Grov

Second Reader: Andrew Ireland

April 27, 2015
Declaration

I, Peter Kovacs, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Date:
Abstract

The growing complexity of computer systems means a higher probability of introducing design errors. Complexity of systems and requirements can be managed by using an incremental model development approach such as refinement development methodology. A refinement development methodology means modelling is started at an abstract level, and is incrementally refined through a number of refinement layers until the final version of the system design is achieved. This methodology addresses the issues of designing systems with complex requirements, however modelling is often started at a too concrete level which may result in introduction of flaws. Using a framework it is feasible to automate the process of generating abstractions in the formal language of Event-B and may help to identify or discard source of flaws present in the model.
Acknowledgements

First of all I would like to express my gratitude to Gudmund Grov and Andrew Ireland for the opportunity to work on this exciting project. I would like to thank my supervisor Gudmund Grov for giving me a lot of help, advice and feedback throughout the year. I would also like to thank Andrew Ireland for giving me guidance and feedback.

Finally, I would like to express my gratitude to my family. I would like to thank my brother, my sister for helping me during the last years and my parents for their help and support during my university studies. Without their support I would not have been able to study at Heriot-Watt University.
Table of contents

Declaration .................................................................................................................. 2
Abstract ..................................................................................................................... 3
Acknowledgements ................................................................................................... 4
I. Introduction ............................................................................................................. 3
   1.1 Motivation ......................................................................................................... 4
Abstractions in formal modelling .............................................................................. 5
   1.2 Document Format ............................................................................................ 6
II. Project Description ................................................................................................. 7
   2.1 Hypotheses ...................................................................................................... 7
   2.2 Aims and Objectives ....................................................................................... 7
   2.3 Methodology .................................................................................................... 8
      2.3.1 Analysis ..................................................................................................... 8
      2.3.2 Implementation ....................................................................................... 9
      2.3.3 Evaluation .............................................................................................. 9
III. Technical Background ......................................................................................... 10
   3.1 Formal methods in software engineering ....................................................... 10
      3.1.1 Formal Specification ............................................................................... 10
      3.1.2 Formal Verification ................................................................................. 10
   3.2 Modelling in Event-B ...................................................................................... 11
      3.2.1 Refinements ............................................................................................ 12
   3.3 Abstractions ..................................................................................................... 15
      Reduction and Abstraction Techniques in Model Checking ......................... 15
      Abstraction in Event-B .................................................................................... 16
   3.4 Rodin Platform ............................................................................................... 16
      3.4.1 Formula .................................................................................................. 17
   3.5 Scala ................................................................................................................ 17
I. Introduction

Software and computer systems have become an essential part of our everyday life. Whether they are built for making our life easier, safer, or just in the purpose of entertaining us, we use them or depend on them every day sometimes without even realizing it. Computer systems have important roles in a vast range of systems including medical systems, security systems, traffic light controllers and banking industry. It is very hard to find a single area today where computer systems are not present.

Correctness of a design and implementation is very important especially in case of safety critical systems where a simple mistake or bug may cost a lot of money or even people’s life. The followings are well known bugs in software engineering:

**Moody’s** Investors Service due to a bug in their computer models awarded the best triple-A ratings to debt products that should have been rated four levels lower. Investors holding these products have lost up to 60 percent of their investment (Jones et al. 2008).

**Toyota** Motor Corp., called back more than 6 million vehicles after discovering five types of safety defects in their vehicles. It cost Toyota about $600 million dollar and a loss of reputation.

**Therac-25** was a computer-controlled radiation therapy machine. A software design bug in the system caused the massive overdose of patients. As a result, six people died and many others were injured (Leveson 1995).

**Flight 501** the maiden flight of the Ariane 5 launcher, exploded 36 seconds after the initiation. It cost about 360 million in US dollars. Errors in capturing the application and environment requirements and in the course of designing process resulted into flaws at the system modules level (Le Lann 1997).

In order to avoid errors that may cause loss of money or even human lives and build correct and reliable systems, use of formal methods can provide a more reliable approach. “Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems” (Aichernig et al. 2014).
1.1 Motivation

The growing functionality of computer systems and use of concurrent and distributed technologies increases the complexity. That means a higher probability of errors present in the system. This complexity can come from the number of the requirements or simply from the complex design of the system. Without using a rigorous approach, complexity can make error detection very hard in the early stages of software development process and fixing errors detected late in the development lifecycle is very expensive. Errors can be flaws introduced in the development process or during the requirements analysis phase by either because of ambiguity or there was a conflict in the set of requirements. The later a specification or design error is realized in the system the higher the cost to fix it (Butler 2012).

Formal methods makes available the analysis and verification of model at any part of the program life-cycle in contrast to the traditional approaches where ensuring correctness are highly based around testing which can only be done once a lot of implementation has been undertaken. Formal methods are more useful in the design stage rather than in the coding stage. “The vital step in a high quality software engineering process is requirements engineering” (Woodcock et al. 2009).

Agile is a software development methodology that focuses on high productivity in software development and provides a flexible approach to accommodate changes in requirements. Software is developed in increments where each increment may introduce new requirements. While the requirements may often change during a project it is important to ensure that a flawless system design is maintained. As the requirements are changing formal models will change as well and a flawless system design can be maintained. Agile methods are becoming increasingly popular in industry nowadays.

One reason it is difficult to identify errors introduced early in the development cycle is the lack of precision in formulating specifications, thus resulting in ambiguities and inconsistencies that may not be realized until they come to surface in a later phase of the development process or in worst case
during the live operation of the system. Using a formal language of mathematics - defined syntax and semantics to describe system properties and behaviours helps avoiding ambiguity and achieve a high degree of precision but does not necessarily solve the problem of complexity. Complexity of systems and requirements cannot be completely eliminated but can be managed by using an incremental model development approach using abstractions (Butler 2012).

**Abstractions in formal modelling**

Abstractions and refinements in formal modelling have been used to address the issues of system requirements and verification complexity. Using a refinement approach in formal modelling, means the system modelling is started at an abstract level, and refined through a number of refinement layers. Each layer is adding more requirements and complexity of the given system. At each refinement step, it has to be proven that the more concrete model does not invalidate what has been done in the abstract model. This approach makes available a systematic development of a model and by proving model and refinement correctness at each step, at the end of the development, the system is correct by construction relative to its surrounding environment (Abrial 2010).

It is often found difficult to find at which level to start, and there is a tendency to start the model development way too concrete, thus making the development and the verification difficult. The problem with starting the development at a too detailed level is that it may not be realized that a flaw is present either in other part or in the whole structure of the system. As each refinement layer is adding complexity and requirements of the system, it may also introduce flaws to the model. These problems serve as a motivation for the project to automate the generation of abstractions of formal models and led to the formation of the hypotheses.
1.2 Document Format

Chapter 1 – An introduction and motivation to the project, how it helps software engineering.

Chapter 2 – A description of the project, hypothesis, aims and objectives and the methodology used.

Chapter 3 – A discussion of the technical background needed to understand the project.

Chapter 4 – Discussion of the analysis performed describing patterns in refinements and use of them in abstractions.

Chapter 5 – A discussion of the changes in the design and implementation of the framework

Chapter 6 – A discussion of the implemented abstractions/transformations.

Chapter 7 – Evaluation of the implementation of abstractions, the language and framework.

Chapter 8 – The conclusion and the summary of this project.

References

Appendix A – Class Diagrams

Appendix B – Generated Transformation/Abstractions
II. Project Description

In the designing phase, starting the system modelling with the right level of abstraction is hard. It is often started at a too concrete level that has higher risks of introducing flaws in it. Starting with the right level of abstraction and progressing towards the concrete model representing the real world design helps to design a flawless system. Automating the generation of abstractions would automate the process of generating an abstraction from a too concrete model that may contain flaws. The focus of the project is to discover feasibility of automating the generation of abstraction in the formal language of Event-B.

2.1 Hypotheses

The main objective of this project to show that:

„It is feasible to automatically generate suitable abstractions from a model in Event-B.“

2.2 Aims and Objectives

In software engineering, patterns are often used to address common design problems that occur frequently. In programming and in modelling as well human nature of solving problems will result in the generation of patterns in the long term. This project will discover common patterns used in Event-B method in order to automate the process of generating abstractions.

An operator framework is available as a source for this project that provides a tool, an operator language to define basic transformations of Event-B models. This framework and language will serve as a core for this project. The suitability of this language and framework for automating abstractions will be evaluated as well.

The prioritized objectives of the project are:

1. Discover patterns of refinements by analysing existing case studies in Event-B.
2. Define abstraction from the discovered refinement patterns
3. Evaluate the suitability of the operator language and framework in generating abstractions
4. Extend the operator language to implement the defined abstractions
5. Evaluate the benefits of abstraction/transformations in focusing and abstracting away flaws
2.3 Methodology

The project has three important parts: the analysis, implementation and evaluation.

2.3.1 Analysis

Event-B is formal language of following a refinement methodology in modelling. It starts by an abstract representation and works towards the final concrete model in increments. The analysis focuses on analyzing formal models developed using refinement approach. Case studies available in Event-B repository and refinement strategies described through various examples by Jean-Raymond Abrial will be used as a basis for discovering refinement patterns and forming abstraction plans for each pattern by reversing the refinement approach. The usefulness of the defined abstractions in discarding flaws will be also analyzed in this part. Transformations and abstractions will be defined and applied following three methodologies:

1. **Abstracting the model**
   
   Generate abstraction layers of a given model.

2. **Discarding the flaw**
   
   Generate abstraction of the model such that the identified flaw is discarded from the design.

3. **Focusing on the flaw**
   
   Generate transformation of the model such that the flaw is still present in the system and the model is simplified in order to focus on the source of the flaw.

![Case 1](image1)

![Case 2](image2)

![Case 3](image3)

Figure 2.1.
The best ways to understand these abstraction/transformations and their benefits is by using an analogy. The expression “Can’t see the forest for the trees” is used when someone is too involved in the details that they do not look at the problem as a whole. This expression describes the problem very well in system development.

By abstracting away we can either get a view where the problem is not present anymore (case 2), or abstract away other parts of the system that are not relevant to the problem thus getting a closer look at the given problem (case 3).

2.3.2 Implementation

Once the abstractions have been defined, the focus will be turned to implementing abstractions using the Operator Language. The abstraction plans would be broken down to a set of basic model transformations that could be implemented by combining a set of operators. New operators would be implemented and the existing ones might be changed to make the application and combination of operators more natural.

This part also serves as an evaluation for the Operator language. It will modify and improve the language in order to implement the defined abstractions and transformations.

2.3.3 Evaluation

The implementation of the abstractions and transformations, the suitability of the operator language and the framework will be evaluated. The efficiency of an implementation may depend on the number and nature of the operators being used for the given implementation. The main goal is to generate a certain abstraction without leading to search space explosion. One possible way to avoid this is applying various filters for example discarding transformations that fail type checking or discarding incorrect abstractions.
III. Technical Background

This chapter describes the technical background needed in order to understand the project and describes the framework available as a source for this project.

3.1 Formal methods in software engineering

Formal methods allow developers to construct reliable systems by providing mathematically sound languages, techniques and tools in order to support the specification and verification of such system. By using formal methods inconsistencies, ambiguities and incompleteness can be detected. Formal Specification and Formal Verification are formal methods used in software engineering

3.1.1 Formal Specification

Formal specification is a formal method to describe system properties using language of mathematically defined syntax and semantic. These properties can be functional behaviour, timing behaviour, performance characteristics or structural properties.

3.1.2 Formal Verification

Formal verification is a formal method used to mathematically prove the correctness of a model. Although formal verification can be time consuming and expensive it is essential in case of safety-critical systems as it gives high assurance of system correctness. To verify the correctness of a model first involves the definition of properties, system requirements using a suitable mathematical language. Given a model and properties it has to be verified the model satisfies the defined properties(Ouimet 2008).

There are two approaches to formal verification: Model Checking and Theorem Proving.

Model Checking

Model Checking is based on building a finite model of the system and checking that the defined properties hold in the model. There are two approaches to model checking.

Model checking algorithms are based on state space exploration. In order to verify a property, all possible behaviour of the model is checked. Although this brute force method is fully automatic, its
disadvantages are the high computation power required and the state explosion problem: The number of states grows very fast with respect to the complexity of the model (Clarke et al. 2012).

**Theorem Proving**

In contrast to brute force model checking methods theorem proving methods focus on reasoning about the model in order to prove a property. Correctness is achieved through the proof of generated mathematical conjecture from a model. Proofs produced by automated theorem proving describe how the conjecture follows from the axioms and hypotheses. Automated theorem provers are capable of solving complex problems but sometimes need to be guided in order to solve a proof in a reasonable amount of time (Sutcliffe).

### 3.2 Modelling in Event-B

In Event-B a system is divided into two main components, a dynamic and a static part. The dynamic part of a model is represented as a machine. A machine consists of a set of variables (states), set of events that change the state, and a set of invariants that describe properties that must hold during the execution of the system. An event consists of a set of guards and actions and may have a set of arguments used as inputs to the events. Guards are used to restrict the type of the arguments to the events and the properties that must be true to perform the actions.

The correctness of the system has to be proven meaning the invariants must always hold during the execution of the system. In order to do that proof obligations have to be proven in order to verify the correctness of the system. Formal verification in the context of Event-B means that all the necessary proof obligations must be proven, either by applying automated theorem provers or interactively directing the proofs.

The static part is represented as a context which serves as an environment for the machine. A machine can see several contexts. It contains a set of constants, sets and axioms. Sets and constants can be used to define types, or entities in a given context. Elements of a defined set can also be fixed in the context. Axioms are properties defined by predicates that are true for the static part of the
system. Axioms are defining properties regarding the sets and constants. A context can extend other contents, adding additional context information to the current existing context.

### 3.2.1 Refinements

Event-B is a formal method which follows a systematic incremental approach to build up a system model using refinements. The developments of a system starts by modelling an abstract representation of the system first, then systematically work through several refinement layers to reach the final blueprint of the system. Refinement levels are used to represent the system at different complexity levels, each level being closer to final complexity of the system. Consistency has to be always maintained between the levels meaning while adding more complexity to the system it never contradicts the model being refined. Figure 3.1 shows a high level structure of a club management system designed in Event-B. The purple boxes represent the machines, the grey boxes the contexts.

![Figure 3.1](image)

Variables of the abstract state are called abstract variables, while the variables of the refined state are called concrete variables. Similarly the machine of the refinement is called concrete machine, and the machine refined is called the abstract machine. The connection between an abstract variable and a concrete variable is expressed using a gluing invariant. Gluing invariants are used to relate the states of concrete and abstract machines. Figure 3.2 further examines the ‘refine’ relationship between the two machines.
In the model shown on Figure 3.2, the abstract machine m0 represents a high level abstraction of the system where Courses can be opened or closed.

In the refinement, extra functionality is present. Participants can be registered to certain courses and when the concrete ‘CloseCourses’ event is executed all participants are removed from the registered course. The notion of participants and instructors have not been present in the abstract machine, they only became defined in the concrete machine. On the diagram it can be seen, the variable ‘crs’ has been defined in the abstract machine, but is visible in the concrete machine as well. There was no need to define its type in the concrete machine as it has been already defined in the abstract one.

Another construct that can be used in a construction of a system are variants. Variants are used to express progression in the machine. When refining a machine new events can be introduced to the concrete machine. They may not refine any events of the abstract machine. Events can be specified to decrease the value of a given expression that has a lower bound. These events are called convergent events and the expression to be decreased is called the variant. A termination of an event can be ensured by introducing a variant. If the variant is specified, a convergent event must decrease it (Abrial and Voisin 2005). Events can be ordinary, convergent or anticipated. Ordinary
events do not have any requirements regarding the variant. Anticipated events may or may not decrease the variant, but cannot increase it. An event refining an abstract anticipated event must be anticipated or convergent. If an event is proved to be convergent, the refinement of it will be convergent as well because of guard strengthening (Jastram and Butler 2014). Guard strengthening is one of the conditions that have to be proved in a refinement process. It means that if the guards of a concrete event hold the abstract guards must hold as well.

**Refinement Verification**

Refinement verification is the part of the Event-B modelling process which defines the rules that have to be proven in order to verify that a concrete model indeed refines its abstraction. The following diagram illustrates the relationship between a model and its refinement.

![Refinement verification diagram](Abrial 2010)

G denotes the guards of the Abstract Event, H the guards of concrete event, I the abstract invariants and J the concrete invariants. To prove a correct refinement it has to be proven that the concrete event transforms the concrete variables w into w’, such that it does not contradict the abstract event. When a transition happens, the abstract variable v is turned into v’. v and w are related by the concrete invariant (Abrial 2010).

Proof obligations define what has to be proved for a model in order to be correct. To prove the correctness of refinement of models the following proof obligations has to be proved.

- **Guard strengthening (GRD):** The concrete guards are stronger than the abstract ones.
- **Invariant preservation (INV):** Each invariant in a machine is preserved by each event.
- Guard merging (MRG): The guards of a concrete event merging two abstract events are stronger than the disjunction of the abstract guards.
- Simulation rule (SIM): The actions of a concrete event do not contradict the actions of the abstract event.

While more and more complexity is added to the model at each level, the consistency has to be always maintained between the layers. The activity of introducing more concrete elements to the model is called horizontal refinement. When applying horizontal refinements, new proof obligations have to be proven in order to ensure the more concrete refinement does not invalidate the abstract one. Vertical refinements or data refinements are when some state is transformed but no new details are introduced to the model.

### 3.3 Abstractions

Fausto Giunchiglia describes abstraction informally as: The process of mapping a ground representation of a problem to a new abstract representation, which “helps to deal with the problem in the original search space by preserving certain desirable properties and is simpler to handle as it is constructed from the ground representation by “throwing away details” (Giunchiglia and Walsh 1992).”

He formalizes abstractions using mathematical notation, gives a formalized description of predicate abstractions and domain abstractions (mapping a theory onto a domain in the abstract theory) and gives a formal description of abstracting inconsistent/consistent models.

**Reduction and Abstraction Techniques in Model Checking**

A major problem in model checking that as the complexity of the model increases, the size of the state space increases exponentially. The state space may contain many redundancies, meaning having many equivalent states. One approach in reducing state space by applying a static analysis on the model and compute a reduced structure on-the-fly (Pelánek 2006).
Abstraction in Event-B

Reversing the refinement modelling approach of Event-B would mean, for an abstraction constructed it has to be proven that the machine on which the abstraction was applied is a refinement of the generated abstraction.

3.4 Rodin Platform

Rodin (Rigorous Open Development environment) is an Eclipse based development environment written in Java used for supporting the development of models using the Event-B method. It is an extensible environment. To support the development of Even-B models, Rodin has the following main tools integrated:

- Static Checker: Well-formedness and type checker of Event-B models
- Proof obligation (PO) generator: Automatically generates all the proof obligations that have to be proved, and generates sequents, which are sent to the provers for performing automatic or interactive proofs.

Several other tools including automatic theorem provers for example B4free prover, Pro-B animation and model checking tools have been implemented as plug-in to this environment. The operator framework that serves as a basis for this project and would be further developed to automate the abstraction and analysis of Event-B models is also a plug-in to Rodin.

Rodin keeps track of the status of a model in several different RootElement Types and files. These elements can be accessed from a Java/Scala program. These files are used in the Operator Framework currently.

<table>
<thead>
<tr>
<th>Event – B construct</th>
<th>Java class</th>
<th>File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-B Machine</td>
<td>IMachineRoot</td>
<td>.bum</td>
</tr>
<tr>
<td>Event-B Statically Checked Machine</td>
<td>ISCMachineRoot</td>
<td>.bcm</td>
</tr>
<tr>
<td>Event-B Context</td>
<td>IContextRoot</td>
<td>.buc</td>
</tr>
<tr>
<td>Event-B Statically Checked Context</td>
<td>ISCContextRoot</td>
<td>.bcc</td>
</tr>
</tbody>
</table>
3.4.1 Formula

Formula is the top level class for the representation of all Event-B formulas in Rodin. There are three classes extending it: Assignment, Expression, Predicate. They will have an important part in the implementation of abstractions. Each has several subclasses representing the different types of formulas. The main predicates subclasses are: AssociativePredicate, BinaryPredicate, RelationalPredicate etc. Similarly Expressions and Assignment have several subclasses. Every formula belongs to one of these types and has a tag that reveals more information about the given formula. For example a formula with a type of RelationalPredicate may have one of the following tags: EQUAL, NOTEQUAL, IN, NOTIN, LT, GT etc.

FormulaFactory is a class that provides a mechanism for constructing any kinds of formulas. In the construction of a formula type information of the elements in the formula can also be defined but it is not necessary. The tags, the types will play an important role in the mapping mechanism of the formulas when generating abstraction.

3.5 Scala

Scala is a programming language that has been used for developing the Operator framework. Scala programs are compiled to the same byte code as Java, and runs on the Java Virtual Machine, thus Scala programs can interact with code written in Java. The key aspects of the language:

- supports both object-oriented and functional programming paradigms
- supports functional abstraction and object oriented abstraction as well
- supports concurrency and asynchronous programming
- supports pattern matching
- allows definition of partial functions and higher-order functions: Functions can take functions as parameter (Odersky et al. 2004)
- Support multiple inheritance via trait

Implicit function definition provides a mechanism to apply a function on a given type without explicitly calling it. For example if an implicit function is defined that takes type ‘a’ and returns type
‘b’, in the scope of the code where this function is imported, it will be call implicitly on all objects of type ‘a’ and consider it as type ‘b’. This feature of the language is used in the new design of the operator language. Features of Scala such as implicit function declarations, algebraic data types, higher order functions makes available to write embedded user friendly domain specific languages in Scala. Domain Specific Language is a language tailored for a specific domain.

3.6 Operator Framework

In order to automate abstractions an Operator Framework has been developed as a plug-in to Rodin platform. The Operator Framework is available as a starting point for this project. It will be improved as the main objective of this project is to show that it is feasible to generate abstractions using the operator language, and it will involve changes of the current design and implementation. It includes the operator language that is used for making transformation of a model and serves as a core for this project.

3.6.1 Current Design and Implementation

The framework was aiming to provide an operator language and an integration of several other tools to perform an analysis on models written in Event-B. This section provides an overview of the design and implementation of the framework already developed and used as source of this project.

AST

The operator language operates on an intermediate AST representation of the Event-B model parsed from Rodin Elements. This representation is implemented internally in the framework in the purpose of making the implementation of transformations and operator language more flexible and tailored for generating transformations. Figure 3.4 shows the current design of the AST. This representation is parsed from the IMachineRoot and IContextRoot element of the Rodin Database which are not type checked by the internal static checker of Rodin Platform thus does not contain any type information about the variables, predicates and expression. These elements are retrieved as String and then parsed back individually to the corresponding formula using the FormulaFactory class. By
explicitly parsing back from String, type information about the different elements of the expression are not available.

*Variable, CConstant, Cset and Parameter are aliases of String

Figure 3.4 (Kovacs 2014)

This representation is parsed from the IMachineRoot and IContextRoot element of the Rodin Database which are not type checked by the internal static checker of Rodin Platform thus does not contain any type information about the variables, predicates and expression. These elements are retrieved as String and then parsed back individually to the corresponding formula using the FormulaFactory class. By explicitly parsing back from String, type information about the different elements of the expression are not available.

Rodin Builder

This part of the framework is responsible for generating the RodinElements from the intermediate AST described above. It provides functionality to create the Rodin constructs for every atomic elements of the AST. After transforming back to Rodin Representation static analysis can be run on the model that can be invoked from the framework and the result can be checked.
**Operator Language**

The aim of this language is to provide a set of operators that can be used efficiently to define new operators and could be applied on a model easily to generate transformations. One operator is responsible for one particular transformation, and to generate more complex transformation these operators can be applied together. The operators can be constrained by specifying the set of active events. The domain of the transformation will be the events specified. If they are not specified all the events are considered active (Kovacs 2014).

The operators currently available are:

- `deleteArgument` – Deletes one of the arguments of the active events.
- `deleteWitness` – Deletes one of the witnesses of the active events.
- `deleteGuard` – Deletes one of the guards of the active events.
- `deleteVariable` – Deletes a variable from the machine, replaces all elements of a predicate with True and deletes all actions that contains it.
- `deleteInvariant` – Deletes an invariant of the machine.
- `deleteAction` – Deletes one of the actions of the active events.
- `deleteEvent` – Deletes an event of the active events.
- `negateGuard` – Negates one guard of the active events.
- `negateAction` – Negates one action of the active events.
- `mergeEvents` – Merges two of the active events.
- `combineEvents` – Combines two events of the active events, which means creating a new event with the actions of event1 and the guards of event2.

Figure 3.5 shows an overview of the Operator language.
Each operator generates a certain type of transformation. The operators take an ‘Eval’ object as input and return an ‘Eval’ object as output. An ‘Eval’ contains the list of ‘MachineState’s generated by operators applied on the given ‘Eval’. In case of each transformation the ‘depth’ representing the number of operators applied can be retrieved. New operators can be defined by defining additional functions in the Operator object.

**Combinators**

Combinators being part of the operator language provides help to apply the operators following certain logic. They can be combined using Scala, but combinators were created to ease this. The following combinators are currently the part of the operator language. New operators can be defined by defining additional functions in the Combinator object.
• or - Returns transformations resulted from applying each operator of the input independently on the model.

• orCond - Apply one operator if a certain condition holds and another one if not.

• repeat - Apply an operator until a certain condition is true.

• first - Returns the first transformation generated by applying each operator (Kovacs 2014).

The following Scala code shows examples of using the Operator Language.

```scala
val res = Combinators.orCond(eval, Cond.hasAbstrablePartialFunction(_), Op.deleteGuard(_), Op.deleteAction(_))
```

Applying each operator independently:

```scala
val res = Combinators.or (Op.deleteGuard(_)), Op.deleteAction(_))
```

If the condition provided as the second parameter is true for an element of eval, the deleteGuard operator will be applied, if false then the deleteAction operator will be applied on each element of eval.

```scala
def transferAbstractCase(eval: Eval): Eval =
    eval Apply (Operator.deleteVariable(_), Operator.mergeEvents(_))
```

The current implementation of the Operator language in the framework does not fully exploit the features provided by Scala, and it is hard to combine and define new operators based on existing operators. In the new design, a basic DSL has been developed on top of the operators, which makes it easy to define new operators, extend the language and implement new semantics and rules.
IV. Analysis

Introduction
The analysis part of the project aimed at analyzing models in order to discover common patterns of representations and construction of refinements, and to examine how these patterns can be used in implementing abstractions. This chapter first provides an overview of the various models that have been used, discusses the discovered patterns present in these models then the applicability of these patterns regarding the objectives. The feasibility of implementation of these patterns using the Operator Framework is discussed in chapter V.

Model examples were taken from different sources, the Event-B repository, other research reports and proposals. In cases where the Event-B sources were not provided for Rodin they were manually constructed and the generated proof obligations have been interactively proved if required. The construction of such model (BookStore System) has resulted in modifications of the model compared to the source, but it was still suitable for analyzing and discovering patterns.

4.1 Overview of models

Club Management System
A system model of a Club Management System developed using the Event-B formalism. The system models the management of courses, participants and instructor. Courses can be opened / closed, participants can be registered to visit certain courses and courses can be assigned to instructors. This system is capable of the management of a finite number of courses that results in invariants that are expressed using universal quantification in the expression, to enforce properties on all occurrence of the entity. This property will be interesting in the case three scenario.

Location Access Controller
This system is able to control the access of people to different connected locations. Each person may have authorizations to enter certain rooms and may not have to enter other ones. The rooms are connected, and people can move from one location to another one. In this example a wide range of
Event-B constructs can be found, witnesses, gluing invariants, convergent, anticipated and ordinary events that makes it a good example for analysis.

**Online Book Store System**

This model have been developed in Rodin based on the example described by Carla Ferreira and Micheal Butler (Ferreira and Butler 2003). This is an electronic store model, specifically an online bookstore system, where each client in the store has a basket and may have an account set up as well. They can place books into their basket, and on successful payment they may leave the store, or exit the store and place the books back to the shelf.

Carla Ferreira and Micheal Butler followed a particular way to deal with the complexity of the system. To deal with the complexity of the construction of the model and determination of the gluing invariants, they constructed a model for a single client first following a refinement methodology. After they dealt with the construction of a single client model, they built the model for any number of concurrent clients which helped to prove the proof obligation of the more complex concurrent client model (Ferreira and Butler 2003). By starting to prove the proof obligations for a weak invariant then strengthening it in an incremental way they proved all the proof obligations. Replicating the strategies from a single client system helped them to prove most of the proof obligations for the more complex multiple client system.

This example can be interesting in the case three scenario, where transforming a model to a single entity model would help to speculate and fix the flaw. The Rodin source of this model was not available, so they were manually constructed based on the description with some modifications.

**Bank Transfer example**

This is simple version of bank transfer protocol which models transactions between bank accounts. This is a simplified version a more complex system and is very good for analyzing the high level of a
transaction system. Several kinds of abstraction and mutation of the base model is available in Rodin sources as well (Kovacs 2014).

**Mondex**

This is a system model of an electronic cash card system containing 10 layers of refinements[]. It is a money transfer system between accounts. The model describes interaction between actors of a given entity, cash cards. The communication pattern present in the model is very similar to the communication protocols in networks, for example the design of connection oriented communication. In this project the first two layers have been used for analysis.

**4.2 Patterns**

This section presents the different patterns discovered in the various models. Several different types of patterns will be described for example patterns of representation and construction of refinements.

**4.2.1 Refinement\Abstraction Patterns**

Event-B is a formal language that focuses on the development of model following a refinement approach that means the system is first described abstractly then incrementally refined. This approach involves the use of refinement constructs in the more concrete models that relates the elements of the concrete machine to the abstract machine. Refinement patterns were analyzed in order to define abstraction.

*Pattern 1: State refinement / State abstraction*

This example illustrates a structure taken from the location access controller model and present in the Mondex example as well. The following figure illustrates a refinement where a variable of an abstract machine is broken down into smaller parts in order to express more functionality.
This methodology introduces more subsets of a state variable in the refinement level in order to express more complexity and functionality. By enriching the machine state, new events may be introduced to the system relating to the new sub states. This refinement pattern was also described as “Set to partition” (Grov et al. 2012).

Reversing this pattern, and using this as an abstraction it can be applied on a model that initially is defined at a too complex level. The diagram below shows the pattern described above from an abstraction perspective.
Because Event-B is a modelling approach focusing on refinements not abstractions, an invariant has to be inserted into the concrete machine expressing that the union of ‘var1’, ‘var2’ and ‘var3’ are the subsets of the var0, which is the generated abstraction variable from the concrete variables.

In order to generate an abstraction of a model, and to express an abstraction relationship between two models following this pattern, the model being abstracted has to introduce the refinement constructs relating to the generated abstraction.

Steps to generate an abstraction of a given concrete model in Event-B:
1. Generate an abstraction of the model by merging variables.
2. Introduce refinement constructs in the concrete version, relating to the abstract elements.

**Occurrence of this pattern at Event level**

In order to make the presentation easier, typing guards of the parameters are expressed using parameter:Type notation instead of a separate event guard.

---

**Concrete machine**

<table>
<thead>
<tr>
<th>See:</th>
<th>context0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable:</td>
<td>var1, var2, var3,…</td>
</tr>
<tr>
<td>Invariants:</td>
<td>var0 ∈ ℙ(ENTITY)</td>
</tr>
<tr>
<td>Events:</td>
<td></td>
</tr>
<tr>
<td>- event_1(a: ENTITY)</td>
<td></td>
</tr>
<tr>
<td>guards:</td>
<td>a /∈ var0</td>
</tr>
<tr>
<td>actions:</td>
<td>var0 := var0 ∪ {a}</td>
</tr>
<tr>
<td>- new_event_1(a: ENTITY)</td>
<td></td>
</tr>
<tr>
<td>guards:</td>
<td>a ∈ var1, a /∈ var2</td>
</tr>
<tr>
<td>actions:</td>
<td>var1 := var1 {a}, var2 := var2 ∪ {a}</td>
</tr>
<tr>
<td>- event_2(a: ENTITY)</td>
<td></td>
</tr>
<tr>
<td>guards:</td>
<td>a ∈ var2, a /∈ var3</td>
</tr>
<tr>
<td>actions:</td>
<td>var2 := var2 {a}, var3 := var3 ∪ {a}</td>
</tr>
</tbody>
</table>

---

**Abstract machine**

<table>
<thead>
<tr>
<th>See:</th>
<th>context0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable:</td>
<td>var0</td>
</tr>
<tr>
<td>Invariants:</td>
<td>var0 = var1 ∪ var2 ∪ var3</td>
</tr>
<tr>
<td>Events:</td>
<td></td>
</tr>
<tr>
<td>- event_1(a: ENTITY)</td>
<td></td>
</tr>
<tr>
<td>guards:</td>
<td>a /∈ var0</td>
</tr>
<tr>
<td>actions:</td>
<td>var0 := var0 ∪ {a}</td>
</tr>
<tr>
<td>- new_event_1(a: ENTITY)</td>
<td></td>
</tr>
<tr>
<td>guards:</td>
<td>a ∈ var1, a /∈ var2</td>
</tr>
<tr>
<td>actions:</td>
<td>var1 := var1 {a}, var2 := var2 ∪ {a}</td>
</tr>
<tr>
<td>- event_2(a: ENTITY)</td>
<td></td>
</tr>
<tr>
<td>guards:</td>
<td>a ∈ var2, a /∈ var3</td>
</tr>
<tr>
<td>actions:</td>
<td>var2 := var2 {a}, var3 := var3 ∪ {a}</td>
</tr>
</tbody>
</table>

---

Figure 4.3: State abstraction
Pattern 2: Set extension / Domain abstraction

The following horizontal refinement pattern can be seen in the Club Management System, where the notion of ‘COURSE’ has been introduced in the abstract machine, while the participants and the variables representing the connection between the participants and the courses have been introduced in the refined machine. This pattern can be discovered in the Mondex model as well, where Purses are introduced in the abstract machine M1, and more properties regarding the purses are defined in the refined concrete machine M2. The following diagram shows the high level overview of this refinement pattern.

This methodology introduces the domain of the property in an abstract machine, and adds properties to the domain using functions in the refinement. The refined machine uses a state variable of the abstract machine as a domain to the function. Reversing this approach, and using it as an abstraction pattern, similarly to the pervious one it requires the introduction of refinement constructs in the concrete machine, however it also requires other precise mutations such as substitution of the domain of the concrete function.

Figure 4.5 shows a simple example when applying this pattern on an initially “complex” model.
Applying the described pattern shown on Figure 4.5 would result in the concrete and abstract machines described on Figure 4.4. The partial function would be transformed to a total function, because the partial function represents that not every instance of \textit{ENTITY} may be mapped to \textit{ENTITY2}. This will be captured by the abstract set ‘\textit{var0}’. An instances of \textit{ENTITY} that are mapped in the concrete machine is element of the ‘\textit{var0}’ set of the abstract machine. If it is element of the set on the abstract level it has to be mapped in the concrete level. This is equivalent to the partial function representation, it may not be element of domain but if it is then it is mapped.

Steps to generate an abstraction of a too concrete model in Event-B:

1. Apply abstractions on the model, and generate an abstracted version of the model.

2. Generate transformation of the complex model using constructs to refine elements of the abstract parent model, substitute the domain with the abstract variable, and transform it to total function.
Figure 4.6 shows another variation of this refinement methodology. In that case the concrete machine expresses the connectedness of the domain by introducing a gluing invariant that describes that the domain of the function is equivalent to the abstract variable.

![Concrete machine](image1)

**Concrete machine**

- **Sees:** context1
- **Variables:** var1
- **Invariants:** var1 ∈ ENTITY → ENTITY2
  \[ \text{dom(var1)} = \text{var0} \]
- **Events:**
  - Event1(p:ENTITY, x:ENTITY2)
    - **Guards:** p ∉ dom(var1)
    - **Actions:** var1(p) := x
  - Event2(p:ENTITY)
    - **Guards:** p ∈ dom(var1)
    - **Actions:**
      - ref_var := \{p\} << ref_var

![Abstract machine](image2)

**Abstract machine**

- **Sees:** context0
- **Variables:** var0
- **Invariants:** var0 ∈ \mathcal{P}(ENTITY)
- **Events:**
  - Event1(p:ENTITY)
    - **Guards:** p ∉ var0
    - **Actions:** var0 := var0 ∪ \{p\}
  - Event2(p:ENTITY)
    - **Guards:** p ∈ var1
    - **Actions:** var1 := var1 \{p\}

```
<< | - domain subtraction
```

Figure 4.6: Set refining via gluing invariant

The application of this variation does not require the modification of existing expressions in the concrete machine. In this case an addition of a gluing invariant would relate the domain to the abstract variable.

Steps to generate this refinement hierarchy from the too concrete model shown on Figure 4.5

1. **Apply abstractions on the model, and generate an abstracted version of the model.**
2. **Generate transformation of the complex model using constructs to refine elements of the abstract parent model, introduce a gluing invariant in the concrete machine**

These refinement patterns may be applied on models defined at a too concrete level and generate a refinement hierarchies from the initial model, in other words transform the model to a refinement of an abstract model generated from it.
4.2.2 Collection patterns

The models listed above have one common similarity. They are all based around collections. Collections are essential in system modelling. They are the basic structures in constructing software. There are two main purposes collection is used for. One is to express that a single instance can have more properties, the other is more than one instance of the entity can exists at a given time.

The online bookstore example and the club management system are both based around a collection of an entity, people in case of the bookstore, and courses in case of the club management system. In these examples the instances of the entity, call them clients are not interacting with each other, they are present in a shared system state but they do not have any interaction between each other. This results in a less complex system model, compared to the Mondex and Bank transfer example where the main characteristics and functionalities of the system comes from the interaction of the clients.

Transforming a multiple client system model to a single client model would significantly reduce the complexity of the model and may help to speculate and prove the source of a flaw if exists. The way a collection is represented, (e.g. the constructs used: set, partial function or total function etc) results in different meaning semantically. Several patterns of collections are described here where there is no interaction between the individual ‘clients’. For each pattern a methodology of turning it to single client system is also shown.

Collection Pattern 1: Simple Set collection

The most basic way of expressing a collection is a set. A set can be used to represent the current active/present instances of an entity (clients) in the system. The following figure shows a simple overview of the pattern and a single client equivalent of it.
Figure 4.7 shows a very abstract representation of a client entering/leaving the store/club, so the transformation to a single client model is straightforward. The set variable would be transformed to a Boolean variable that represents if the single client is in the state or not.

**Collection Pattern 2: Partial Functions**

The presence of a client is often represented by a use of partial function. A partial function is a function that may not map all the elements of the domain to a value. This property of the partial function is often used to assign properties to the domain values (clients) if they are present. This pattern is the extension of the previous one. In this case more properties are expressed such as a client may have different properties or can hold elements of other entity. Figure 4.8 shows a possible transformation of a model to a single client model where this pattern is present.
In the transformed single client model, similarly to the simple set pattern the domain of the partial function is transformed to a single value, in this case a Boolean value that represents the presence of the single client. However in this case it needs to be expressed that if the ‘domvar1’ is false then var1 is null or empty. Similarly if domvar2 is false then var2 does not hold any value, is null.

The model shown on Figure 4.8 is a very simple scenario however there may exist different relationships between the domains of each function. For example if it expressed the domains equal to each other, it means when a client is inserted to one (enters the system), then it is inserted to all relations. Table 4.1 shows different relationships that may exist between the relations and the transition of these to a single client equivalent.
4.3 Application of patterns in terms of objectives

This section intends to show how applying the described patterns on specific models will help to generate abstraction, discard flaws or focus on flaws described in the project objectives. Only the model transformations are shown here, by applying the patterns described in section 4.2. Implementation details can be found in Chapter VI.

4.3.1 Case 1 and 2 – Discarding flaw by abstraction

The following scenario tries to imitate a common problem during the construction of a system, such that the design is started at a too concrete level. The working example is a simplified implementation of a Bookstore system in Rodin, constructed during this project based on a report (Michael Butler).

In the following model there are four variables, ‘basket’, ‘budget’, ‘shelf’ and ‘price’. The domain of the basket and budget variables represents the current online clients. The domain of these variables are equivalent which is ensured by an invariant ‘dom(basket) = dom(budget)’ invariant. When a client arrives, it is added to these variables. Put_In_Basket event represents the event when a client places an item to its basket. The Pay event is a very abstract representation of the payment, it empties the basket and the customer can leave via the Exit event or decide to continue to browse.
the store. ‘Exit’ event places all the books back to the shelf that has not been paid for and the client leaves the store.

**BookStore Machine**

| Sees: context0: SETS: BOOK,CLIENT |
|-------------------|-------------------|-------------------|
| Variables: basket | Invariants: basket ∈ CLIENT → I(BOOK) |
| budget            | budget ∈ CLIENT → ℕ |
| shelf             | shelf ∈ BOOK → ℕ |
| price             | price ∈ BOOK → ℕ1 |
|                   | dom(basket) = dom(budget) |

Events:

- **Arrive(a: N1, c: CLIENT)**
  - guards: $c \notin \text{dom(basket)}$
  - actions:
    - basket := basket U (c↦∅)
    - budget := budget U (c↦a)

- **Put_In_Basket(b: BOOK, c: CLIENT)**
  - guards: $c \in \text{dom(basket)}$
  - actions:
    - shelf(b) := shelf(b) − 1
    - basket(c) := basket(c) U (b)
    - budget(c) := budget(c) − price(b)

- **Pay(c: CLIENT)**
  - guards: $c \in \text{dom(basket)}$
  - actions:
    - basket(c) := ∅
    - budget(c) := budget(c) − price(b)
    - shelf(b) := shelf(b) + (λbook·book ∈ basket(c) | shelf(book) +1)

- **Exit(c: CLIENT)**
  - guards: $c \in \text{dom(basket)}$
  - actions:
    - basket := [c]− [basket]
    - budget := [c]− [budget]

**Figure 4.9**

Generating an abstraction of this model, following pattern 2, the model can be scaled out to several machines, giving a clearer structure to the model, which can be beneficial in case of a more complex system. Figure 4.10 shows the generated abstraction and the refinement of the abstraction of this model.
The generated abstraction represents a viewpoint which is further from the real model. It models the clients entering and leaving the bookstore. This abstraction gap results in loss of details, and may discard potential flaws present in the model related to the Put_In_Basket event. In a scenario where the Arrive event does not add the client ‘c’ to the budget variable, certain proof obligations would fail. By applying an abstraction on the model it would be easier to notice that when the client arrives, is not added to one of the variables however the invariant expresses that all variables should be equal.

To express the refinement relationship, refines constructs have to be inserted between the Event-B elements. In this case also a gluing invariant, that expresses that the domain of the concrete variables budget and basket equals to abstract ‘basket_clients’ and ‘budget_clients’ variable.

4.3.2 Case 3 – Toy Example – Focusing the flaw

The following methodology uses transformation/mutation of a given model in order to focus the possible flaws present in the system. The working example will be the same model used in the
previous example as well, which may have a flaw present in it. In the concrete model the variable ‘basket’ is a partial-function from CLIENT to power set of books. The variable budget is a partial function from CLIENT to the set of natural number. The domain of these variables equals to each other and represents the present clients in the system. Variable ‘shelf’ represents the number of books in the bookstore and ‘price’ is the price of the books. This model shown on Figure 4.9 has a collection pattern described in section 4.2.2. The model shown on Figure 4.11 describes the single client equivalent of the collection model shown on Figure 4.9.

Figure 4.11

The transformation of this model to a single client may help to find the source of the flaw. A thorough inspection of the system can be difficult in case of parallel clients. As the clients are not interacting with each other a single client model would still contain the possible flaw. In case of the mondex example this is different as the system is aimed at modelling the interaction of the clients.
V. Operator Framework - design and implementation

In this chapter, the limitations of the operator framework available at the start of this project is highlighted first, then a description of the new design is given that was necessary and has been developed in order to address the issues that came to surface during the use of the existing implementations.

The difficulties and the issues appeared during the implementation of abstractions. Initially small solutions, additional classes have been developed in order to provide a solution for a specific problem, however as more problems came to surface a redesign and reimplementation of the AST model representation and the Operator language was needed which involved changes of another components as well such as the RodinParser that builds the intermediate AST representation and RodinBuilder that transforms back the intermediate representation to Rodin platform.

Methodology

In the beginning of the project it was known that the implementation of an abstraction may be difficult and improvements would be needed in order to achieve a meaningful implementation. One of the aims of this project was to evaluate the usefulness of the Operator Framework in generating transformation and improve it in order to make it suitable for generating transformation. However, it was not possible to identify in advance the requirements, as most of the needs for changes came up during the implementation thus an experimental methodology were more beneficial.

5.1 Limitations of Original framework and tool

AST and Operator Language

When implementing a specific abstraction, it appeared that the AST representation (described in 3.6.1) that is the core part of the Operator Framework was very inflexible in terms of transformations. This was visible in the length, complexity and readability of already implemented Operators, and became more problematic during the implementation of new operators and complex transformations, abstractions.
5.2 New Design

5.2.1 Model Representation

In a refinement chain of Event-B Machines and Contexts, where there are different relationships between the constructs such as ‘refine’, ‘sees’, ‘extends’, ‘witnesses’, it may be hard to implement transformations on an AST that has a solid structure, especially when the transformation involves a lot of structural changes such as adding / removing constructs and relationships. The design of the classes representing the intermediate AST of an Event-B model implicitly included the correctness of the high level structure, for example: a machine has events, an event has guards, an event has actions. This representation had a solid nested structure. In order to find and change an atomic element located at the bottom of the AST for example action, would mean the traverse of all events, then the traverse of all of its action. The removal of all the actions that meets a certain property would mean the copy of the event without that given action. The immutable structure and the copy method provided by scala case classes, was very efficient in terms of memory usage, because only the creation and of new objects resulted in allocation of new memory, all the other elements existed only once in the memory. This made available the creation of a large amount of transformations, however this made the implementation of Operators and transformations really long and complex.

Abstract Syntax Graph and Triples

Instead of an AST, the model having selected gets parsed into triples that together build up a graph of the model. A triple, in this context named EventBTriple, is built up of a subject – the source of the relation, a property – the type of the relationship and an object – the target of the relationship. An EventBTriple is similar to an RDF triple used in Semantic Technologies. RDF(Resource Description Framework) is a framework to describe, represent and link information using triples so it can be better processed and understood by machines.

```scala
case class EventBTriple(subj:Element, pred:Prop, obj:Element)
```

The subject and an object of an EventBTriple can be any type of Element. An element can be any element of an Event-B Model. Figure 5.1 shows the hierarchy of Elements.
Trait in Scala is similar to an interface in Java. It defines the methods that the implementing classes have to implement or can provide a default implementation for the method as well. It also provides an additional type property for the implementing classes which is the main reason it is used here. The ‘matches’ method implemented in the top level Element, and may be overridden in its implemented classes plays a key role in the traversal of the graph, since it is used for finding matching element. Figure 5.2 shows the Elements of type ProjectElement.

Class diagrams of the different elements, Machine, Context, Event, Action and Guard can be found in Appendix A.

A model in the intermediate representation is essentially a set of elements with relationships described between them. On a selection of a RodinElement / Event-B component, all the components gets parsed that the given element refers to, and added to the EventBTriple store, called a Model. In the implementation level this is present as a linked set of EventBTriples. A linked
set retains the order in which the elements are inserted to the set, thus the order of elements can be retrieved when transforming back to RodinElements. Figure 5.3 shows a graph of a model represented as a set of EventBTriples.

Figure 5.3

The nodes of the graph are types of Element, the edges between them are the relationships, links described by the Triple. These relationships currently can be: refines, sees, extends, hasVar, hasProperty, hasVariant, hasEvent, isElemOf, hasArg, isGuardedBy, Acts, hasWitness, isTypeOf, hasCons, hasSet, hasAxiom, refinesEvent, exists. The relation isTypeOf is used for defining the type of arguments and variables. The typing invariants and typing guards are filtered on the traversal of the RodinElement. isElemOf is used for defining RelationalPredicates with the “∈” symbols.

5.2.2 EventBTriple Query Language

A query language has been built on top of the graph. This language can be used to retrieve triples of the graph that satisfies the specified conditions or matches the specified patterns. Several examples of queries are shown below. They are written in Scala code and can be used to retrieve elements/triples from the graph.

The query below returns all triples expressing refine relationship between a node and Machine named “m1”. EventBQl is an object which is the main managing unit of query language. Blank is a keyword (implemented as object) used to match any Element in the graph.

```
EventBQl Select Where (Blank, Prop.refines, Machine("m1")) From model
```
The following query returns all guards of those events that satisfy a condition. The number in the filter clause refers to index of the Element in the Triple for which the condition must hold.

```
EventBQIL Select Where (Blank, Prop.isGuardedBy, Blank) filter (1, condition) From model
```

‘condition’ can be any function that takes an Element and returns a Boolean.

In the design and implementation of the underlying graph, triple data structure and the implemented query language, code snippets and ideas have been used from an open source project, called Scardf. Scardf provides an API for writing and querying RDF graphs (https://code.google.com/p/scardf/).

### 5.2.3 Operator Language

The Operator Language has been built and designed following a methodology of building a domain specific language in Scala. The new design of the language is based on several different DSLs written in different areas and code snippets and ideas has been used and tailored for this domain from different resources. A usability of an internal DSL of an asset management system (Debasish, Designing Internal DSLs in Scala) and the scalability of a basic DSL (Greg, Functional Thursday) has been used to redesign and construct the Operator language. The new Operator language highly utilizes implicit function definitions, algebraic data types, and higher order functions, in contrast with the old one that only took advantage of higher order functions.

**Design**

The low level design of the Operator language is built up of two main components: the builder and the evaluator. The builder is an Object called `EventBOpLang`, which provides the interface for the language and builds up the nested object-language using algebraic data types. The evaluator `EventBopEvaluator` evaluates the built up object-language. The following sentence written in Scala code describes a transformation that applies the Operator ‘`moveVariableToAbstractLayer`’ if the model has an abstractlayer, otherwise applies the Operator ‘`insertAbstractLayer`’ then ‘`moveVariableToAbstractLayer`’ on the model given.
This sentence in the background is built up using the rule 'IfElseRule' by the 'EventBOpLang'.

case class IfElseRule(condition: Condition, action: Action, else_Act: Action) extends Rule

IfElseRule(
    ModCond(Cond.hasAbstractLayer),
    ConcreteAction(Op.moveVariableToAbstractLayer),
    AndAction(
        ConcreteAction(Op.addAbstractLayer),
        ConcreteAction(Op.moveVariableToAbstractLayer)
    )
)

On the execution which is invoked by the ‘On’ function (taking model as an argument), this Rule is evaluated by EventBOpEvaluator class on a model.

Conditions can be combined using ‘or’ and ‘and’ and ‘not’ operators and can be infinitely nested. An ‘Action’ represents an application of Operator on the model. Actions can be also nested, and combined using ‘andApply’ and ‘parAnd’ operators. Op1 andApply Op2 represents the application of Op1 first then Op2 on the resulted transformations. Op1 parAnd Op2 represents the application of Op1 and Op2 independently on the model and returns the resulted transformations. This is equivalent to the ‘Or’ and ‘And’ combinators in the previous design (See section 3.6.1).

The production rules currently available are:

- BasicRule – Applies an Action on the model
- IfRule – Applies an Action on the model if certain condition is true
- IfElseRule- Applies an Action on the model if certain condition is true otherwise applies another Action
- RepeatUntilRule - Applies an Action on the model until a certain condition is true. Returns the last generated transformations on which the Condition was still true.
- WhileRule - Applies an Action on the model until a certain condition is true. Returns the last generated transformation on which the Condition was already false.

- AccumulatorRule - Applies an Action on the model until a certain condition is true. Returns all the elements generated until the Condition became false.

Two of these rules(IfElseRule,RepeatUntilRule) have been part of the already implemented Operator language present as Combinators, however it was more difficult to read and write combinations of Operators.

The new language is extensible, new elements and production rules can be added to the language. However on an addition of a production rule a new builder has to be added to the EventBOpLang object as well. This is necessary in order to be able to construct transformation in a more writable and readable way.

Similarly to the original design, elements of the model can be set as an active component that provides operators extra information on the domain they work. Some operators require the specification of an active element.

Using this language, now it is possible to define new operators following more complex logic, and use existing operators in the implementation of new operators. To define a new Operator and be able to use it the EventBOpLang language, simply a new function definition needs to be added to the ‘Op’ object. ‘Op’ object contains all the Operators. Operators already implemented in the previous framework have been integrated to this language. In the following scenario we would like to use the transformation described above as an individual Operator and use it in combination with other Operators. In order to do that the following definitions needs to be added to the ‘Op’ object. This function takes a model as an argument and returns a list of models.

```python
def abstractVariable = (model:Model) =>
EventBOpLang
   If (Cond.hasAbstractLayer)
      Then (Op.moveVariableToAbstractLayer)
```
Now this type of transformation can be used as an individual Operator, and may be used in implementing future transformations. The example below shows the use of this Operator in defining a new transformation.

```
EventB opi ng While (Cond.hasVariablesOfSameType)
Accumulate (Op.mergeVariable andApply Op.abstractVariable)
On model WithActiveElements Machine("m1")
```

This example was for illustration purposes of the language. This transformation would merge two variables of the machine called “m1” if it has variables of same type then apply the newly defined abstractVariable operator. It does this until the machine has two variables of same type and in the end it returns all the models generated during the iterations. In each iteration it generates all possible combinations of merging two variables and moving a variable to the abstract machine. In the end it may result in duplications of generated models. Figure shows the activity diagram of the WhileAccumulator using the described transformations.

![Activity Diagram](image)

When defining a While Accumulator the applied operators to be applied have to modify the model in a way that the condition converges towards being false, otherwise it would result in an infinite loop.

**Factory**

During the implementation of abstractions it is necessary to perform Formula transformations as well. Formula and FormulaFactory is part of org.eventb.core.ast package (See section3.4.1)
FormulaFactory class provides functionalities to construct new formulas. There are functionalities in this package to substitute a given identifier with an expression, however in order to perform transformations such as substitute a given expression with another expression in a formula new classes have been implemented. Object ExpressionFactory and PredicateFactory can be used to map expression present in predicates and expression to other expression. AssignmentFactory has been implemented to map assignments to other assignments. It has one mapfunction called ‘mapParfAssignment’ which aids the implementation of mapping assignment when a partial function is abstracted.

**EventBParser**

The parser has been integrated to build intermediate representation using the EventBTriples and Graph. The underlying data structure of the graph is a LinkedHashSet. This data structure retains the order in which the elements have been inserted. This is important when transforming back to Rodin representation as the order of the elements in Event-B is important to pass type checking.

In contrast with the previous parser, the model is parsed from the static checked RodinFiles(See section 3.4). This makes available to retrieve the type information of any elements from the formulas.

**EventBCreator and RodinBuilder**

This class is responsible for transforming back the intermediate representation to Rodin representation required only minor changes to fit into the new implementation. The interface of the Elements did not change significantly. For example a Machine is stored as a simple node (Machine("name")) in the triple set. The variables are not stored as part of the Machine object, but it still provides a method getVariables to retrieve the variables of the given machine from the triple set.
VI. Implementation

This section first provides a description of new implemented Operators and Conditions then describes the combination of these using the Operator language to implement the transformations and abstractions shown in the Analysis part.

Conditions

Conditions are implemented in the Condition object. This object serves as a collector for the conditions that can be used when applying Operators. These are simple functions taking a model and returning a Boolean.

\textit{Cond.hasAbstractLayer} – returns true if the machine set as active in the model has an abstract layer, otherwise false.

\textit{Cond.hasVariableOfTypePartialFunction} – returns true if the machine set as active has a variable of type PartialFunction.

\textit{Cond.hasVariableOfTypePowerset} – returns true if the machine set as active in the model has a variable of type powerset of anything.

\textit{Cond.hasAbstractablePartialFunctionVariable} – returns true if there exists a variable in the concrete machine that has a type of partial function, and there not exists a gluing invariant that makes the domain of the concrete variable equal to an abstract variable.

\textit{Cond.hasVariablesWithSameType} – returns true if the machine set as active in the model has at least two variables with same type.

\textit{Cond.hasOneEvent} – returns true if the active machine has at least on event.

Operators

The operators available at the start of this project have been integrated to the new design and the following operators have been added. For the following operators to be applicable one machine has to be set as active in the graph.

\textit{Op.insertAbstractLayer} – If the machine set as active does not have an abstract layer, it inserts an empty one above it and adds a ‘refines’ construct to the active machine. If it does have it inserts an
intermediate layer between the machine and its abstract layer, and adds the necessary refine constructs. The layers inserted are empty machines.

**Op.addAbstractLayer** – If the machine set as active does not have an abstract layer, it inserts an empty one above it and adds a refines construct to the active machine. It also adds the same sees construct to the abstract layer that the concrete layer has. If it does have an abstract layer it does not do anything. This operator is implemented using other operators.

```plaintext
EventBOpLang If (not(Cond.hasAbstractLayer)) Then (Op.insertAbstractLayer andApply Op.addSeesContextToAbstractLayer) On model
```

**Op.addSeesContextToAbstractLayer** – Adds the sees construct of the active machine to its abstract layer. Requirement: The Machine set as active must have an abstract layer.

**Op.moveVariableToAbstractLayer** – It moves a variable of the Machine to its abstract layer.

Requirement: The Machine set as active must have an abstract layer.

**Op.mergeVariable** – Merges two variables of the same type in the machine, and replaces all references to the variables with the new variable.

**Op.extractDomainOfPartialFunction** – Creates a variable in the machine with the powerset type of a domain of a partial function, then it replaces all references to the domain with the new extracted variable. Example:

```
<table>
<thead>
<tr>
<th>machine_0</th>
<th>apply Operator</th>
<th>machine_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>var0 ∈ ENTITY0 → ENTITY1</td>
<td></td>
<td>var1 ∈ ℙ(ENTITY0)</td>
</tr>
<tr>
<td>Predicates before</td>
<td></td>
<td>var0 ∈ var1 → ENTITY1</td>
</tr>
<tr>
<td>c ∈ dom(var0)</td>
<td></td>
<td>after</td>
</tr>
<tr>
<td>c ∈ dom(var0)</td>
<td></td>
<td>c ∈ var1</td>
</tr>
<tr>
<td>dom(var0)=varx</td>
<td></td>
<td>c ∈ var1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>var1 = varx</td>
</tr>
</tbody>
</table>
```

**Figure 6.1**

**Op.abstractDomainOfPartialFunction** – Creates a variable with the power set type of a domain of a partial function, places it to the abstract layer then introduces an invariant in the concrete machine that glues the created abstract variable to the domain of the concrete variable. Requirement: the
machine set as active must have an abstract layer. An empty abstract layer can be inserted using the
insert abstract layer operator.

<table>
<thead>
<tr>
<th>machine_0</th>
<th>apply Operator</th>
<th>abstract_machine_0</th>
<th>machine_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1 ∈ ENTITY0 ↔ ENTITY1</td>
<td>abs_var ∈ ℙ(ENTITY0)</td>
<td>var1 ∈ ENTITY0 ↔ ENTITY1</td>
<td></td>
</tr>
<tr>
<td>dom(var1) = abs_var</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2

This operator also abstracts the events by mapping the concrete guards and actions to an abstract
equivalent. The following tables summarize the currently supported forms of guards and actions that
the implementation can currently map/abstract. Actions and guards that contain variables that are
not present in the abstract layer are discarded.

### Guards

<table>
<thead>
<tr>
<th>concrete machine</th>
<th>apply Operator</th>
<th>abstract machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>c ∈ dom(var1)</td>
<td>-</td>
<td>c ∈ abs_var</td>
</tr>
<tr>
<td>c ∈ dom(var1)</td>
<td>-</td>
<td>c ∈ abs_var</td>
</tr>
</tbody>
</table>

### Actions

<table>
<thead>
<tr>
<th>concrete machine</th>
<th>apply Operator</th>
<th>abstract machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1 := {c} &lt;−</td>
<td>var1</td>
<td>abs_var := abs_var{c}</td>
</tr>
<tr>
<td>var1 := var1{c→ a}</td>
<td>-</td>
<td>abs_var := abs_var{c}</td>
</tr>
<tr>
<td>var1 := var1 U {c→ a}</td>
<td>-</td>
<td>abs_var := abs_var{c}</td>
</tr>
</tbody>
</table>

Figure 6.3

**Op.mergePowerSetVariable** – Invokes mergeVariable operator while the model has a variable of
type powerset. This Operator is constructed as a combination of other operators.

```python
def mergePowerSetVariable = (model:Model) =>
    EventBOpLang While (Cond.hasVariableOfTypePowerset)
        Accumulate (Op.mergeVariable)
        On model
```
The following operator has been implemented in a separate object called Transformer. It can be used in the language in the same way as the Operators.

Transformer.transformCollectionToSingle(entity: String) – Transforms all variables that has a type of partial functions and the domain is a set entity, to a single client variable. Figure 6.4 demonstrates the mapping of variables and their corresponding typing invariant transformation when Transformer.transformCollectionToSingle(“ENTITY”) is applied.

<table>
<thead>
<tr>
<th>machine_0</th>
<th>apply Operator</th>
<th>machine_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1 ∈ ENTITY0 → ℙ(ENTITY1)</td>
<td></td>
<td>var1_dom ∈ BOOL var1 ∈ ℙ(ENTITY1)</td>
</tr>
<tr>
<td>var2 ∈ ENTITY0 → ENTITY2</td>
<td></td>
<td>var2_dom ∈ BOOL var2 ∈ ENTITY2</td>
</tr>
<tr>
<td>dom(var1)=dom(var2)</td>
<td></td>
<td>basket_dom=budget_dom</td>
</tr>
</tbody>
</table>

Figure 6.4

This transformer invokes a call to ‘transformCollectionGuards’ and ‘transformCollectionActions’ that are transforming the guards and actions to their single client equivalent. The following table summarizes the forms of actions and guards that the current implementation is able to map. The name of the elements are for representative purposes.

Transformer.transformCollectionGuards

<table>
<thead>
<tr>
<th>Guards</th>
<th>before</th>
<th>apply Operator</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>c∈dom(var1)</td>
<td>c∈dom(var1)</td>
<td></td>
<td>var1_dom = FALSE var1_dom = TRUE</td>
</tr>
</tbody>
</table>

Figure 6.5

The parameters of type ENTITY are removed as a part of transformCollectionGuards function.

Transformer.transformCollectionActions

<table>
<thead>
<tr>
<th>Actions</th>
<th>before</th>
<th>apply Operator</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1 := var1 U {c→∅}</td>
<td></td>
<td></td>
<td>var1_dom := TRUE var1 := ∅</td>
</tr>
<tr>
<td>var2 := var2 U {c→ a}</td>
<td></td>
<td></td>
<td>var2_dom := TRUE var2 := a</td>
</tr>
<tr>
<td>var1 := {c} ⊑</td>
<td>- var1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var2 := {c} ⊑</td>
<td>- var2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.6
Implementation of Case examples

Case 1 and Case 2 Example

The following scala code shows the implementation of partial function abstraction (described in section 4.3.1) using the implemented Operators, Conditions and Operator language.

```scala
def abstractPartialFunctions = (model: Model) =>
  EventBOpLang While (Cond.hasAbstractablePartialFunctionVariable)
  On model
```

The following statement invokes this on the bookstore model.

```scala
EventBOpLang Apply (Op.abstractPartialFunctions)
  On parsedModel WithActiveElements Machine("bookstore_0")
```

‘parsedModel’ is a variable referencing the parsed graph of the Event-B model. The bookstore that this is applied on has one machine called “bookstore_0”, and one context “c_0”. It has two variables with a type of partial function. Until the model has an abstractable partial function it inserts an abstract layer and abstracts the domain of the partial function. This statement results in two generated abstraction. Although these abstractions are identical they are generated in two different paths. The generated abstraction can be found in Appendix B:Bookstore_Example.

Changing the ‘Apply’ keyword to ‘Accumulate’ keyword would return all the four generated models shown in the above graph. In the first iteration two models are generated, one with the ‘basket’ and one with the ‘budget’ variable abstracted. In the second iteration two models are generated, each of them having both the basket and budget variables abstracted. The final model where both ‘basket’ and ‘budget’ variables are abstracted is generated twice but on different paths. This resulted in duplication, but all possible combinations of partial function abstractions are generated. Using the ‘apply’ keyword to ‘Apply’ keyword in the first statement would return the two identical models generated in the last iteration of the accumulator, where both ‘basket’ and ‘budget’ variables are abstracted.
**Case 3 Example**

The implementation of the collection transformation does not deal with refinement hierarchies. It can be applied on a model where there is one single machine and its context. The predicates, expressions and assignments it can transform are described on. The transformer function takes one parameter, the entity that the collection is based on. The scala code to transform the bookStore example to a single client version:

```scala
EventBOpLang  Apply (Transformer.transformCollectionToSingle("CLIENT"))
On model WithActiveElements Machine("bookstore_0")
```

This statement returns one generated transformation, where all the variables that has type of partial function with the domain of type “CLIENT” are transformed. Actions and guards that are affected are also transformed following the methodology shown on Figure 6.6, however mapping some forms of guard are not supported. From the invariants only the transformation of typing invariants are supported at the moment. The generated transformation can be found in Appendix B:BookStore Example - Single Client
VII. Evaluation

This chapter will evaluate the implemented operators, the operator language and the framework. First the implemented operators will be rated on different metrics, then the efficiency of the new operator language design will be compared to the old design, finally the suitability of the Operator framework in automating abstractions will be evaluated.

7.1 Operators

The book store model has been used as a working example to implement operators and conditions in order to describe an abstraction and mutation. The models used for evaluating the implementations are the Club Management System, Location Access Controller System, Mondex System and the Bank Transfer system. The operators are split to two categories: Atomic and Complex Operators.

Atomic Operators

The following operators are categorized as atomic operators because they perform atomic transformations. They add/move/copy/extract simple constructs of a model:

- Op.insertAbstractLayer
- Op.addAbstractLayer
- Op.addSeesContextToAbstractLayer
- Op.moveVariableToAbstractLayer
- Op.extractDomainOfPartialFunction

The operators available at the start of the project and later integrated to the new language are also categorized as atomic operators.

During the implementation of atomic operators they were continuously tested on the models to test if they perform the desired behaviours in any environment. A basic testing suite has been implemented in the framework to write test cases for the operator language.

The first three operators when applied generate one model each. Op.moveVariableToAbstractLayer generates as many models as many variables the active machine has in the model.
Op.extractDomainOfPartialFunction generates as many models as many partial functions the active machine has. The number of models generated may depend on the condition and the rule used: while, until, if.

**Complex Operators**

The following operators describe more complex transformations. They either use other operators or implement more complex transformations that may affect the whole structure of a model.

- Op.mergeVariable
- Op.abstractDomainOfPartialFunction
- Transformer.transformCollectionToSingle

Applicability of each operator is discussed in case of each model and if applicable the following results were recorded.

- Number of generated models
- Number of models passed type checking
- Generation time
- Type checking time
- Number of meaningful models

These operators also perform formula transformations. They support the transformations of a few forms of Assignments and Predicates and Expression (described in section x.x), but it can be extended. The following tests were performed to measure the efficiency of formula transformations.

- Number of successful invariant transformation/total number of invariants containing the element
- Number of successful guard transformation/total number of guards containing the element
- Number of successful action transformation/total number of actions containing the element

Affected guards, actions, invariants are elements that contain the elements that the operator is transforming.
**Op.mergeVariable**

Applicable on models where there is at least two variables of the same type. With n variables of the same type it generates \( \binom{n}{2} \) machines.

In Mondex, machine m1 has two variables ‘abal’ and ‘lost’ with the same type.

Transfer model has two variables ‘pend’ and ‘trans’ having the same type.

Location access controller system machine doors_4 has 9 variables of the same type

<table>
<thead>
<tr>
<th>Model:</th>
<th>Mondex:m1</th>
<th>Transfer: Transfer</th>
<th>LAC:doors_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of generated models:</td>
<td>1</td>
<td>1</td>
<td>36</td>
</tr>
<tr>
<td>Number of models passed type checking:</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Generation time:</td>
<td>62ms</td>
<td>13ms</td>
<td>748 ms</td>
</tr>
<tr>
<td>Type checking time:</td>
<td>678ms</td>
<td>613ms</td>
<td>92s 811ms</td>
</tr>
<tr>
<td>Successful transformations / Total number of invariants to be transformed</td>
<td>2/2</td>
<td>5/5</td>
<td>References to the variables in predicates have been successfully substituted</td>
</tr>
<tr>
<td>Successful transformations / Total number of guards to be transformed</td>
<td>3/3</td>
<td>2/2</td>
<td></td>
</tr>
<tr>
<td>Successful transformations / Total number of actions to be transformed</td>
<td>0/7</td>
<td>0/4</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.1

In case of Mondex and the Transfer model, invariants and guards containing the variables are replaced with the new variable. The type checking fails because this operator does not support the mapping of assignments yet.

In case of the location access controller the merged variables has been successfully substituted with the new one in the predicates but not in the assignments.

**Op.abstractPartialFunctions**

This operator is built up of other operators and used as generating an abstraction of the bookstore model. It can be applied on other models using the following statement.

```
EventB0pLang Apply Op.abstractPartialFunctions On model WithActiveElements Machine("machine_name")
```
It is applicable on models defined at a too concrete level and generates an abstraction of the partial function of the machine. Although it generates more meaningful results when applied on a single machine like Transfer example and BookStore example it is also tested on models that have been developed using a refinement approach, thus already having a structured set of refinement layers. This operators in contrast with the previous one does generate an abstract layer, moves the abstraction of the partial function to the new abstract layer, and introduces the refines constructs, invariants and refines relationships.

Transfer model has two partial functions, ‘pend’ and ‘trans’

Club Management System m2_dataref machine has one: ‘attendees’

<table>
<thead>
<tr>
<th>Model:</th>
<th>BookStore</th>
<th>Transfer:Transfer</th>
<th>Club:m2_dataref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of generated models:</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Number of models passed type checking:</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Generation time:</td>
<td>469ms</td>
<td>62ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>Type checking time:</td>
<td>1s005ms</td>
<td>378ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>Successful abstractions / Total number of invariants to be abstracted</td>
<td>2/3</td>
<td>2/3</td>
<td>N/A</td>
</tr>
<tr>
<td>Successful abstraction / Total number of guards to be abstracted</td>
<td>4/4</td>
<td>0/3 Only typing guards</td>
<td>N/A</td>
</tr>
<tr>
<td>Successful abstractions / Total number of actions to be abstracted</td>
<td>4/6</td>
<td>4/4</td>
<td>N/A</td>
</tr>
</tbody>
</table>

In case of the Transfer example an abstract top layer has been successfully created, the abstracted variables have been added, and an invariant that expresses the relationship between the domain of the concrete and abstract variables have been inserted to the concrete layer. However it failed to transform most of the guards. It is due to the fact that the guards having the form of "(x→y)↦z ∈ a" that should be abstracted to "(x→y) ∈ dom_a" is not supported yet, however the implementation can be extended to map these guards as well.

In Bookstore example and Transfer example as well one invariant was not abstracted. The invariant in both cases was the one expressing that the domain of the variables equals to each other.
Concrete invariant “dom(var1) = dom(var2)” should be abstracted to “abs_var1=abs_var2” The concrete invariants have been successfully inserted saying dom(var1) = abs_var1, dom(var2)=abs_var2. The implementation would need small accommodation to successfully map this predicate as well. The generated abstractions can be found in Appendix B.

Applying this abstraction on the 3rd layer of the club management system generated 0 model. The reason can be found in the implementation of this abstraction. The initial condition Cond.hasAbstractablePartialFunctionVariable returns false, because the machine m2_dataref has a gluing invariant that states that the domain of ‘attendees’ equals to a variable present in the abstract machine m1_participants.

**Transformer.transformCollectionToSingle**

The limitation of the implementation of this transformation is that it does not support the transformation of structured refinements. This transformation provides a very basic transformation of a collection with a type “x ⇸ y” to a single entity on a single machine, such as the bookstore one however the models available has a more complex structure built around entities, for which this implementation is not sufficient enough.

### 7.2 The framework and the Operator Language

The project has been structured to two main levels. On one level models have been analysed to realize patterns. On the other level the Operator framework has been further improved to make it suitable for the implementation of abstractions.

**Operator Language**

To combine operators together in order to define more complex transformation and to achieve an extensible language required a design of a more efficient language then the available one. The following list summarizes the implemented features in the new language that were not available at the start of the project.

- Deep nesting of conditions using “or”, “and” and “not”
- Use of extensible rules, if, while, repeat, until, accumulator
- Deep nesting of combination of operators using ‘and’ and ‘parAnd’ keywords.
- Combination of operators to define new operators

The available operator language at the start of the project:

- If else rule in function call of Combinator.orCond
- “or” conditions as function Combinator.or
- Combinator.repeat and Combinator.first as rules

The new implementation makes possible, the deep nesting of model conditions, and deep nested combinations of Operator application. It is extensible, however in order to implement a specific production rule, an evaluator and a builder has to be implemented as well for the specific rule. The builder makes possible the definition of complex operators in a user friendly way. The following transformation describes a definition of an operator in the new design and the old design.

New language:

```scala
def example = (model: Model) =>
EventBOpLang If (Cond.hasAbstractablePartialFunctionVariable or
not(Cond.hasAbstractLayer))
Else Op.otherOperator
On model WithActiveElementsMachine("bookstore_0")
```

Previous language:

```scala
def example(eval: Eval) =
val res = Combinators.orCond(eval, Cond.hasAbstractablePartialFunctionVariable, Op.addAbstractLayer) parAnd(Op.abstractDomainOfPartialFunction)
```

This statement represents a simple if else condition. The writing of the nested conditions and operators shown above would require the definition of temporary function definitions. The usability
testing of the improved language was not in the scope of this project but the improvements were inevitable to achieve an extensible language, to be able to define transformation strategies and to automate transformations in the future.

**Model Representation and Query language**

The current intermediate model representation provides two interfaces to access elements: the EvenBQL query language of the Event-B Triple store and the interface of the elements. The nodes (Machine,Context,Event) store only a simple String, their name. They use the underlying query language to retrieve elements “belonging” to them.

Having the interface to retrieve elements of a node in the Event-B model such as variables of a machine, and having the query language over the store at the same time have many advantages. The query language is beneficial when certain atomic elements have to be modified across the model for example all guards, or all actions meeting certain criteria. The interface of the elements is beneficial when elements of the given node have to be retrieved for example guards of a certain event.

Having the triple representation of the model also makes easier the introduction of new constructs such as refinement, extends and sees. Creating a new machine and making other machine to refine it takes a few lines of code. Whereas in the previous algebraic model representation creating a machine meant all elements of the machine to be created are known. If not known and new events or relationships are added the whole AST had to be regenerated again. The immutable structure worsened the situation but resulted in huge memory savings that was beneficial when large amount of models were generated. Initially a Rewriter class have been implemented that served as an interface for performing changes on the AST but it have not proved to be very efficient and were discarded later on. The need for flexible establishment and modification of relationships (refines, extends, witnesses etc) in the model has resulted in the implementation of the new triple representation. It also required the integration of other parts of the framework as well. The size of
implementation of the existing operators available at the start of the project reduced from about 40 lines to 4 lines, although they have not been used in the implementation of abstractions.

**Type Checker**

As the results show in table 1 applying the type checking on the generated projects can be very slow. In case of the Location Access Controller system that contains 4 machines, the type checking of 36 projects took more than 90 seconds on an average performance desktop computer. During the type checking the Rodin platform significantly slowed down or was unavailable. The reason is behind the implementation of the current type checker in the framework. To type check a generated model it is transformed back to Rodin representation which means .bum files are created on the hard disk, then the Static Checker of Rodin is invoked that creates the .bcm files that contains the static checked information about the model. When the static checker finished Rodin raises the warnings/errors about the project if there is any. This process is a huge overhead in the performance but the warnings/errors raised may be beneficial in the future to achieve more successful mapping of formulas.
VIII. Summary and Conclusion

8.1 Summary

This report started by summarizing the advantages of automating abstractions. In the analysis part a number of atomic patterns used in the formal language of Event-B have been discussed and a possible use of them in the generation of abstractions. Their usefulness in discarding flaws has been shown and a basic transformation example have been covered that would help to focus on flaw by mutation of the model. Their application has been presented on an experimental model that was developed as a part of this project.

The implementation part focused on the evaluation and improvement of the operator language and the implementation of the abstraction discussed in the analysis part. The operator language has been improved to a more extensible language where future complex transformations can be developed using already developed atomic operators. A few atomic operators have been developed that can abstract atomic constructs for example partial functions, and are able to map a number of expressions and predicates to their abstract equivalent. These implementations are not complete, but can be extended to support the mapping of more forms of elements.

An implementation of an abstraction has been shown on the experimental example: BookStore, then the implemented abstraction has been tested on other models, where in cases it showed partially successful results. However the generation of abstractions from a concrete machine of a model that already has a structured layer of refinements seems to be harder as the concrete machines of the model refers to elements of the abstract machine.

8.2 Critique

Analysis and Implementation both were important in order to achieve any result in the project. The analysis part needs more patterns and abstractions to be defined, and the usefulness of them has to be analyzed in more details as well. Although the focus unintentionally has moved on the implementation, operators have to be extended to support mapping of more formulas.
8.3 Future Work

The framework and the operator language can provide a promising extension to Rodin Platform. First of all, more mapping of low level structures (such as relations, expressions, predicates etc) to their abstract equivalent should be implemented and the current ones need to be extended. More conditions can be implemented as well. They can be used to check if the model under development has certain properties and apply the applicable operators. Using the conditions and the low level mappings of structures, abstractions can be generated automatically and can be shown for the users of Rodin that they can apply on their model and generate an abstraction of the model automatically.

Automating the generation of abstractions may also help the refinement development approach to fit into the agile development approach where the requirements may frequently change. A change in a requirement may mean the changes in all refinement layers. By automating the generation of abstractions, the change in requirements can be addressed at the concrete level and the abstraction would be automatically generated that would help the user to get an abstract overview of their model and the provers as well.

8.4 Conclusion

Using a basic example it was shown that it is feasible to automate abstractions when it is applied on a single layer that initially has been defined at a complex level or it is applied on the top level of the refinement hierarchy. The top-down approach of the refinement methodology makes difficult the methodology of generating abstraction. In order to generate an abstraction of a model, the concrete model having abstracted, has to introduce the refine constructs (refines relationships, witnesses, gluing invariants). For these reasons automating the generation of abstractions/ refinement hierarchy is possible when the process is started on a single complex layer or on the top layer.
Appendices

Appendix A: Class Diagrams

Model – Class Diagram

Machine Elements
Event Elements

```
<<Trait>>
EventElement

Argument
+ arg : Freedefiner
+ getType : Expression
+ setType(t:Expression)
+ getTypingGuard : Predicate

Guard
+ grd : Predicate
+ isTheorem : Boolean

Action
+ act : Assignment

Witness
+ wtn : Predicate
+ source : String
```

Project Elements

```
<<Trait>>
ProjectElement

Machine
+ name : String
+ getVariables(model:Model) : List[Variable]
+ getInvariants(model:Model) : List[Invariant]
+ getVariants(model:Model) : List[Variant]
+ getEvents(model:Model) : List[Event]
+ appendVar(model:Model, nVar:Variable)
+ appendProp(model:Model, inv:Invariant)
+ preAppendProp(model:Model, inv:Invariant)
+ preAppendVar(model:Model, nVar:Variable)
+ getAbstMachine(model:Model) : List[Element]
+ getSeesContext(model:Model) : List[Element]
```

Context

```
+ name : String
+ getAxioms(model:Model) : List[Axiom]
+ getSets(model:Model) : List[CSet]
+ getConstants(model:Model) : List[Constant]
+ getExtends(model:Model) : List[Context]
```
Appendix B – Generated Abstractions/Transformations

Transfer Example

Transfer Model – Generated Abstract Layer

MACHINE
    abs_transfer

SEES
    context

VARIABLES
    pend_dom
    trans_dom

INVARIANTS
    inv11: pend_dom ∈ ℙ(ACCOUNT × ACCOUNT) not theorem
    inv12: trans_dom ∈ ℙ(ACCOUNT × ACCOUNT) not theorem

EVENTS

INITIALISATION: not extended ordinary
END

start: not extended ordinary
ANY
    a1
    a2
    m

WHERE
    type_guard0: a1 ∈ ACCOUNT not theorem
    type_guard1: a2 ∈ ACCOUNT not theorem
    type_guard2: m ∈ ℤ not theorem
    grd0: m ∈ ℕ not theorem

THEN
    act0: pend_dom := pend_dom ∪ \a1 ↦ a2\nEND

debit: not extended ordinary
ANY
    a1
    a2
    m

WHERE
    type_guard0: a1 ∈ ACCOUNT not theorem
    type_guard1: a2 ∈ ACCOUNT not theorem
    type_guard2: m ∈ ℤ not theorem
    grd0: m ∈ ℕ not theorem

THEN
    act0: trans_dom := trans_dom ∪ \a1 ↦ a2\n    act1: pend_dom := pend_dom \ \a1 ↦ a2\nEND

credit: not extended ordinary
ANY
    a1
WHERE
  type_guard0: a1 ∈ ACCOUNT not theorem
  type_guard1: a2 ∈ ACCOUNT not theorem
  type_guard2: m ∈ ℤ not theorem

THEN
  act0: trans_dom := trans_dom \ {a1 ↦ a2}

END

Transfer Model - Modified Concrete Layer - Inserted Invariants

transfer

REFINES
  abs_transfer

SEES
  context

VARIABLES
  active
  bal
  pend
  trans

INVIARNTS
  inv0: active ∈ ℙ(ACCOUNT) not theorem
  inv1: bal ∈ ACCOUNT → ℕ not theorem
  inv2: pend ∈ ACCOUNT × ACCOUNT → ℕ not theorem
  inv3: trans ∈ ACCOUNT × ACCOUNT → ℕ not theorem
  inv4: dom(pend) ∈ ACCOUNT → ACCOUNT not theorem
  inv5: active = dom(dom(pend)) \ dom(dom(trans)) not theorem
  inv6: dom(dom(trans)) \ dom(dom(pend)) = ∅ not theorem
  inv7: dom(trans) \ dom(pend) = ∅ not theorem
  inv8: dom(trans) ∈ ACCOUNT → ACCOUNT not theorem
  inv9: trans_dom = dom(trans) not theorem
  inv10: pend_dom = dom(pend) not theorem

EVENTS

INITIALISATION: not extended ordinary

THEN
  act0: active := ∅
  act1: bal := ACCOUNT × {1}
  act2: pend := ∅
  act3: trans := ∅
WHERE
  type_guard0: a1∈ACCOUNT not theorem ›
  type_guard1: a2∈ACCOUNT not theorem ›
  type_guard2: m∈Z not theorem ›
  grd0: a1∉active not theorem ›
  grd1: m∈ℕ not theorem ›
THEN
  act0: pend := pendU{a1 ↦ a2 ↦ m} ›
  act1: active := activeU{a1} ›
END

debit: not extended ordinary ›
REFINES
debit
ANY
  a1 ›
  a2 ›
  m ›
WHERE
  type_guard0: a1∈ACCOUNT not theorem ›
  type_guard1: a2∈ACCOUNT not theorem ›
  type_guard2: m∈Z not theorem ›
  grd0: a1 ↦ a2 ↦ m∈pend not theorem ›
  grd1: bal(a1)≥m not theorem ›
  grd2: m∈ℕ not theorem ›
THEN
  act0: bal := bal \{a1 ↦ bal(a1) − m}\ ›
  act1: pend := pend \ {a1 ↦ a2 ↦ m}\ ›
  act2: trans := transU{a1 ↦ a2 ↦ m}\ ›
END

credit: not extended ordinary ›
REFINES
credit
ANY
  a1 ›
  a2 ›
  m ›
WHERE
  type_guard0: a1∈ACCOUNT not theorem ›
  type_guard1: a2∈ACCOUNT not theorem ›
  type_guard2: m∈Z not theorem ›
  grd0: a1 ↦ a2 ↦ m∈trans not theorem ›
THEN
  act0: bal := bal \{a2 ↦ bal(a2)+m}\ ›
  act1: trans := trans \ {a1 ↦ a2 ↦ m}\ ›
  act2: active := active \ {a1}\ ›
END
**BookStore Example**

**BookStore Model – Generated Abstract Layer**

\[ \text{abs_bookstore}_0 \]

\[ \text{SEES} \]

\[ \text{c\_0} \]

\[ \text{VARIABLES} \]

\[ \text{basket\_dom} \]

\[ \text{budget\_dom} \]

\[ \text{INVARIANTS} \]

\[ \text{inv7: basket\_dom} \in \mathbb{P} \text{(CLIENT)} \text{ not theorem} \]

\[ \text{inv8: budget\_dom} \in \mathbb{P} \text{(CLIENT)} \text{ not theorem} \]

\[ \text{EVENTS} \]

\[ \text{INITIALISATION: not extended ordinary} \]

\[ \text{END} \]

\[ \text{Arrive: not extended ordinary} \]

\[ \text{ANY} \]

\[ \text{a} \]

\[ \text{c} \]

\[ \text{WHERE} \]

\[ \text{type\_guard0: a} \in \mathbb{Z} \text{ not theorem} \]

\[ \text{type\_guard1: c} \in \text{CLIENT not theorem} \]

\[ \text{grd0: a} \in \mathbb{N} \text{ not theorem} \]

\[ \text{grd1: c} \notin \text{basket\_dom} \text{ not theorem} \]

\[ \text{THEN} \]

\[ \text{act0: budget\_dom := budget\_domU\{c\} } \]

\[ \text{act1: basket\_dom := basket\_domU\{c\} } \]

\[ \text{END} \]

\[ \text{Put\_In\_Basket: not extended ordinary} \]

\[ \text{ANY} \]

\[ \text{b} \]

\[ \text{c} \]

\[ \text{WHERE} \]

\[ \text{type\_guard0: b} \in \text{BOOK not theorem} \]

\[ \text{type\_guard1: c} \in \text{CLIENT not theorem} \]

\[ \text{grd0: c} \in \text{basket\_dom not theorem} \]

\[ \text{END} \]

\[ \text{Pay: not extended ordinary} \]

\[ \text{ANY} \]

\[ \text{c} \]

\[ \text{WHERE} \]

\[ \text{type\_guard0: c} \in \text{CLIENT not theorem} \]

\[ \text{grd0: c} \in \text{basket\_dom not theorem} \]

\[ \text{END} \]

\[ \text{Exit: not extended ordinary} \]

\[ \text{ANY} \]

\[ \text{c} \]

\[ \text{WHERE} \]
type\_guard0\: \: \mathsf{c} \in \mathsf{CLIENT} \not\ \text{not theorem} \\
\text{grd0}\: \: \mathsf{c} \in \mathsf{basket\_dom} \not\ \text{not theorem} \\
\text{THEN} \\
\text{act0}\: \: \mathsf{budget\_dom} := \mathsf{budget\_dom} \setminus \{\mathsf{c}\} \\
\text{act1}\: \: \mathsf{basket\_dom} := \mathsf{basket\_dom} \setminus \{\mathsf{c}\} \\
\text{END} \\
\text{END} \\

\textbf{BookStore Model - Concrete Layer - Inserted Invariants} \\

\textbf{bookstore\_0} \\
\text{REFINES} \\
\text{abs\_bookstore\_0} \\
\text{SEES} \\
\mathsf{c\_0} \\
\text{VARIABLES} \\
\mathsf{basket} \\
\mathsf{budget} \\
\mathsf{price} \\
\mathsf{shelf} \\
\text{INVIARANTS} \\
\text{inv0}: \: \mathsf{basket} \in \mathsf{CLIENT} \Rightarrow \mathbb{P}(\mathsf{BOOK}) \not\ \text{not theorem} \\
\text{inv1}: \: \mathsf{budget} \in \mathsf{CLIENT} \Rightarrow \mathbb{N} \not\ \text{not theorem} \\
\text{inv2}: \: \mathsf{price} \in \mathsf{BOOK} \Rightarrow \mathbb{N} \not\ \text{not theorem} \\
\text{inv3}: \: \mathsf{shelf} \in \mathsf{BOOK} \Rightarrow \mathbb{N} \not\ \text{not theorem} \\
\text{inv4}: \: \mathsf{dom(basket)} = \mathsf{dom(budget)} \not\ \text{not theorem} \\
\text{inv5}: \: \mathsf{budget\_dom} = \mathsf{dom(budget)} \not\ \text{not theorem} \\
\text{inv6}: \: \mathsf{basket\_dom} = \mathsf{dom(basket)} \not\ \text{not theorem} \\
\text{EVENTS} \\
\text{INITIALISATION}: \: \not\ \text{extended ordinary} \\
\text{THEN} \\
\text{act0}: \: \mathsf{basket} := \emptyset \\
\text{act1}: \: \mathsf{budget} := \emptyset \\
\text{act2}: \: \mathsf{price} := \mathsf{BOOK} \times \{1\} \\
\text{act3}: \: \mathsf{shelf} := \mathsf{BOOK} \times \{2\} \\
\text{END} \\
\text{Arrive}: \: \not\ \text{extended ordinary} \\
\text{REFINES} \\
\text{Arrive} \\
\text{ANY} \\
\mathsf{a} \\
\mathsf{c} \\
\text{WHERE} \\
\text{type\_guard0}: \: \mathsf{a} \in \mathbb{Z} \not\ \text{not theorem} \\
\text{type\_guard1}: \: \mathsf{c} \in \mathsf{CLIENT} \not\ \text{not theorem} \\
\text{grd0}: \: \mathsf{c} \in \mathsf{dom(basket)} \not\ \text{not theorem} \\
\text{grd1}: \: \mathsf{a} \in \mathbb{N} \not\ \text{not theorem} \\
\text{THEN} \\
\text{act0}: \: \mathsf{basket} := \mathsf{basket} \cup \{\mathsf{c} \mapsto \emptyset\} \\
\text{act1}: \: \mathsf{budget} := \mathsf{budget} \cup \{\mathsf{c} \mapsto \mathsf{a}\} \\
\text{- 69 -}
ENDD

Put_In_Basket: not extended ordinary  
REFINES
   Put_In_Basket
ANY
    b
    c
WHERE
    type_guard0: b ∈ BOOK not theorem
    type_guard1: c ∈ CLIENT not theorem
    grd0: c ∈ dom(basket) not theorem
    grd1: b ∈ dom(price) not theorem
    grd2: shelf(b) > 0 not theorem
    grd3: budget(c) − price(b) ≥ 0 not theorem
THEN
   act0: shelf := shelf ovr {b ↦ shelf(b) − 1}
   act1: basket := basket ovr {c ↦ basket(c) ∪ {b}}
   act2: budget := budget ovr {c ↦ budget(c) − price(b)}
END

Pay:  not extended ordinary  
REFINES
   Pay
ANY
    c
WHERE
    type_guard0: c ∈ CLIENT not theorem
    grd0: c ∈ dom(basket) not theorem
THEN
   act0: basket := basket ovr {c ↦ ∅}
END

Exit: not extended ordinary  
REFINES
   Exit
ANY
    c
WHERE
    type_guard0: c ∈ CLIENT not theorem
    grd0: c ∈ dom(basket) not theorem
THEN
   act0: shelf := shelf ovr (λbook · book ∈ basket(c) | shelf(book) + 1)
   act1: basket := {c} ∪ {basket}
   act2: budget := {c} ∪ {budget}
END
END
BookStore Model – Generated Transformation To Single Client

bookstore_0

SEES
c_0

VARIABLES
price
shelf
basketDom
basket
budgetDom
budget

INVARINTS
inv0: \text{price} \in \text{BOOK} \rightarrow \mathbb{N} \text{ not theorem}
inv1: \text{shelf} \in \text{BOOK} \rightarrow \mathbb{N} \text{ not theorem}
inv2: \text{basketDom} \in \text{BOOL} \text{ not theorem}
inv3: \text{basket} \in \mathbb{P}(\text{BOOK}) \text{ not theorem}
inv4: \text{budgetDom} \in \text{BOOL} \text{ not theorem}
inv5: \text{budget} \in \mathbb{N} \text{ not theorem}
inv6: \text{basketDom} = \text{budgetDom} \text{ not theorem}

EVENTS

INITIALISATION: \text{not extended ordinary}
THEN
act0: \text{price} \leftarrow \text{BOOK} \times \{1\}
act1: \text{shelf} \leftarrow \text{BOOK} \times \{2\}

END

Arrive: \text{not extended ordinary}
ANY

a
WHERE
type_guard0: a \in \mathbb{Z} \text{ not theorem}
grd0: \text{basketDom} \neq \text{TRUE} \text{ not theorem}
grd1: a \in \mathbb{N} \text{ not theorem}
THEN
act0: \text{basket} \leftarrow \text{basketU} \emptyset
act1: \text{basketDom} \leftarrow \text{TRUE}
act2: \text{budget} \leftarrow a
act3: \text{budgetDom} \leftarrow \text{TRUE}

END

Put_In_Basket: \text{not extended ordinary}
ANY

b
WHERE
type_guard0: b \in \text{BOOK} \text{ not theorem}
grd0: \text{basketDom} = \text{TRUE} \text{ not theorem}
grd1: b \in \text{dom} (\text{price}) \text{ not theorem}
grd2: \text{shelf}(b) > 0 \text{ not theorem}
X grd3: \text{budget}(c) - \text{price}(b) \geq 0 \text{ not theorem}
THEN
act0: \text{shelf} \leftarrow \text{shelfovr}(b \mapsto \text{shelf}(b) - 1)
X act1: basket := basket(c) ∪ {b} ›
act2:  basket_dom := TRUE ›
X act3: budget := budget(c) − price(b) ›
act4:  budget_dom := TRUE ›
END

Pay: not extended ordinary ›
ANY
 c ›
WHERE
 type_guard0: c∈CLIENT not theorem ›
 grd0:  basket_dom=TRUE not theorem ›
THEN
 act0:  basket := Ø ›
 act1:  basket_dom := TRUE ›
END

Exit: not extended ordinary ›
ANY
 c ›
WHERE
 type_guard0: c∈CLIENT not theorem ›
 grd0:  basket_dom=TRUE not theorem ›
THEN
 X act0: shelf := shelf over (λbook·book∈basket(c) | shelf(book)+1) ›
 act1:  basket_dom := FALSE ›
 act2:  basket := Ø ›
 act3:  budget_dom := FALSE ›
 act4:  budget := 0 ›
END

X-failed to transform

END
References


Sutcliffe, G. Automated Theorem Proving [Online]. Available at: [Accessed: 15 November].
