UNIVERSITY OF SOUTHAMPTON

Methodology of Refinement and Decomposition in UML-B

by

Mar Yah Said

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering and Applied Science

Department of Electronics and Computer Science

September 2010

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Mar Yah Said

UML-B is a UML-like graphical front end for Event-B that provides support for object-oriented modelling concepts. In particular, UML-B supports class diagrams and state machines, concepts that are not explicitly supported in plain Event-B. In Event-B, refinement is used to relate system models at different abstraction levels. The same abstraction-refinement concepts can also be applied in UML-B. This work introduces the notions of refined classes, refined state machines and extended classtypes to enable refinement of classes and state machines in UML-B. This work makes explicit the structures of class and state machine refinement in UML-B. This work also introduces seven refinement techniques which are, adding new attributes and associations, adding new classes, elaborating state, elaborating transition, moving a class event (or a state machine transition), adding new attributes and associations, and adding new classtypes.

In Event-B, decomposition is used to decompose a system into components. The same decomposition concepts can be applied in UML-B. This work introduces the techniques of flattening state machines and state grouping to facilitate a decomposition of a UML-B machine. This work also introduces the notion of composed machine which composes the component machines. The composed machine refines a machine which is being decomposed. The composed machine is used to ensure the composition of the component machines is a valid refinement. Together with the composed UML-B machine, the notions of included machine, composed event and constituent event are introduced.

The UML-B drawing tool and Event-B translator are extended to support the new refinement and decomposition concepts. A case study of an auto teller machine (ATM) is presented to validate the extensions of UML-B with regards to the above notions. The ATM case study also demonstrates the above techniques introduced in refinement and decomposition. In addition, this work provides guidelines for performing refinement and decomposition in UML-B and presents a number of generic invariants that may be used when refining a middleware. The middleware is a component via which a requesting component such as an ATM and a responding component such as bank interact in a distributed system.

Contents

Acknowledgements **xi**

1 Overview of Research Project **1**
1.1 Introduction . . . 1
1.2 Problem Statement . . . 1
1.3 Objective . . . 2
1.4 Methodology . . . 2
1.5 Tools . . . 3
1.6 Research Questions . . . 3
1.7 Summary of Contributions . . . 4
1.8 Thesis Structure . . . 5

2 Literature Review **6**
2.1 Introduction . . . 6
2.2 Formal Methods . . . 6
2.3 B-Method . . . 7
2.4 Event-B . . . 9
2.5 Refinement . . . 9
2.5.1 Refinement and Proof Obligations in Event-B . . . 11
2.5.2 Records Extension in Event-B . . . 12
2.6 Decomposition in Event-B . . . 14
2.7 Rodin Event-B Tool . . . 15
2.8 Unified Modelling Language . . . 16
2.8.1 Diagrams in the UML . . . 16
2.8.2 Classes . . . 17
2.8.3 State Machines . . . 18
2.9 Model Driven Development Architecture (MDA) . . . 18
2.10 UML-B . . . 20
2.10.1 UML-B Modelling Environment . . . 21
2.10.2 UML-B Metamodel . . . 26
2.11 Other Modelling Languages . . . 27
2.11.1 Z Specification Language . . . 28
2.11.2 Vienna Development Method (VDM) . . . 29
2.11.3 Comparison between B, Z and VDM . . . 31
2.12 Programming Specification Languages . . . 32
2.12.1 Larch . . . 32
2.12.2 Java Modelling Language (JML) . . . 33

2.13 Other Work on the Integration of Formal Methods with UML 35
2.14 Summary 37

3 An ATM Case Study in Event-B 38
3.1 Introduction 38
3.2 Case Study: Auto Teller Machine System 38
3.2.1 Description of the ATM system 38
3.2.2 Summary of the ATM Requirements for cash withdrawal 38
3.3 Event-B Model for ATM 39
3.3.1 ATM Abstract Machine 40
3.3.2 First Refinement 40
3.3.3 Second Refinement 42
3.3.4 Third Refinement 43
3.3.5 Fourth Refinement 44
3.3.6 Fifth Refinement 45
3.3.7 Sixth Refinement 46
3.3.8 Experience modelling the ATM system in Event-B 46
3.4 Event-B and UML-B Refinement 48
3.5 Summary 49

4 Refinements in UML-B 50
4.1 Introduction 50
4.2 Limitations of the previous UML-B 51
4.2.1 Limitations in Refining Classes 51
4.2.2 Limitations in Refining State Machines 51
4.2.3 Limitations in Extending Classtypes 54
4.3 Refinement of Classes in UML-B 55
4.4 Refinement of State Machines in UML-B 58
4.5 Event/Transition Movement 61
4.6 Extension of Classtypes in UML-B 65
4.7 Summary 65

5 UML-B Metamodel Extension 67
5.1 Introduction 67
5.2 Extending the UML-B Metamodel and Tool to Support Class Refinement 67
5.3 Extending the UML-B Metamodel and Tool to Support State Machine Refinement 70
5.4 Extending the UML-B Metamodel and Tool to Support Classtype Extension 73
5.5 Extending the UML-B Drawing Tools 74
5.6 Summary 76

6 Modelling The ATM Case Study in UML-B 78
6.1 Introduction 78
6.2 ATM Case Study: An overview 78
6.3 Abstract Machine 79
6.4 First Refinement 80
6.5 Second Refinement 83
6.6 Third Refinement 85

6.7 Fourth Refinement . . . 90
6.8 Experiences from the ATM system in UML-B . . . 95
6.9 Summary . . . 96

7 Decomposition and Composition in UML-B 97
7.1 Introduction . . . 97
7.2 Flattening State Machines . . . 99
7.2.1 Formal Definition of Flattening . . . 101
7.3 State Grouping . . . 107
7.3.1 Formal Definition of Grouping . . . 108
7.4 Composed Machines in Event-B . . . 113
7.5 Composed Machine, Included Machine, Composed Event and Constituent Event . . . 114
7.6 Extending the UML-B Metamodel and Tool to Support Composition . . . 116
7.7 Extending the UML-B Drawing Tool . . . 120
7.8 Summary . . . 121

8 Modelling The ATM Case Study in UML-B ((De)composition) 123
8.1 Introduction . . . 123
8.2 ATM Case Study: An overview . . . 123
8.3 Fifth Refinement . . . 125
8.4 Sixth Refinement . . . 126
8.5 Seventh Refinement . . . 127
8.6 Decomposition of the Seventh Refinement and Composition (Eight Refinement) . . . 134
8.6.1 ATM Component: Machine *mATM* . . . 136
8.6.2 Bank component: Machine *mBank* . . . 137
8.6.3 Middleware component: Machine *mMW* . . . 138
8.6.4 Shared Transitions . . . 138
8.6.5 Eighth Refinement: Composed Machine . . . 140
8.7 Refinement of The Middleware Component . . . 141
8.7.1 Context *ATM_CXR3* . . . 141
8.7.2 First Refinement of Middleware: Machine *mMW_R1* . . . 142
8.7.3 Second Refinement of Middleware: Machine *mMW_R2* . . . 145
8.8 Statistics of Proof Obligations . . . 146
8.9 Guidelines for Refinement and Decomposition . . . 146
8.10 Patterns for Invariants when Introducing Messages in the Refinements of the Middleware . . . 149
8.11 Summary . . . 152

9 Conclusion 153
9.1 Contributions . . . 153
9.2 Limitations . . . 154
9.3 Comparison to Other Work on Translating to B . . . 154
9.4 Comparison to Other Work on the Class and State Machine Refinements 156
9.5 Comparison of UML-B to Plain Event-B . . . 158
9.6 Comparison of UML-B to Goal Diagram and Event Refinement Diagram . 159
9.7 Future Work . . . 162

A Requirement Document of the ATM System for Event-B Development **164**

B Requirement Document of the ATM System for UML-B Development **167**

C ATM Case Study: Using Rodin Event-B **172**

D ATM Case Study: Using Rodin UML-B **214**

Bibliography **373**

List of Figures

2.1 A Structure of B Method Abstract Machine . . . 8
2.2 The *Team* Machine: Abstract Specification . . . 10
2.3 The *TeamR* Machine: Concrete Specification . . . 11
2.4 Syntactic Sugar for Record Types . . . 12
2.5 Event-B Context for Record Types . . . 12
2.6 Abstract Context . . . 13
2.7 Refined Context . . . 14
2.8 Illustration of the Event-based decomposition . . . 15
2.9 Illustration of the State-based Decomposition . . . 15
2.10 OMG's Model Driven Architecture . . . 19
2.11 Controllable Incremental Refinement . . . 20
2.12 Example of Package Diagram . . . 22
2.13 Examples of Context Diagram and Event-B Translation . . . 23
2.14 Subtyping a Classtype in UML-B . . . 23
2.15 Example of Class Diagram . . . 24
2.16 Example of a State Machine Diagram . . . 24
2.17 Properties of the *withdraw* Transition . . . 24
2.18 Generated Event-B for the Events Using State Function Representation . . . 25
2.19 Generated Event-B for the *withdraw* Event Using State Sets Representation 26
2.20 UML-B Metamodel (part of) . . . 27
2.21 Schema Definition . . . 28
2.22 A Z Specification of the Team Schema . . . 29
2.23 VDM Module . . . 30
2.24 A VDM Specification of the Team Module . . . 31
2.25 LCL Interface Specification . . . 32
2.26 A JML Specification of the Interface *Gendered* . . . 33
2.27 A JML Specification of the Class *Animal* (part of) . . . 34
2.28 A JML Specification of the Class *Patient* (part of) . . . 34

3.1 ATM Context . . . 40
3.2 ATM Machine in Event-B . . . 41
3.3 ATM System State . . . 42
3.4 Sub-states of the States *transactionOption* and *performWithdrawal* . . . 43
3.5 Distributed States . . . 44

4.1 Package Diagram, Class Diagram and Generated Implicit Contexts . . . 52
4.2 Generated Event-B Machines . . . 53
4.3 Example of State Machine Refinement . . . 53

4.4 Generated Event-B Machine from UML-B State Machines 54
4.5 Package Diagram, Context Diagram and Generated Contexts 55
4.6 Package Diagram and UML-B Specification of Machine M1 57
4.7 Package Diagram and Class Diagram of Machine M2 58
4.8 Refinement of State Machine (Machine M2 refines Machine M1) 59
4.9 An Alternative of Modelling a Nested State Machine 62
4.10 Example of the UML-B Specification with an Event in a Class CA of an Abstract Machine 62
4.11 Example of the UML-B Refinement for the First Method 63
4.12 Example of the UML-B Refinement for the Second Method 64
4.13 Package Diagram and Context Diagram 66

5.1 UML-B Metamodel Extensions for Classes 68
5.2 UML-B Metamodel Extensions for Attributes 69
5.3 UML-B Metamodel Extensions for Name Elements 70
5.4 Changes to UML-B Metamodel for State Machine 71
5.5 Changes to UML-B Metamodel for State 72
5.6 Changes to UML-B Metamodel for Transition 73
5.7 UML-B Metamodel Extensions for Classtypes 74
5.8 Drawing Tool Extensions for a Class Diagram Editor 75
5.9 Drawing Tool Extensions for a State Machine Diagram Editor 76

6.1 ATM Package Diagram 79
6.2 UML-B Specification of ATM Abstract Machine 80
6.3 UML-B Specification of ATM First Refinement 82
6.4 UML-B Specification of ATM Second Refinement 84
6.5 UML-B Specification of ATM Third Refinement 86
6.6 Class Diagram of the Fourth Refinement 90
6.7 UML-B Specification of the Fourth Refinement: Request Event 91
6.8 UML-B Specification of the Fourth Refinement: Response Event 94

7.1 Decomposition of State Machines 98
7.2 Example of State Machine Refinement (Nested State Machine) 99
7.3 Example of Refinement by Flattening and Grouping State Machines . . . 101
7.4 A Structure of Event-B Composed Machine 113
7.5 Partitioning of the State Machines for Decomposition 115
7.6 UML-B Model of the Composed Machine 116
7.7 State Machine of the Client Component 117
7.8 State Machine of the Server Component 117
7.9 State Machines of the Middleware Component 117
7.10 Generated Event-B Machine for Composed Event 118
7.11 UML-B Metamodel Extensions for Composed Machine 118
7.12 UML-B Metamodel Extensions for Included Machine 119
7.13 UML-B Metamodel Extensions for Composed Event 120
7.14 UML-B Metamodel Extensions for Combined Event 120
7.15 Drawing Tool Extensions for a Package Diagram Editor 121

8.1 ATM Package Diagram 124

8.2 Flattening the State Machine ATM_SM in the Fifth Refinement 126
8.3 The Refined State Machine ATM_SM of the Sixth Refinement 127
8.4 The State Machine *waitingResponseSM* of the Sixth Refinement 128
8.5 Classes of the Seventh Refinement 129
8.6 Partitioning of the Nested State Machine 134
8.7 Architectural Illustration of Decomposition 135
8.8 UML-B Machine for ATM Component 136
8.9 UML-B Machine for Bank Component 137
8.10 UML-B Machine for Middleware Component 138
8.11 UML-B Composed Machine ATM_R8 140
8.12 UML-B Context with Message Types and Properties 142
8.13 UML-B Machine for the First Refinement of the Middleware 143
8.14 Class Diagram of the Second Refinement for the Middleware Component . 145
8.15 Example of a Bad State Machine 147
8.16 Example of a Good State Machine 148

9.1 Example of Goal Diagram 161
9.2 Example of Event Refinement Diagram 161
9.3 State Machine of ATM in UML-B 162

List of Tables

3.1 Statistics from the Proof Effort 48

6.1 Statistics from the Proof Effort 95

8.1 Statistics from the Proof Effort 146

Academic Thesis: Declaration Of Authorship

I, Mar Yah Said, declare that the thesis entitled Methodology of Refinement and Decomposition in UML-B and the work presented in it are my own. I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;

2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;

3. Where I have consulted the published work of others, this is always clearly attributed;

4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;

5. I have acknowledged all main sources of help;

6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;

7. Parts of this work have been published as [83].

Signed: .....

Date: .....

Acknowledgements

Many thanks to Prof. Michael Butler for being a great and helpful supervisor. My thanks also goes to Dr. Colin Snook who helps me with the UML-B and Eclipse EMF and GMF. Also, thanks to Vitaly Savicks who help me with the GMF. I also would like to express my gratitude to my friends Kriangsak, Renato, Nurlida and Tossaporn for their help and support. Finally I would like to thanks all staff and colleagues in the DSSE research group for their support.

To my dearest husband, Azmi for your love, support and patient has given me the strength.

To my respectful dad and moms for their love and care has given me the courage.

To my supportive sisters and brothers for your love and helps has given me the patient.

To my precious sons, Anwar and Adam, for your laughter has brighten my day.

Chapter 1

Overview of Research Project

1.1 Introduction

This chapter gives an overview of this thesis. The remainder of this chapter includes sections on problem statement, objectives, methodology, tools, research questions, the summary contributions and thesis structure.

1.2 Problem Statement

UML-B [96, 95, 97] is a graphical formal modelling notation that has some resemblance with UML [2, 39] and is based on Event-B [69]. UML-B is defined precisely by its metamodel [96] which is an abstract syntax of the structure of the UML-B language. The UML-B notation is supported by the UML-B tool which is a plug-in extension feature to the Rodin Event-B verification tool [4, 22]. The UML-B tool (U2B translator [94]) generates Event-B models corresponding to a UML-B development and the Rodin tool is then used to discharge proof obligations associated with the generated Event-B models. The Rodin Event-B tool supports Event-B which is a new variant of the B method.

The B method [5, 84] is based on stepwise refinements and decomposition of a problem. An abstraction is made to capture the most essential properties of a system after initial informal specification of requirements. This abstract specification is made more concrete and detailed in steps at each refinement or decomposition step, proof obligations are generated and must be discharged in order to prove that the outputs of the step are a valid refinement of the previous level. At each step when more detailed requirements are introduced or implementation steps are taken, it is proved that they respect all the previous levels. This method ensures that the developed program obeys the properties expressed in all the levels of specification from which it is derived. Such proof is not always easily achieved. The form and style of the formal B specification can greatly affect

the ease of achieving these proof obligations. Hence a primary criterion for developing specifications in B is ease of proof. This is why refinement and decomposition are significant mechanisms in building a B specification.

Since the B-method and Event-B method are reflected in UML-B, the abstraction-refinement concepts must also be catered for in UML-B. However, an effective style of performing refinement in UML-B had not yet being determined when starting our research. Also, the UML-B tool still has limitations with modelling refinement. Likewise, UML-B does not support decomposition.

The purpose of the research is to study and propose an effective methodology of refinement and decomposition in UML-B. We believe that the result of our research will be a methodology of refinement and decomposition in UML-B which will assists modelling in UML-B.

1.3 Objective

The objectives of this research are:

- To extend UML-B to support refinement and decomposition.
- To produce a set of refinement and decomposition styles for UML-B models.
- To test the effectiveness of the UML-B extensions and, UML-B refinement and decomposition styles through experiments with case studies.

1.4 Methodology

Methodology is the methods, procedures and techniques used to collect and analyse information in a research process. The methodology for this research is listed as follows:

- Case studies by using the Rodin Event-B tool and UML-B tool.
- Create a proposal for extending the UML-B metamodel to support refinement and decomposition in UML-B based on experiences with case studies.
- Extend the UML-B tool by implementing the new (extension) features proposed in the UML-B metamodel.
- Evaluate the existing and extended features of the UML-B tool through the case studies.

All the four activities listed above were performed in an iterative manner.

1.5 Tools

Several tools or application software are used during the duration of this research and they are listed as follows:

- ProB
  ProB is a model checker and animation tool for the B method. It is used to gain understanding and feedback in modelling the B method.
- Rodin Event-B
  The Rodin Event-B tool is used in modelling the ATM case study to gain experience in modelling and performing refinement in plain Event-B.
- Rodin UML-B
  Rodin UML-B is a tool which integrates UML and B. It is used to gain understanding and experience in modelling and performing refinement in UML-B. This tool was extended in this study to support modelling refinement and decomposition in UML-B.
- Eclipse
  Eclipse is used as a Java Integrated Development Environment (IDE) in implementing the extensions to the UML-B tool.
- IBM Rational Software Architect (RSA)
  Rational Software Architect provides integrated design and development support for model-driven development with the Unified Modelling Language (UML). RSA is used as an editing tool in order to make the UML-B metamodel more presentable in this report.
- Java
  Java is the programming language used to implement the U2B translator.

1.6 Research Questions

- What are the extensions that can be made to the UML-B language and tool to support refinement and decomposition?
- What are ways or styles for performing refinement and decomposition in UML-B?
- How effective are the language and tool extensions?

1.7 Summary of Contributions

In this research study, we intended to provide a way of performing refinement in UML-B. In Chapter 4 we introduce a notion of refined class, a notion of refined state machine and a notion of extended classtype which are the intended contributions. Several refinement techniques are introduced in conjunction with the notions. The techniques are listed as follows:

- adding new attributes and associations to refined classes
- adding new classes in a refinement
- elaborating refined states into sub-states
- elaborating transitions
- moving events or transitions to refined classes or new classes in a refinement
- adding new attributes and associations to extended classtypes
- adding new classtypes in a refinement

We also intended to provide a way of performing decomposition in UML-B. In Chapter 7 we introduced a notion of composed machine, included machine, composed event and constituent event. These notions allow a modeller to compose a number of decomposed machines. The composition is done to ensure that the decomposed machines are valid refinements. We also introduce two techniques involving structuring the hierarchy of state machines which are flattening state machines and state grouping.

The contributions of this work involved the extensions made to the UML-B metamodel and tool to support refinement and decomposition in UML-B.

Further contribution of this work is a development of the case study in UML-B. The case study is done to validate the extensions to the UML-B. The case study also demonstrates the use of the above mentioned techniques.

Other contributions are the guidelines for performing refinement and decomposition in UML-B and the generic invariants in the refinement of a middleware component are discussed.

1.8 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2 describes a background study on related domain knowledge including the foundation of formal methods, B method, Event-B, refinement, decomposition, Rodin tools, UML, MDA, UML-B, other formal methods and related work.

Chapter 3 describes a case study of an auto teller machine (ATM) developed using Rodin Event-B. There are seven machine levels of the development. At the end of the chapter, a section on the experience of the development is included.

Chapter 4 introduces the notion of refined classes, refined state machines and extended classtypes. The chapter discussed a section on the limitations of UML-B. Then, the notions of refined classes, refined state machines and extended classtypes are described with simple examples. The refinement techniques listed in Section 1.7 are described.

Chapter 5 describes the extensions to the UML-B metamodel and the extensions to the UML-B drawing tools to support the introduced notions in Chapter 4.

Chapter 6 describes the development of the ATM case study using UML-B. The case study applies and validates the notions of refined classes, refined state machines, extended classtypes and the techniques which are mentioned in Chapter 4.

Chapter 7 introduces the techniques of flattening state machines and state grouping in UML-B. These techniques may be performed preceding decomposition of a refinement machine. This chapter also introduces the notions of composed machine, included machine, composed event and constituent event. These notions are described with a number of simple examples. This chapter also describes the extensions to the UML-B metamodel and the extensions to the UML-B drawing tool to support these notions.

Chapter 8 is a continuation of the development of the ATM case study in UML-B. The case study applies and validates the techniques and the notions introduced in Chapter 7. The guidelines for performing refinement and decomposition in UML-B are put forward. This chapter also put forward a set of patterns for constructing invariants when refining a middleware component.

Chapter 9 concludes the thesis. The contributions of the thesis and limitations of the work are summarized. Comparisons of UML-B with other work on the integration of UML and B are put forward. Comparison of this work with other work on the class and state machine refinements are provided. Comparisons of modelling in Event-B and UML-B are discussed. Comparisons of the goal diagram and event refinement diagram with this work are presented. This chapter also outlines some future work.

Chapter 2

Literature Review

2.1 Introduction

This chapter presents a description of the related background studies of the research topic. The topics include formal methods, B, Event-B, refinement, the Rodin Event-B tool, UML-B, Unified Modelling Language (UML) and Model Driven Architecture (MDA). This chapter also contains topics about other formal specification languages like Z [98], VDM [52], Larch [43] and JML [42] and related work.

A background study of formal method is included in this chapter because the research is concerned with UML-B that integrates semi-formal and formal modelling concepts. The B and Event-B methods are included because the research is concerned with them. Clearly, topics on refinement and Unified Modelling Language (UML) are included because the research is directly related to them. A section on the Rodin Event-B tool is included because it is the main tool used during the research. Model Driven Architecture (MDA) [70] is concerned with modelling and model transformation. MDA is relevant to the research because model transformation is related to refinement. Z and VDM are similar to the B method which means they merit investigation. Whereas Larch and JML are different from B method, they are also included to gain wider knowledge of different kinds of formal methods. JML utilizes the object-oriented principles and this is another reason it is included in this chapter as the UML-B also utilizes the object-oriented principles.

2.2 Formal Methods

Formal methods [68, 45, 102] are methods that based on mathematical notation for specifying, developing and verifying software and hardware systems. Formal methods is a sub field of software engineering. Formal methods are used to uncover ambiguity,

incompleteness and inconsistency in a system. Using formal methods early in the system development process, can help reveal design faults which would otherwise perhaps be discovered during costly testing and debugging phases.

A formal specification is an actual outcome of applying a formal method. A formal specification is a description of software and hardware using mathematical notation that may be used to develop an implementation. It express what the system should do. The advantage of formal specification is that it can be verified using formal verification techniques to show that a candidate design is correct with respect to the specification. Formal specifications are expressed in languages with formally defined syntax and semantics. Examples of formal specification languages are Z, Vienna Development Method (VDM), Larch, JML and the B-method.

Formal specification languages can be categorised into two kinds which are model-oriented and property-oriented. For model-oriented methods, a system's behaviours are defined directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions and sets. For property-oriented methods, a system's behaviours are defined indirectly by stating a set of properties, usually in the form of a set of axioms that the system must satisfy. Z, VDM and B are model-oriented methods for specifying the behaviours of sequential programs and abstract data types. Larch and JML are property-oriented methods.

Much more specification, design and documentation is written using informal methods like natural language and diagrams compared to formal methods. This maybe because informal methods are easier to comprehend and learn whereas the formal notations are difficult to understand. However, informal methods tend to be ambiguous compared to formal methods. Specifications written using formal methods are precise and unambiguous. The critics believe that formal methods are difficult. However, formal method's supporters believe that formal methods can revolutionise development. Nevertheless, there is not much published evidence to support either side [45].

2.3 B-Method

B is a method for the specification, design and implementation of software systems and is originated by Abrial [5]. The B method supports formally verified software development from a specification through refinements which lead to an implementation.

The B method is based on the Abstract Machine Notation (AMN) and set theory. Set notation is important in B in which it is used for data modelling. State modifications are described using generalised substitutions. The B method uses the set theoretic constructs such as sets and relations in expressing specifications. Functions which are a particular kind of relations are used extensively in the B method.

The B method emphasises simplicity. Complicated programming constructs are purposely excluded so the designers are forced to use clear and well-understood statements. The structuring mechanisms provided by the B-Method are also characterised by simplicity and are designed particularly with verification in mind. A basic building block of a specification is the abstract machine. Shown in Figure 2.1 is the structure of an abstract machine written in the B method.

MACHINE
SETS
CONSTANT
PROPERTIES
VARIABLES xx
INVARIANT $xx \subseteq \mathbb{N}$
INITIALISATION xx:=0
OPERATIONS
END

FIGURE 2.1: A Structure of B Method Abstract Machine

A description of clauses in Figure 2.1 is as follows:

- The MACHINE clause introduces the name of the machine.
- The SETS clause introduces the sets which are used in the machine.
- The CONSTANT clause introduces the constants which are used in the machine.
- The PROPERTIES clause contains the definition of constants.
- The VARIABLES clause introduces the variables of the state of the machine. In the example there is only one variable, xx.
- The INVARIANT clause introduces the invariant property of the state of the machine. The invariants consist of predicates separated by the conjunction operator. The example constrained the variable xx to be any natural number.
- The INITIALISATION clause defines the initial state of the machine. For example, the state variable xx is initially 0.
- The OPERATIONS clause describes the operation that can cause state changes in the machine.

The B method was used successfully to develop the software for several industrial applications [8], such as, Line 14 of Paris Subway [79] and Shuttle at Roissy airport [14].

2.4 Event-B

Event-B [6] is a new variant of the B method and is based on Action Systems [12]. An action system is a collection of actions on some set of state variables. An action system describes the state space (the set of possible assignment values to the state variables) of a system and the possible actions that can be executed in the system. Event-B aims to overcome the limitation of the B method in modelling a system that consists of both software and hardware components. A limitation of the B method is that at the highest level of abstraction, the (component) interface is fixed, that means it cannot be refined later [96]. When modelling systems, an observation of a closed system with no inputs and outputs is made. This closed system has state that is altered by spontaneous occurrence of events. As the state is refined to have more detailed state, correspondingly more events are observed occurring during that state.

In contrast to classical B, Event-B distinguishes between contexts and machines [4, 34]. A context contains definitions and properties of types and constants. A machine contains state variables, invariants and events that update the variables. A machine may see several contexts. Operations are called events in Event-B. Each event has *guards* and *actions*. The *guards* are a predicates formed on constants or variables which represent the necessary conditions for the event to happen. The *actions* define the state variable changes that happen as the event occurs.

2.5 Refinement

Refinement [5, 7, 72] is a technique which is used to alter the abstract model of a software system, i.e., its specification, into another mathematical model that is more concrete, i.e., its refinement, while maintaining the same abstract properties. Refinement is an important technique for managing the complexity of a system being developed. It is important to have an abstract model of a system so that the core functions of a system can be focused on. Further refinements of the abstract model allows the specifier to focus on different aspect of the system at different refinement levels.

Two main styles have been proposed for refinement which are "posit-and-prove" and "transformational" [27, 60]. Posit-and-prove is where a refinement of a specification is proposed and then justified against its abstract specification via the verification of a set of proof obligations. That is, in order to verify model *M1* is refined by model *M2*, a tool is used which generates proof obligations from both *M1* and *M2* that can be verified using theorem provers or possibly checked using model checkers. On the other hand, transformational refinement is where algorithms or rules are applied to a specification to generate a more concrete specification. That is, a transformation is applied to all or part of *M1* that automatically constructs *M2* in a way that guarantees refinement.

MACHINE *Team*
SETS $ANSWER$={*in*,*out*}
VARIABLES *team*
INVARIANT *team* ⊆1..22 ∧ card(*team*)=11
INITIALISATION *team*:=1..11
OPERATIONS
aa ⟵ *query*(*pp*) ≙
PRE *pp* ∈ 1..22
THEN
IF *pp* ∈ *team*
THEN *aa*:=*in*
ELSE *aa*:=*out*
END
END
END

FIGURE 2.2: The *Team* Machine: Abstract Specification

Transformation might result in occurrence of side conditions which need to be verified but discharging them should be less effort than proving *M1* is refined by *M2* in the posit-and-prove style.

There are two ways a refinement model can be more concrete. First, it can contain more requirements by adding variables and events. Second, it can be closer to an implementation by replacing abstract variables by concrete variables. The former is called *superposition* refinement and the later is called *data-refinement* refinement.

An example of data refinement is illustrated in Figure 2.2 from the B method book of Schneider [84]. The figure shows a specification of a football team where the machine *Team* maintains the set of players during a football game. There are 22 players, numbered from 1 to 22 (represents by *team* ⊆1..22) and 11 players will be formed in the team from these (represents by card(*team*)=11). These properties are stated in the INVARIANT clause. The team is initialised as the first 11 players. The machine consists an operation *query* that tells whether a particular player (*pp*) is in the team. The output (*aa*) of query operation is from the enumerated set *ANSWER*.

The machine *Team* can be refined by the refinement machine *TeamR* in Figure 2.3. The state variable *teamr* refines its abstract variable *team* and it is a function from an array of size 11 to the set of players 1..22. The ⤔ represents an injective function that means all the players in the array are different players. The *gluing invariant*, *team* = *ran*(*teamr*) represents the members of the team by the array's elements. The INITIALISATION clause initialises the initial state in which *teamr* is assigned the function in which the players appear in the array in numerical order. The condition of the operation *query* is evaluated on *teamr*.

The refinement process often can reduce nondeterminism of the abstract specification, replace abstract mathematical data structures by data structures implementable on a

REFINEMENT $TeamR$
REFINES $Team$
VARIABLES $teamr$
INVARIANT $teamr \in 1..11 \rightarrowtail 1..22 \wedge team = ran(teamr)$
INITIALISATION $teamr := \bigtriangledown nn.(nn \in 1..11 \mid nn)$
OPERATIONS
$aa \longleftarrow query(pp) \triangleq$
IF $pp \in ran(teamr)$
THEN $aa := in$
ELSE $aa := out$
END
END

FIGURE 2.3: The *TeamR* Machine: Concrete Specification

computer, and eventually, gradually introduce implementation decisions.

2.5.1 Refinement and Proof Obligations in Event-B

A development in Event-B is done through refinement. A development in Event-B is accompanied by a number of proof obligations that justify its correctness. At the most abstract model, it is required to form an *invariant* that defines the static properties of the data being modelled. This will create a number of proof obligations relating to the model consistency in order to ensure that the invariant is preserved in all the events of the model. Each refinement in the Event-B development will add further invariants relating the abstract model and the refined model.

Refinement in Event-B is done by refining both its state and its events. This is essentially done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variable, x, and the concrete state variable, y, are linked together by a predicate called a *gluing invariant* J(x, y). The gluing invariant is used to create a number of refinement proof obligations which are needed to ensure that each abstract event is correctly refined by its concrete version, each new event refines *skip*, no new event take control for ever (live-lock free) and there is always at least one enabled event (no dead-lock) [28, 47, 100]. There are three status of events namely *convergent*, *anticipated* and *ordinary*. The new events which satisfy live-lock freeness are *convergent*. In cases where the convergence of some events can only be shown in later refinement, their convergence is *anticipated*. *Ordinary* events mean the events are not *convergent*.

The convergent property of an event is proved by having a *variant*. The variant needs to be decrease every time the event is triggered but no lower than zero. When it reaches zero, this means the event will stop which preserves the live-lock freeness and allow other events to be triggered.

There are two main differences between Event-B and classical B with regards to refinement of events. In Event-B, several events may refine an abstract event whereas in classical B, only one event can refine an abstract event. The other difference is that in Event-B, we may have new events that refine *skip* whereas in classical B, this is not allowed.

2.5.2 Records Extension in Event-B

Work by Evans and Butler [34] has introduced a method to structure data by records in Event-B by stepwise development through refinement. This method for structuring data is used in the fourth refinement of the ATM case study in Section 3.3.5, Section 3.3.6 and Section 8.7.1.

The method does not involve changes to the semantics of Event-B but adds an extension to its syntax. The record syntax uses the *SETS*, *CONSTANTS* and *AXIOMS* clauses. Consider an example where there is a record R with two fields *r1* and *r2* of type A and B respectively. Using the *SETS* clause, three deferred sets R, A and B are declared, where the sets R represents the record type and the sets A and B correspond to the types of the fields *r1* and *r2*. A VDM composite-like declaration for a record has been proposed which can be seen in Figure 2.4 using the given example.

CONTEXT *Func*
SETS
R::$r1$:A; (1)
$r2$:B;
END

FIGURE 2.4: Syntactic Sugar for Record Types

CONTEXT *Func*
SETS
R
A
B
CONSTANTS
r1
r2
AXIOMS
$r1 \in \mathbb{P}(R)$
$r2 \in \mathbb{P}(R)$
$r1 \in R \longrightarrow A$
$r2 \in R \longrightarrow B$
END

FIGURE 2.5: Event-B Context for Record Types

Figure 2.5 shows the Event-B specification corresponding to the syntactic sugar in Figure 2.4. The declaration (1) specifies the R record as having the field *r1* of type R and

links the messages with their corresponding A. This syntax implicitly types the *r1* as:

$$r1 \in R \longrightarrow A$$

In UML-B, the record R is represented by a classtype.

Two forms of record refinement been introduced which are record extension and record subtyping. For a record extension, the following syntax is proposed:

EXTEND R **WITH** *r3:C* (2)

The declaration (2) specifies that the record R is extended with the field *r3*. In the corresponding Event-B, a constant *r3* is introduced and (2) implicitly types *r3* as

$$r3 \in R \longrightarrow C$$

For subtyping,

Q **SUBTYPES** R **WITH** *r3:C*

where R and Q are the record type and *r3* is a field of type C. In the corresponding Event-B, the record type Q is a constant and it is a subset of the record type R. The field *r3* is of type Q and links the messages with their corresponding C.

An example of record extension is described using an electronic mail delivery system where users can send and receive messages. The context, called *Context* is shown on Figure 2.6. It declares two sets *User* and *Message*, and one record type *Send_ interface* with fields *dest* and *mess*, of type *User* and *Message* respectively.

CONTEXT *Context*
SETS
User;*Message*;
Send_interface::*dest*:*User*;
mess:*Message*;
END

FIGURE 2.6: Abstract Context

The *Context* is refined to introduce more detail and realistic architecture in which each user is associated with a mail server that is responsible for forwarding and retrieving mail from the middleware. This can be seen in Figure 2.7. In *Context2*, the record *Send_interface* is extended to add a new field *source* that contains the identities of the senders. Also, a record type *Package* is declared with fields *destination*, *recipient* and *contents*. Additionally, a new set *Server* is declared to represents the different mail servers and a function *address* which returns the server hosting a particular user.

CONTEXT *Context2*
REFINES *Context*
SETS
Server;
Package::*destination*:*Server*;
recipient:*User*;
contents:*Message*;
EXTEND *Send_interface* **WITH** *source*:*User*
CONSTANTS
address;
PROPERTIES
$address \in User \rightarrow Server$
END

FIGURE 2.7: Refined Context

2.6 Decomposition in Event-B

Decomposition is where an Event-B machine is separated into a number of smaller components which are easier to manage. These decomposed components can be refined independently from one another. There are two kinds of decomposition which may be used in Event-B namely, *event-based* and *state-based*.

Event-based decomposition is proposed by Butler in [23, 24] based on a parallel composition method developed for action systems. Event-based decomposition decomposes a model into separate components on its events. That means, events are split between the components. The variables are encapsulated in the separate components and the events that affect the variables are specified in that component model. The events that have been split need to be synchronised to form the original functionality of the model. Figure 2.8 illustrates the event-based decomposition. The lines connecting the ovals (indicating events) and the box (indicating variables) represent the dependencies. For example, event *e1* may read from or assign value to variable *v1*. *M1* and *M2* are the decomposed components which are sharing the event $e2$. Component *M1* contains events *e1* and *e2* and variable *v1*. *M2* contains events *e2* and *e3* and variable *v2*.

State-based decomposition is proposed by Abrial and Hallerstede in [7]. State-based decomposition splits and shares some variables between the components. Events are added to components to simulate the shared variables in other components. Figure 2.9 is an illustration of the state-based decomposition showing the decomposition of the two components *M1* and *M2*. *M1* contains the events *e1* and *e2* and variables *v1* and *v2*. *M2* contains events *e3* and *e4* and variables *v2* and *v3*. From the figure, it can be seen that the events *e2* and *e3* may read from or assign to the variable *v2*. The reason for this is, when refining component *M1*, any new invariant involving the variable *v2*, it is

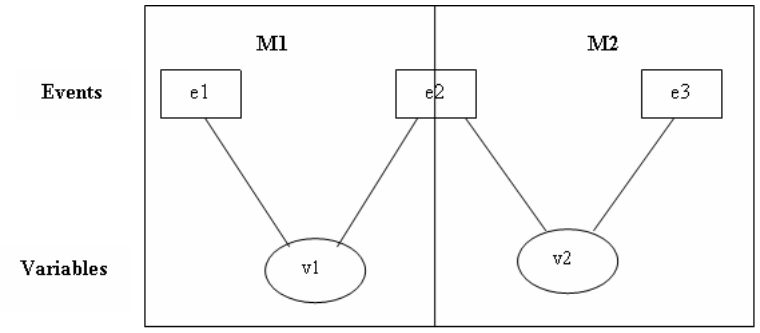


FIGURE 2.8: Illustration of the Event-based decomposition

needed to proof that the invariant holds for not just all events in *M1* but also holds for event *e3*. This is because *e3* has a dependency with *v2*. Therefore in the component *M1*, event *e3* is added. Similarly in *M2*, event *e2* is added.

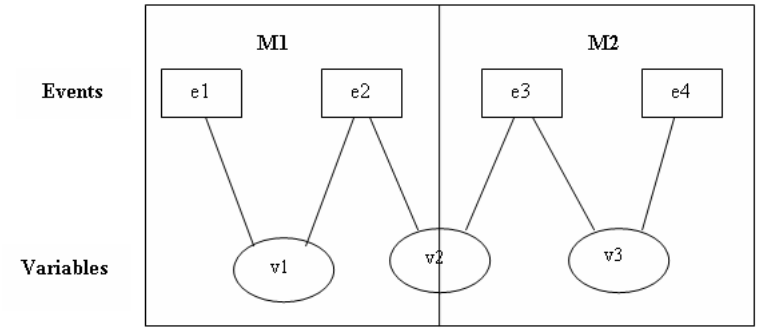


FIGURE 2.9: Illustration of the State-based Decomposition

2.7 Rodin Event-B Tool

Rodin Event-B tool [4, 22] is an open tool set for Event-B development which is built on top of the Eclipse platform. The aim of Rodin is to allow the integration of multiple tools by various parties in order to support rigorous development methods. These tools are integrated into the Rodin tool as plug-ins. This extensibility feature is one of the two main features of the Rodin tool.

The Rodin tool consists of a database that stores the models under development and several other main plug-ins. These plug-ins are the static checker, proof obligation generator and provers. The static checker analyses Event-B contexts and machines and gives feedback to users about syntactical and typing errors. The proof obligation generator produces proof obligations from the Event-B models. The provers compute proofs for the proof obligations. The Rodin Event-B tool includes automatic and interactive provers.

The interactive prover requires users to interact with the tool in order to discharge the proof obligations.

2.8 Unified Modelling Language

The Unified Modelling Language (UML) is a standard modelling language for visualising, specifying and documenting the artifacts of a software system. A modelling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. The UML has been used effectively for such domains as enteprise information systems [73], telecommunications [46], transportation [57], defense/aerospace [33] and distributed web-based services [31].

2.8.1 Diagrams in the UML

The UML includes nine diagram types:

- Class diagram
  A class diagram shows a set of classes, interfaces and collaborations and their relationships. These diagrams are the most common diagrams found in modelling object-oriented systems. Class diagrams address the static view of a system.
- Object diagram
  An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view of a system as class diagrams, but from the perspective of real cases.
- Use case diagram
  A use case diagram shows a set of use cases and actors and their relationships. Use case diagrams address the static use case view of a system. These diagrams are used for organising and modelling the behaviour of a system.
- Sequence diagram
  A sequence diagram is an interaction diagram (shows an interaction, consisting of a set of objects and their relationships) that emphasises the time-ordering of messages. It address the dynamic view of a system.
- Collaboration diagram
  A collaboration diagram is an interaction diagram that emphasises the structural oganization of the objects that send and receive messages. It address the dynamic view of a system.

- Statechart diagram
  A statechart diagram shows a state machine, consisting of states, transitions, events and activities. Statechart diagrams address the dynamic view of a system. They are used for modelling the behaviour of an interface or class and emphasise the event-ordered behaviour of an object which is especially useful in modelling reactive systems.

- Activity diagram
  An activity diagram is a special kind of statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are used for modelling the function of a system and emphasise the flow of control among objects.

- Component diagram
  A component diagram shows the organisations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams since a component typically maps to one or more classes or interfaces.

- Deployment diagram
  A deployment diagram shows a configuration of run-time processing nodes and the components that are used on them. Deployment diagrams address the static deployment view of an architecture. They are related to component diagrams given that a node typically encloses one or more components.

2.8.2 Classes

This sub-section gives an overview of UML classes. A class represents a set of objects that have the same attributes, operations and relationships.

An *attribute* is a property of a class that describes a value that instances of the property may hold. An *operation* is a definition of a service that is provided by any object of the class. A class may or may not have operations. A class can connect to another class with a relationship. Three common relationships are dependencies, generalization and associations. A *dependency* is a using relationship. For example, pipes depend on the water heater to heat the water they carry. A *generalization* is a relationship between a general thing and a more specific kind of that thing. For example, rectangle and circle are more specific kind of a shape. An *association* is relationship that specifies objects of one thing are connected to objects of another. For example, a person works for a company.

2.8.3 State Machines

This sub-section gives an overview of UML state machines. A state machine models the behaviour of an object. It specifies the sequences of states that an object goes through its life time in response to events together with its reactions to those events.

A *state* is a situation during an object's life time. An *event* is a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state may perform certain actions and enter the second state when a certain event occurs and specified conditions are satisfied. An *initial state* indicates the starting place for the state machine whereas a *final state* indicates the execution of the state machine has complete. A *self-transition* is the transition that leaves and reenters the same state. *Sub-states* are an advanced feature of states to simplify the modelling of complex behaviours. A *sub-state* is a state that is nested inside another state called composite state.

UML has been revised leading to the current UML 2.0 release. One of the major motivations for the move to UML 2.0 was to add the ability for models to capture more system behaviour and increase tool automation. A technique called Model Driven Architecture (MDA) offers the potential to develop executable models that tools can link together and to raise the level of abstraction above traditional programming languages. UML 2.0 is vital to the MDA effort [80].

2.9 Model Driven Development Architecture (MDA)

Model Driven Architecture (MDA) [17, 21, 70, 88, 101] is a framework adopted by the Object Management Group (OMG) in 2001. The purpose of MDA is to make software designs easily portable between different operating platforms. The general idea is to develop an abstract business model for the problem domain of a software system using a modelling notation such as UML, which is independent of any operating platform. This allows the designer to focus on the business model that will still be valid as the operating platforms technologies evolve. The objective is to represent the software system which can be transform to a specific operating platform with minimal customization.

Figure 2.10 outlines the MDA approach. The OMG envisages MDA to include a full range of large scale services. At its core is the OMG modelling standards: UML, the Meta-Object Facility (MOF) [77], that provides a standard repository of a model and the Common Warehouse Metamodel (CWM) [76], the standard for integration of data repositories. The layer next to the core layer is the middleware environment. The outermost layer represents various domain of applications of MDA.

MDA provides a conceptual framework for using models and applying transformations

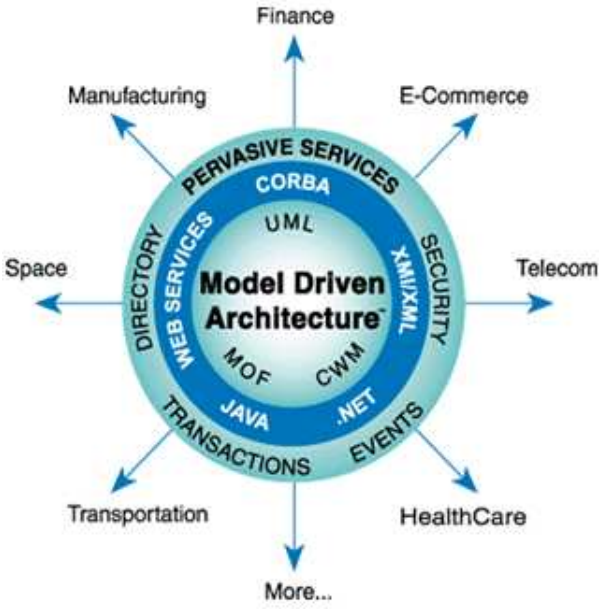


FIGURE 2.10: OMG's Model Driven Architecture

between models. A specific set of layers and transformations that provide a conceptual framework and vocabulary for MDA has been defined by OMG. Four types of models classified by OMG are Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) described by a Platform Model (PM), and an Implementation Specific Model (ISM).

A set of metamodels is important to the model representations and in supporting the transformations. The ability to analyze, automate, and transform models requires a clear, unambiguous way to describe the semantics of the models. The OMG acknowledges the importance of metamodels and formal semantics for modelling, so the MOF is defined as a set of metamodelling levels as well as a standard language for expressing metamodels. The MOF is used by a metamodel to define the abstract syntax of a set of modelling constructs.

The models and the transformations between them are specified using open standards. The OMG has defined a number of important industry standards for specifying systems and their interconnections. Through standards such as CORBA, UML and CWM, a level of system interoperability that was impossible in the past, can be achieved by the software industry.

Figure 2.11 shows a schematic view of how MDA works in practice. The key process is an incremental refinement, that starts with an abstract specification and works on producing a more detailed specification. A refinement definition which is maintainable and can be updated, is controlling the translation process. This process can be repeated many times, with the detailed specification becoming the abstract specification.

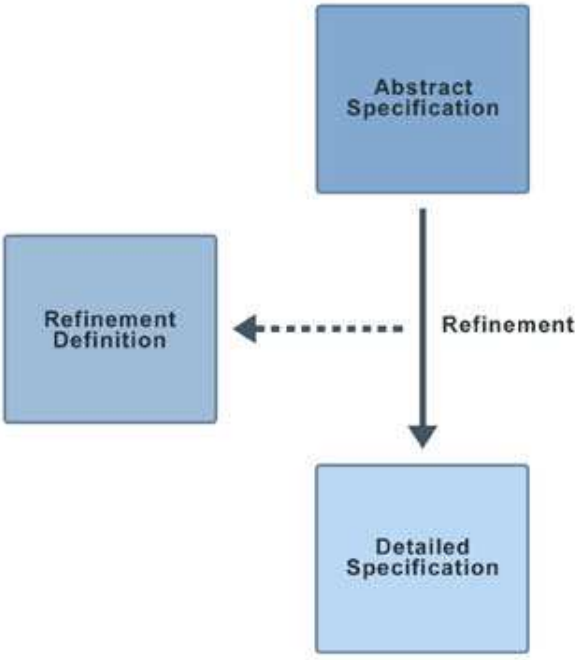


FIGURE 2.11: Controllable Incremental Refinement

2.10 UML-B

UML-B [96, 95, 97] is work that integrates UML and B. The purpose of integration is to address the lack of formal semantics of UML and to make B method more approachable. There are old and new versions of UML-B.

The old or previous version of UML-B is a profile of the UML that defines a subset and specialisation of UML. It is a subset of the UML which includes packages, class diagrams and state charts. The UML-B profile uses stereotypes to specialise the meaning of UML entities. The UML-B profile defines tagged values, which are UML-B clauses, that are used to attach details such as invariants and guards. UML-B provides a diagrammatic, formal modelling notation where B's infrastructure is hidden and mathematical constraints and action specifications are packaged into small sections which are presented in the context of its owning UML entity. This previous version was implemented using the Rational Rose UML tool. A U2B translator was included to generate 'classical B'.

The new or current version of UML-B is a UML-like formal modelling language based on Event-B. UML-B provides four kind of diagrams. They are package, class and state machine diagrams. A package diagram is a top-level diagram that shows the structure and relationships between components (machines and contexts) in a project. A context is described in a context diagram which is similar to class diagram but has only constant data and structured types. Axioms (given properties about the constants) and theorems (assertions requiring proofs) may be attached to classtypes in a context diagram. A

machine is specified by a class diagram and state machine diagram(s) representing data structures that may be changed by events or transitions. Events may be attached to classes in a class diagram. Events can also be represented by the transitions on a state machine diagram. Each UML-B context gives rise to an Event-B context (i.e., the UML-B tool generates a corresponding Event-B context). Each UML-B machine gives rise to both an implicit Event-B context and an Event-B machine. The implicit context is used to define types for the classes and states in the UML-B machine. In the generated Event-B machine classes, attributes and associations become variables. Events and transitions in classes and state machines become events in the generated Event-B machine. Invariants and theorems maybe attached to classes and states.

Micro B (μB) is used as a notation for textual constraints and action. μB borrows from the Event-B notation. μB used an object-oriented style dot notation to show ownership of entities, i.e attributes and operations, by classes. Variables used in an expression can represent owned features using the dot notation. For example, $i.x$ refers to the value of the variable x which belongs to instance i. When an expression is attached to a feature belonging to a class, the owning instance for the current contextual instance is referenced using the reserved word *self*. This new version is implemented in Eclipse and closely integrated with the Rodin Event-B tool. A U2B translator is included to generate Event-B specifications.

2.10.1 UML-B Modelling Environment

This section describes UML-B modelling diagrams which consists of package, context, class and state machine diagrams [96]. Figure 2.12 shows the user interface of the UML-B drawing tool. The left part is the project explorer view which lists the UML-B projects and the diagrams in the projects. The middle part is the drawing canvas for drawing the UML-B model elements. The right part is the tool palette which contains the creation tools for creating UML-B model elements on the drawing canvas. The bottom part is the properties view which contains information about the selected model element on the drawing canvas and reports error messages.

A package diagram defines the structure and relationships between UML-B machines and contexts in a project. Figure 2.12 shows an example of a package diagram. The package diagram consists of the machines *M1* and *M2*, and refinement relationships between them. It also contains the contexts *CX1* and *CX2*, and the extension relationships between them and which contexts are seen by machines. The properties view shows information about the selected machine *M1*.

A context diagram defines the static part of a model. A context diagram may have classtypes. Each classtype may has attributes and associations. Association is a relationship between two classtypes and it is a special case of an attribute. The multiplicity

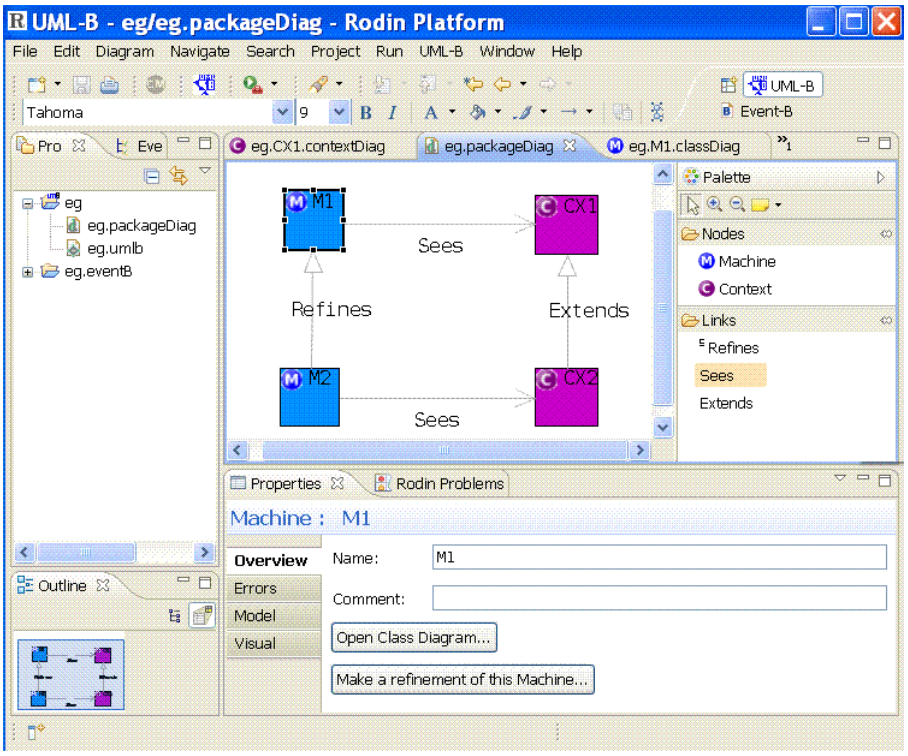


FIGURE 2.12: Example of Package Diagram

properties for attributes are described using mathematical terminology (*surjective, injective, total, functional*) and also with the UML style multiplicity annotated automatically on the diagram (for associations). Figure 2.13 shows an example of a context diagram. The classtype *CUSTOMER* has an attribute, *ident* and an association *accounts* with the classtype *BANK*. The multiplicity of the association accounts indicates that it is a total function. Axioms and theorems may be attached to a classtype. Classtypes are used to define types and also to define constant attributes of those types. The attribute *ident* and the association *accounts* are translated as constants. Apart from the example of a context diagram, Figure 2.13 also shows its generated translation to Event-B. Each Event-B statement is preceded by its label which defines its purpose. For example, *ident.type* is a label for the Event-B statement $ident \in CUSTOMER \leftrightarrow \mathbb{N}$.

In UML-B, another kind of association between two classtypes, *subtyping*, changes the identity of a class. Figure 2.14 shows an example of subtyping the classtype *Account* into *CurrentAccount* and *SavingAccount* classtypes that changes the identity of an account into current account or saving account. In an Event-B machine, the subtyping association will give a type of the *CurrentAccount* and *SavingAccount* classtypes as *Account* i.e., $CurrentAccount \subseteq Acc$ and $SavingAccount \subseteq Account$. The sub-classtypes *CurrentAccount* and *SavingAccount* give rise to constants in the generated Event-B context.

A class diagram is used to describe the behavioural part of a model. A class diagram may contain classes. Each class may have attributes, associations, events and state machines.

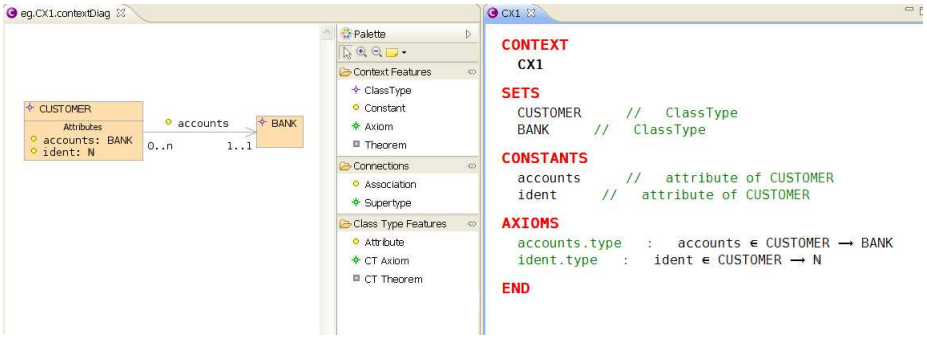


FIGURE 2.13: Examples of Context Diagram and Event-B Translation

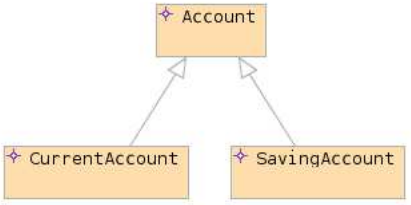


FIGURE 2.14: Subtyping a Classtype in UML-B

Classes may represent subsets of the classtypes in a context diagram. An attribute defines a data value of an instance of a class. An association is a special case of an attribute that defines a relationship between two classes. Class events replace traditional object oriented methods. An event defines operations of a class and involves modification to some or all the attributes of a class. A state machine defines the behaviour of a class in terms of transition between discrete states. Associations and attributes of a class are similar to those in the context but instead of constants, they become variables in the generated Event-B machine. Figure 2.15 shows an association *accounts* between the *account* class and *bank* class. This association will be translated to Event-B as a variable of type $bank \leftrightarrow account$ and initialised to $\{\}$. Class *account* consists of attributes *odLim* and *balance*. The invariant in class *account* specifies that the account's *balance* must be greater than its overdraft limit, *odlim*.

Similar to the subtyping association between classtypes, a class my also subtype another class. The difference is the classes give rise to variables in the generated Event-B machine.

A state machine diagram may be attached to a class. The state machine *bal_state* in Figure 2.16 shows its two states, *black* and *red* and the transitions. The solid circle is the initial state, whereas, the solid circle with an outer circle is the final state. A transition can be triggered when the instance is at its source state, and changes the instance state to the target state. For example, the transition or event *withdraw* can be triggered when the instance is at state *black* and move to state *red*. A transition

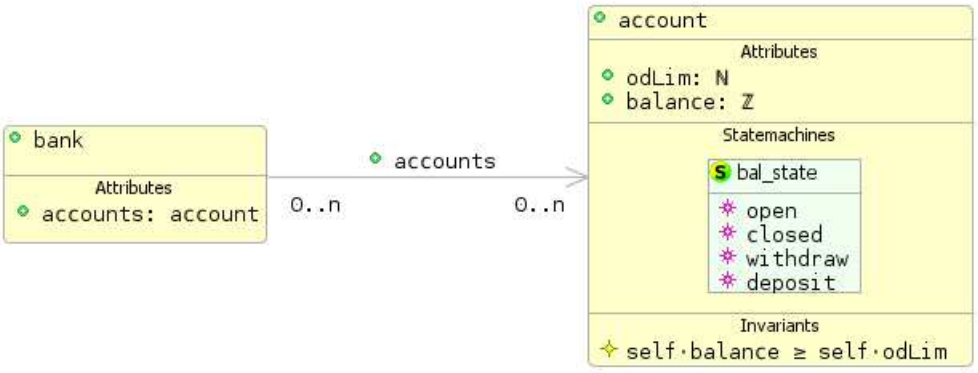


FIGURE 2.15: Example of Class Diagram

may have parameters, guards and actions which are define explicitly in the properties. Figure 2.17 show the properties for the transition withdraw showing its parameter, guards and actions. The translation to Event-B for a state machine can either be

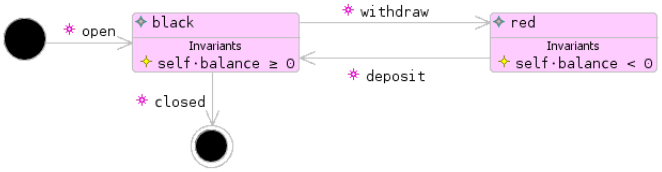


FIGURE 2.16: Example of a State Machine Diagram

a disjoint sets representation or state function representation. These two styles are introduced in [26] and they are supported in the UML-B tool. UML-B allows modellers to switch between these two representation. For a disjoint sets representation, a disjoint

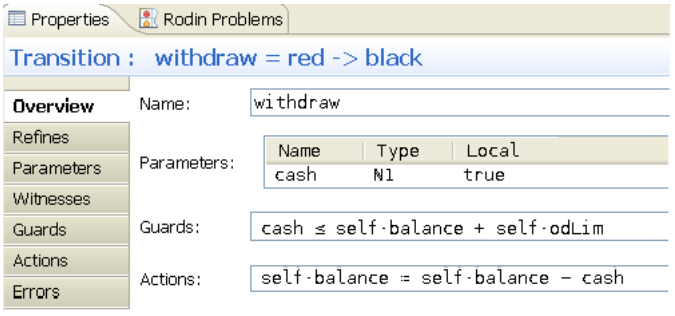


FIGURE 2.17: Properties of the *withdraw* Transition

sets of *account* are introduced as variables as follows:

$$black \in \mathbb{P}(account)$$
$$red \in \mathbb{P}(account)$$
$$black \cap red = \varnothing$$

That is, variable *black* represents the set of instances of account that are in the state *black* and similarly for *red*. An invariant specifying that the states are disjoint is generated in the Event-B machine. For a state function representation, a variable *bal_state* (i.e., the state machine belonging to the class *account*) is introduced representing a function mapping *account* to an enumerated set of states, *bal_state_STATES* as follows:

$$bal\_state\_STATES = \{black, red\}$$
$$bal\_state \in account \longrightarrow bal\_state\_STATES$$

That is, *bal_state* maps each instance of *account* to its state. The generated Event-B machine for *M1* using the state function representation is shown in Figure 2.18. The transitions represent events with additional behaviour associated with the change of state implied by the transition. An instance of *account* changes its state when a transition fires. For each transition there is a guard that specifies an instance source state (labeled as *..isin..*) and action that specify its target state (labeled as *..enterState..*). The parameter, *self*, indicates an instance of a class. A transition from an initial state such as *open*, defines a constructor for the class. The translation of *open* selects an unused instance and adds it to the set of *account* (labeled *self.type*). A transition to a final state such as *closed* is a destructor which removes an instance from current instances and from the domain of all the class variables. Figure 2.19 shows a translation

```
open   ≙
STATUS
  ordinary
ANY
  self     //   constructed instance of class account
WHERE
  self.type   :   self ∈ ACCOUNT \ account
THEN
  account_constructor   :   account ≔ account ∪ {self}
  bal_state_enterState_black   :   bal_state(self) ≔ black
END

withdraw   ≙
STATUS
  ordinary
ANY
  self     //   contextual instance of class account
  cash
WHERE
  cash.type   :   cash ∈ N
  self.type   :   self ∈ account
  bal_state_isin_black   :   bal_state(self) = black
  withdraw.Guard1   :   cash > balance(self)
  withdraw.Guard2   :   cash ≤ balance(self) + odLim(self)
THEN
  bal_state_enterState_red   :   bal_state(self) ≔ red
  withdraw.Action1   :   balance(self) ≔ balance(self) − cash
END
```

```
closed   ≙
STATUS
  ordinary
ANY
  self     //   contextual instance of class account
WHERE
  self.type   :   self ∈ account
  bal_state_isin_black   :   bal_state(self) = black
THEN
  bal_state_finalState   :   bal_state ≔ {self} ⩤ bal_state
  account.balance_destructor   :   balance ≔ {self} ⩤ balance
  account_destructor   :   account ≔ account \ {self}
  account.odLim_destructor   :   odLim ≔ {self} ⩤ odLim
END
```

FIGURE 2.18: Generated Event-B for the Events Using State Function Representation

of the *withdraw* event using the state sets representation. The difference between the state sets representation with the state function representation can be seen at the statement labeled *..isin..* that defines the current state of an instance and at the statement *..enterState..* that defines the target state of an instance. With the state sets representation, an additional action is generated for the event *withdraw* to specify the departure of an instance from the state *black* (labeled as *..leaveState..*).

```
withdraw   ≙
STATUS
  ordinary
ANY
  self     //   contextual instance of class account
  cash
WHERE
  cash.type   :   cash ∈ N
  self.type   :   self ∈ account
  bal_state_isin_black   :   self ∈ black
  withdraw.Guard1   :   cash > balance(self)
  withdraw.Guard2   :   cash ≤ balance(self) + odLim(self)
THEN
  bal_state_enterState_red   :   red ≔ red ∪ {self}
  bal_state_leaveState_black   :   black ≔ black \ {self}
  withdraw.Action1   :   balance(self) ≔ balance(self) − cash
END
```

FIGURE 2.19: Generated Event-B for the *withdraw* Event Using State Sets Representation

2.10.2 UML-B Metamodel

The UML-B metamodel [96] is using UML class diagrams to defined the abstract syntax of the structure of the UML-B language. The UML-B metamodel is described using a small subset of UML's class diagram features that corresponds to the OMGs Meta Object Facility (MOF). Generalisation is used extensively which ensures that common attributes in UML-B model elements are defined. The metamodel is an exact description of the abstract syntax of the UML-B language and is used to generate automatically repository and editing utility code using the Eclipse modelling Framework (EMF) technology. The Eclipse Modelling Framework (EMF) [36] is a framework and code generation facility for building applications based on a model. Another Eclipse framework, the Graphical Modelling Framework (GMF) [37], is used to generate automatically the code for UML-B graphical modelling tool from the EMF model. Any constraints are modelled on the corresponding metaclasses as operations but have to be populated with Java bodies after code generation. These constraints are added to prevent creating invalid models.

Figure 2.20 shows part of the UML-B metamodel. There are three kinds of relationship used between classes in the metamodel that are, subtyping or specialisation, association and containment. An example of a subtyping (a link with triangle arrowhead) is between the class *UMLBPredicate* and *UMLBelement*. This relationship indicates that the metaclass *UMLBPredicate* inherits all the attributes of the metaclass *UMLBelement*. An example of an association (a link with arrow) is from the class *UMLBMachine* to itself. This association specifies that a machine may refine at most one machine and a machine may be refined by many machines. An example of a containment (a link with solid diamond arrowhead) is between the class *UMLBContext* and *UMLBClassType*. The containment relationship specifies that a context may contain many classtypes.

The class *UMLBelement* is a base class and its provides a name and error marking to all model elements. *UMLBconstrainedElement* is a subtype of *UMLBelement* and it provides a base for elements that own constraints and theorems. Another subtype of the base class is *UMLBPredicate* which has a string attribute, *predicate*, representing

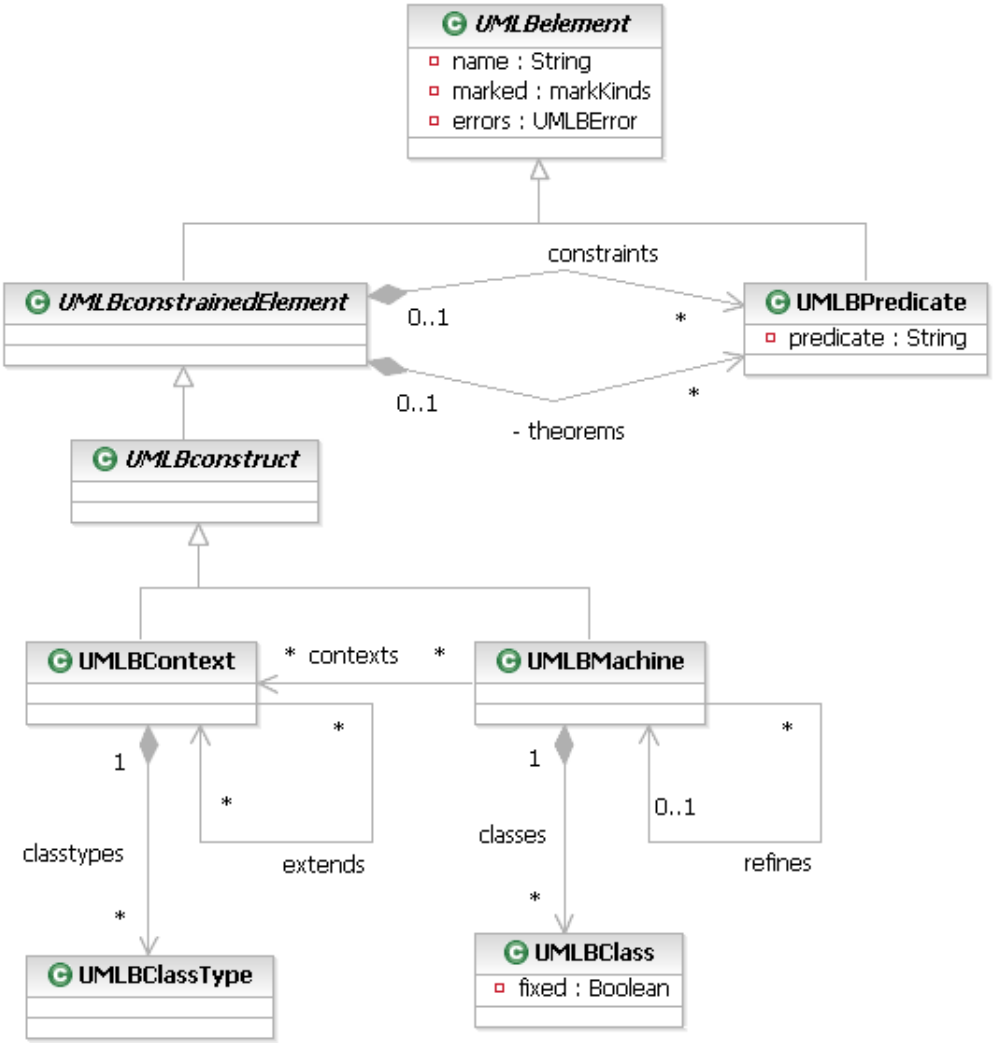


FIGURE 2.20: UML-B Metamodel (part of)

the syntax of predicates. The definition of this syntax depends on that of Event-B with a few changes to adapt the UML-B object oriented features. *UMLBMachine* and *UMLBContext* are subtypes of *UMLBConstruct* and they reflect the main modelling components of Event-B. *UMLBMachine* can have *UMLBClasses* and *UMLBContext* can have *UMLBClassType*. Figure 2.20 leaves out many features of the metamodel such as statemachines, variables and events.

2.11 Other Modelling Languages

This section gives an overview of Z and VDM modelling languages which are similar to the B method.

2.11.1 Z Specification Language

Z [20, 19, 64, 98] was initiated by Abrial in France and has been developed at Oxford University since late 1970s by members of the programming Research (PRG), lead by Hoare. Z is a typed language based on set theory and first order predicate logic. Compared to B, Z is designed mainly for specification and proof, whereas, B is a tool-based formal method for software development.

The Z notation consists of mathematical language and schema language. The mathematical language is used to describe states, properties and operations. The schema language is used for structuring the specification. A schema can be written horizontally as follows

$$\text{Schema name} \triangleq [\text{Declaration} \mid \text{Predicate}]$$

or vertically as shown in Figure 2.21 [87]. The *declaration* part contains variables and the *predicate* part specify values of variables.

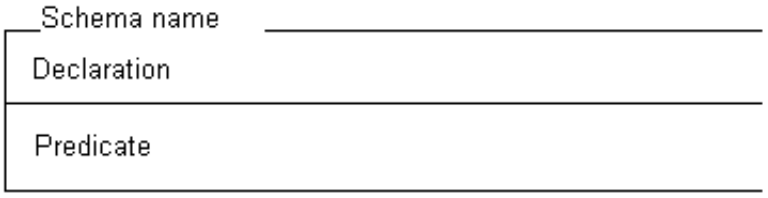


FIGURE 2.21: Schema Definition

Schemas are used to model states (state schema) and operations (operation schema) in Z. State schemas describe the states types and relationship between states and restriction on the states. Operation schemas describe the changes of these states before and after the execution of the operations. *Schema Calculus* is used to construct a large Z specification from a number of schemas. This is done by using the schema operators such as conjunction, disjunction, composition and others.

The specification of a football team in Figure 2.2 which is modelled using the B method is used as an example for state schema and operation schema. A set $ANSWER$ consists of the elements in and out is declared as follows:

$$ANSWER ::= in \mid out$$

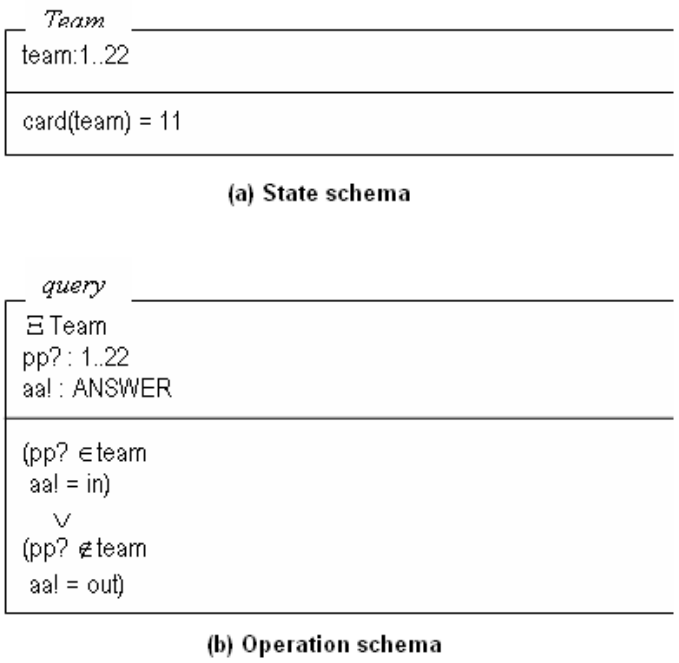


FIGURE 2.22: A Z Specification of the Team Schema

The state schema is shown in Figure 2.22(a) and the operation schema is shown in Figure 2.22(b). The state schema specified that a set team is a subset of a sequence integer from 1 to 22 and the total elements of the set team is 11. The operation *query* tells whether a particular player is in the team or not. The Xi notation in the first line of the operation schema means that the schema *query* extended the schema *Team* and the schema *query* does not change the state data *team*. The symbol *?* after the variable *pp* indicates that *pp* is an input to the operation and the symbol *!* after *aa* indicates that *aa* is an output from the operation.

2.11.2 Vienna Development Method (VDM)

VDM [52, 87] was invented at the IBM Vienna laboratory carried out by a number of different researchers in the general areas of programming language definition.

VDM provides a language as a notation for modelling specifications. Similar to B, VDM also is a tool-based formal method for software development. There exists an ISO standard [50] of the VDM-SL specification language.

VDM specification are represented by a module. Shown in Figure 2.23 is the simplest form of a module. The main components of a VDM specification are listed below:

- type definitions - which introduces the various types to be used in the specification.

module *VDM-module*
definitions
types
.
.
values
.
.
state
.
.
functions
.
.
operations
.
.
end *VDM-module*

FIGURE 2.23: VDM Module

- values definitions - are used to introduce global constants into the specification and also to give values to variables.
- state definitions - represents the mechanism by which a module retains a knowledge of the history of operation calls.
- operation definitions - specifies the behaviour of a system.
- functions definitions - a way of defining a rule for obtaining a result from zero or more arguments.

The specification of a football team in Figure 2.2 which is modelled using the B method is used as an example for VDM module shown in Figure 2.24. The specification maintains the set of players during a football game where there are 22 players and there are 11 players in a team. The operation *query* tells whether a particular player is in the team or not. The second line states that *ANSWER* is a set with two elements *in* and *out*. The keyword *inv* defines the invariant that 11 players will be formed into a team. The keyword *init* initialised the set team to the first 11 players. The keyword *ext* after the operation *query* means the operation will access the component *team* of the state *Team*. The keyword *rd* indicates that the component *team* is to be read only by the operation *query*. The keyword *post* indicates the post-condition of the operation.

```
types
        ANSWER={in,out}

state Team of
        team : 1..22
inv   mk-Team(team) ≜
      card(team) = 11
init  mk-Team(team) ≜
      team = 1..11
end

query(pp:1..22) aa:ANSWER
ext rd team:1..22
post  if  pp ∈team
      then  aa = in
      else  aa = out
```

FIGURE 2.24: A VDM Specification of the Team Module

2.11.3 Comparison between B, Z and VDM

This section gives a comparison of Z and VDM with B in terms of the structuring mechanism, the treatment of precondition, invariants and proof obligations, tool support and refinement.

Structuring Mechanism B and VDM have a clear structuring mechanism which is very different from Z which uses schema. VDM models are structured in modules and B models are structured in machines. B has the notion of layered development which allows a decomposition of a complex development.

Preconditions, Invariants and Proof Obligations The treatment of preconditions and invariants in B, Z and VDM are different. In B and VDM, the preconditions are explicit whereas in Z, preconditions are calculated from the delta schema definitions. B and VDM differ with their treatment of the invariant. In VDM, an invariant is assumed to be an implicit part of every pre and post condition. Because of this, the only proof obligation that has to be discharged with respect to an operation is one of feasibility. In B, an invariant is not assumed to be part of the post condition of operations. The statement of an invariant is redundant with the operation and does not change the meaning of the operation. The operations have to be defined in a way that they preserves the invariant. Because of this redundancy, proving that every operation preserves the invariant is necessary.

Tool support B, VDM and Z are supported by a number of tools. For B, there are tools called B-Toolkit, AtelierB, ProB and Rodin. VDM tools are such as VDMTools and Overture. Example of tools for Z are Proofpower and Z/Eves.

```
uses TaskQueue;
mutable type queue;
immutable type task;
task *getTask(queue q){
   modifies q;
   ensures
      if isEmpty(q^)
         then result = NIL ∧ unchanged(q)
         else (*result)' = first(q^) ∧ q'=tail(q^);
```

FIGURE 2.25: LCL Interface Specification

Refinement B, VDM and Z have a one-to-one relationship between refinements for event refinement. In contrast, Event-B has a many-to-one relationship between refinements. Also, Event-B refinement may have new events.

B, Z and VDM are similar in that they are based on the construction of models and they are not used with any programming language, as opposed to Larch and JML which support the use of formal specifications in programming languages.

2.12 Programming Specification Languages

This section gives an overview of Larch and JML as examples of programming specification language.

2.12.1 Larch

Larch [43] is a family of languages which support two-tiered specification style and is adapted to several programming languages. A Larch specification is written in two languages. One language is designed for a specific programming language which is called the *Larch interface language*. Another language which is independent of any programming language is called the *Larch Shared Language (LSL)*. There are *Larch interface languages* for C [44], C++ [59], Modula-3 [53], Ada [82], ML [103], Smalltalk [29], CLU [48].

The interface language is used to specify the interfaces between program components. How the components communicate across the interface is a critical part of each interface. Methods of communication from programming language to programming language are different. Writing the interface specification language that reflects a particular programming language makes it easier to be precise about the communication. Generally, a specification written in such interface language is shorter that specification written in a "universal" interface language.

An example of an interface specification for a small portion of a scheduler for an operating system can be seen in Figure 2.25. The specification is written in LCL which is a *Larch interface language* for C programming language. The *mutable* clause means the abstract value of an object queue can change over time. The *immutable* clause means the abstract value of an object task cannot be changed. The clause *ensures* specified a postcondition which is typically a predicate that is defined on the pre-state and post-state of a procedure call. The clause *modifies* specifies a list of objects that can be modified by an execution of the specified procedure. The specification consists of two abstract types *queue* and *task* and a procedure for selecting a task from a task queue, named *getTask*. The symbol * represents a pointer as in C, *result* is the value returned by the procedure *getTask*, the symbol ^ represents the value in a location when the procedure is called (pre-state) and the symbol ' represents the value when the procedure returns (post-state).

2.12.2 Java Modelling Language (JML)

JML [40, 42, 41, 78] is a notation for formally specifying the behaviour and interfaces of Java classes and methods. It is a behavioural interface specification language for Java that builds on the Larch family of interface languages. JML adds annotations to Java code that allows designers to specify the functions of methods without specifying how it is to be implemented. JML adopts many Java's expression syntax and also attempts to have similar semantics to Java.

```
public interface Gendered {
  //@ model instance String gender;

  //@ ensures \result <==> gender.equals("female");
  /*@ pure @*/ boolean isFemale();
}
```

FIGURE 2.26: A JML Specification of the Interface *Gendered*

Figure 2.26 shows an example [41] of a JML specification of the interface *Gendered*. The JML annotations are written in comments start with at-sign (@). The modifier *model* means the field *gender* is not used in Java code but is only used for specification as an abstraction of concrete states. The modifier *instance* means the field *gender* is a non-static field in all classes that implements the *Gendered* interface. Omitting the modifier instance will make the field *gender* static and final like the default fields in Java. Method specifications are written before the header of the Java method being specified. In the example, the clause *ensures* specifies the postcondition of the method *isFemale*. The postcondition says that the method return value (represented by *return*) is equivalent to "*female*". The clause *pure* means a correct implementation of the method *isFemale* has

no side effects. The clause pure implicitly includes the specification *assignable \nothing* which means there is no assignment made to any fields.

Figure 2.27 shows part of class *Animal* that implement the interface *Gendered*. The class Animal inherits the abstract field *gender* via its concrete state *gen*. The *represents* clause represents the expression for the value of the field gender so that whenever gender occur in specification, its value is given by the expression of this represent clause. JML

```
public class Animal implements Gendered {
  protected boolean gen;
  //@ protected represents gender <- (gen ? "female" : "male");

}
```

FIGURE 2.27: A JML Specification of the Class *Animal* (part of)

specifications also contain invariants, which are predicates that must be hold in the pre-state and post-state of each method execution. The example in Figure 2.28 shows two *invariant* clauses. Both invariants are declared as public instances. The first invariant indicates the value of field *age* is between 0 and 150. The second invariant indicates that all elements of the *List history* are instances of type *String*. The *spec_public* modifier in the declaration of *history* is similar to the concept of model field. The *rep* keyword represents ownership. For example in the declaration of *history*, it means *history* is owned by the *Patient* object. The clause *requires* in the constructor specification of Figure 2.28 specify precondition. The precondition indicates that the argument *g* must be either "*female*" or "*male*".

```
public class Patient extends Person {
  //@ public invariant 0 <= age && age <= 150;

  protected /*@ spec_public rep @*/ List history;

  //@ public invariant (\forall int i; 0 <= i && i < history.size();
                                history.get(i) instanceof rep String);


  //@ requires g.equals("female") || g.equals("male");
  //@ ensures gender.equals(g);
  public Patient(String g) { super(g); history = new /*@ rep @*/ ArrayList(); }
}
```

FIGURE 2.28: A JML Specification of the Class *Patient* (part of)

2.13 Other Work on the Integration of Formal Methods with UML

The work of integrating UML with formal methods has been investigated for the past few years. Related work on integration of UML with B method is by Lano et al [54]. They have extended the RSDS method [10]. In RSDS, a small subset of UML state chart diagrams was used as specification and from the specifications, B, Java and SMV are generated. But the RSDS is not suitable for a big control system where a larger subset of UML is required. [54] extends RSDS by: (1) Translating UML class diagram in to B (2) Translating a subset of OCL into B (3) Synthesizing the code of methods from OCL contraints. In the paper, a translation from UML class diagrams and a subset of OCL into B is defined. They have presented the translation from UML-RSDS model into B for types, classes and associations, inheritance, constrained attached to a class (class invariants), association constraints. They have described the synthesis of event code in B from the constraint written in the language LOCA which is a variant of OCL 2.0.

Integration between UML and B also has been investigated by Ledang et al [62, 63]. They have described their approach for deriving B specifications from use case diagrams, class operations and state-chart diagrams. Each use case is modelled as a B operation. Each use case and its involved classes are modelled in the same B abstract machine. The B operations of the included and extended use cases refines or implements the operations of the use cases being includes and extends. Each class operation is modelled as a B operation in an abstract machine. The B operation of the called operation (from interaction or activity diagrams) is called in the refinement of the B operation of the calling operation, i.e., the abstract machine for the called operation is imported in the refinement machine for the calling operation. Their approach of translating the state chart diagrams are in two stages: (1) Each event in the state chart is translated as a B abstract operation and the data are derived from the states in the state chart. (2) The abstract B operation in (1) is refined by calling B operations for the triggered transition and actions. A prototype has been developed AgroUML+B which transform a class diagram and state chart diagrams into a B specifications.

Idani et al [49] have investigated the reverse approach in which they propose an approach and its tool support for the construction of UML class diagrams from B specifications. They have introduced a notion of *pertinent context*. This notion ensures that each class and each operation appears only once in the class diagram. They have presented an algorithm to identify a set of pertinent contexts. The paper also describes the transformation rules that transform the contexts into classes. A tool has been developed to generate the class diagrams.

Laleau and Mammar [56] have worked on translating UML diagrams to B specifications for database applications. Translations into B have been presented from class diagrams,

state diagrams and collaboration diagrams including the basic update operations of database applications. Each class and association class are translated as B machines. Each state machine which is attached to a class is also translated as a B machine. A collaboration diagram is translated as one B machine. They have suggested a three level architecture of B models which consists of B machines and their relationship using the *uses* and *includes* clauses. This architecture is transform from the three kinds of UML diagrams in a UML model. OCaml language [104] was used to implement the automatic translator from the UML models into B models.

Integration of UML with Z has also been investigated. Moller et al [66] have integrated the formal method, namely CSP-OZ [35] into UML and Java. A UML profile for CSP-OZ is developed. A UML profile contains an extension mechanism that consists of *stereotype* and *tag* definitions. This profile which integrates UML and CSP-OZ is similar to the UML-B profile [97] of previous version of UML-B. In this work, class diagram, state machine and the UML-RT [86] structure diagrams are translated to CSP-OZ specifications. Each UML class is translated to CSP-OZ class where the attributes and methods are obtained from class diagrams. The CSP part is obtained by translating the state machine associated with the classes. The architecture of the system modelled in the structure diagrams is translated into a CSP part involving parallel composition. The specification in CSP-OZ is used to generate JML and CSP*jassda* [71] specification. Then, the final hand-written Java program is checked against the assertions in JML and CSP*jassda* to preserve the precision of the formal specification in the implementation.

Amalio et al [74] also have investigated an integration between UML and Z. They have introduced a framework called UML+Z for building, analysing and refining models bases on UML and Z. UML+Z models consists of class, state and object diagrams. An important feature of the framework is a catalogue of *templates* and *meta-theorems*. Templates are the generic representation of sentences of formal languages that when instantiated, produced the actual language sentences. They have developed the formal template language (FTL) to express the framework templates. FTL enables an approach to proof with template representation of Z and to produce meta-theorems. Example of meta-theorem is formulation of a pre-condition. The meta-theorems may reduce the proof effort. In the reported work, the process of instantiating templates is manual.

The integration work of UML and VDM has been done by Frey [38]. Frey has listed two drawbacks of modelling using UML. One of the drawbacks is the lack of precision and ambiguity of the use case notation and the other drawback is the difficulty in finding a suitable set of classes using the entities found in the use case alone. Frey has introduced a methodology where UML and VDM-SL are used together in modelling to take advantage of both notations. Frey has outlined a step by step approach that combines both notations and presented the case study Seating Arrangement System (SAS) to describe the concept of combining UML and VDM-SL. The case study starts with a list of twenty one user requirements. Based on the that requirements, eight are

considered as the functional requirements and a use case was built. Next is creating the VDM specifications. During creating the specifications, some user requirements need to be changed because of their inadequacy exposed by VDM-SL. In the paper, Frey has claimed that the gap between a use case diagram and the task of finding a set of classes was easier using the data models of the VDM-SL specification.

Lausdahl et al [55] have work on a bi-directional translation between UML class diagram and VDM++. The translation of the sequence diagram is done from UML to VDM++. The translations are implemented as a plug-in to the Overture [58] toolsets. The object-orientation features of both languages made them easier to be translated to each other compared to B. The translation of sequence diagram is intended to automate test input into VDM++. The sequence diagrams are transformed to VDM++ trace definitions. This traces of VDM++ enable to define a set of test cases.

2.14 Summary

This chapter presented the related background studies of the research. The background studies on the Event-B, UML, UML-B, refinement and decomposition provides the motivation for the notions introduced in this research which will be described in the following chapters. Other related background studies included in this chapter were formal methods, classical B, Rodin tool, MDA, Z, VDM, Larch, JML and other work on the integration of UML and formal methods.

Chapter 3

An ATM Case Study in Event-B

3.1 Introduction

This chapter contains a description of the development of an auto teller machine (ATM) case study in Event-B using the Rodin Event-B tool. This development provides insight into modelling and refinement in Event-B that helps to understand what is required in UML-B to support refinement.

3.2 Case Study: Auto Teller Machine System

This section gives a description of an ATM system and a summary of the cash withdrawal requirements.

3.2.1 Description of the ATM system

The auto teller machine is a machine that allows bank customers to do some of the banking transaction 24 hours per day. It allows bank customers to withdraw cash, check account balance, print mini statement and others. In order to perform these functions through an auto teller machine, bank customers need to use their ATM cards which are provided to them by the bank. The case study focused only on the requirements for cash withdrawal so only the requirements for this function are focused on in this case study.

3.2.2 Summary of the ATM Requirements for cash withdrawal

The cash withdrawal function allows a user to withdraw money if the withdrawal amount is less than the account balance. During a withdrawal, a possible exception is that a

withdrawal amount is more than account balance. For this exception the bank will not permit the transaction and will respond with a failure status. The development in Event-B for the case study is based on the requirements document in Appendix A.

3.3 Event-B Model for ATM

This section describes the development of the ATM system that includes the abstract machine and six refinement machines. The summaries for each machine are as follows:

Abstract Machine This level models the cash withdrawal function.
First Refinement This level introduces a mechanism of achieving the cash withdrawal function through an ATM machine using an ATM card.
Second Refinement This level models the communication between an ATM and the bank as a single system state.
Third Refinement This level refines the communication between an ATM and the bank as two separate states.
Fourth Refinement This level introduces a form of communication between an ATM and the bank using message passing and is modelled using a record style. In this refinement, the passing of messages are via two channels for request and response messages.
Fifth Refinement This level extends the record of the fourth refinement by adding more fields and also subtyping the record type into two subtypes for request only and response only messages.
Sixth Refinement This level refines the two channels communication for request and response messages into one channel in order to reflect the actual implementation.

The abstract machine was refined into six refinement levels in order to achieve more concrete model. These refinements used a set of patterns proposed by Ball and Butler as described in [16]. Each refinement has a small gap between itself and its abstract machine in order for the refinement to be proved easily.

There are two kinds of refinement which were applied in modelling refinements of ATM, that is, superposition refinement and data refinement. Superposition refinement means incrementally adding ATM requirements in successive refinements whereas data refinement introduces design or implementation detail in successive refinements. The first, second and third refinements are superposition refinements whereas the fourth, fifth and sixth refinements are data refinements. In this work we focus on safety-preserving refinement and do not deal with liveness. In this development, it is assumed that within the system there are many auto teller machines (ATMs) and there is only one bank.

The Event-B specification for the abstract machine will be shown in the following section. The Event-B specifications for the refinement machines are in Appendix C.

CONTEXT ATMC
SETS
ACCOUNT

FIGURE 3.1: ATM Context

3.3.1 ATM Abstract Machine

The ATM abstract machine named *ATMM* modelled one of the core functionalities of banking transaction which is cash withdrawal. Machine *ATMM* sees the context *ATMC* shown in Figure 3.1. The context contains only one set which is *ACCOUNT* that differentiates accounts in a bank. Figure 3.2 shows the abstract machine for the ATM system. The abstract machine has two state variables *account* and *bal*. The variable *account* represents the set of accounts that currently exist in the system. The variable *bal* represents the balances of accounts. The abstract machine has two invariants which specified that *account* is a subset of the set *ACCOUNT* (inv1) and that specifies balance is defined for all account (inv2). The cash withdrawal specification is specified in the event *withdraw*. The withdraw event can only be triggered if all the guards are met.

3.3.2 First Refinement

The first refinement of the *ATMM* abstract machine is where a mechanism of using an ATM card is added to perform cash withdrawal.

The *ATMC* context is extended by *ATMC_E*1 which consists of three sets namely, *CARD*, *PIN* and *ATM* which defines the types that are going to be used in *ATM_R1*. These types are introduced to differentiate ATM cards, pin numbers and machines respectively.

A control state for an ATM was defined to impose an order of events and states. The state diagram can be seen in Figure 3.3. An ATM can be in one of the following five states: *idle*, *validating*, *transactionOption*, *performWithdrawal* or *endWithdrawal*. An ATM which is not being used is in the *idle* state. From the *idle* state, an ATM may go to the *validating* state when the event *insertCard* which represents an ATM card been inserted to the ATM machine is triggered. From the *validating* state, an ATM may go to the *transactionOption* state on successful ATM card validation (represents by the event *validateCardOK*). The event *validateCardOK* will check that the inserted card is valid based on the pin number. For simplicity, it is assumed that there is no exception regarding card validation. From the *transactionOption* state, an ATM may go to the *performWithdrawal* state when the event *withdrawBankOK* or event *withdrawFail* is triggered. *withdrawBankOK* is an event which deducts the withdrawal amount from

MACHINE ATMM

SEES ATMC

VARIABLES

$account$
bal

INVARIANTS

$inv1 : account \in \mathbb{P}(ACCOUNT)$
$inv2 : bal \in account \rightarrow \mathbb{N}$

EVENTS

INITIALISATION

BEGIN
$act1 : account := \varnothing$
$act2 : bal := \varnothing$

END

EVENT withdraw

ANY
ac
am

WHERE
$grd1 : ac \in account$
$grd2 : am \in \mathbb{N}$
$grd4 : am \leq bal(ac)$

THEN
$act1 : bal(ac) := bal(ac) - am$

END
END

FIGURE 3.2: ATM Machine in Event-B

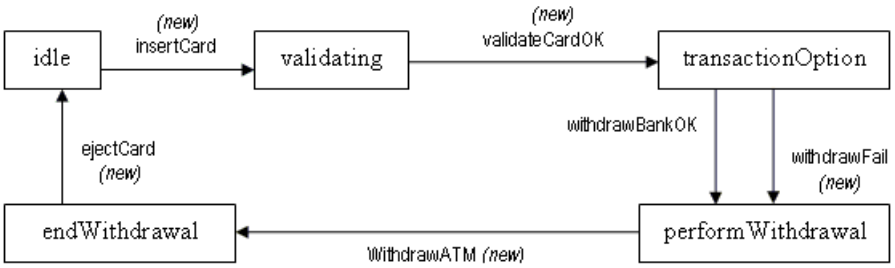


FIGURE 3.3: ATM System State

the bank account provided that all the conditions are fulfilled. *withdrawFail* is the event which handles the exception when the balance in the account is less than the withdraw amount which prevents the withdraw transaction from happening. From the *performWithdrawal* state, an ATM may go to the *endWithdrawal* state when the ATM dispenses cash (represents by event *withdrawATM*). Finally, from the *endWithdrawal* state, an ATM may go to *idle* state when *ejectCard* event is triggered which will eject the ATM card from the ATM.

A variable $atm \in \mathbb{P}(ATM)$ is introduced to model the set of ATM that have been created. Another variable $active\_atm \subseteq atm$ is introduced to model the set of ATM which are being used, that is an ATM which has an ATM card in it. The states of an ATM are modelled using the disjoint sets representation as outlined in Section 2.10.1 where the states are defined as disjoint sets of *atm*.

In the first refinement, the two abstract variables are kept and some new variables are added. Five new events are added namely *insertCard*, *validateCardOK*, *withdrawFail*, *withdrawATM* and *ejectCard*. In this refinement, the event *withdrawBankOK* refines the abstract event *withdraw*.

3.3.3 Second Refinement

The second refinement models the communication between the ATMs and the bank for accessing the cash withdrawal function. In this refinement, the communication is modelled as a single system state where the two states (*transactionOption* and *performWithdrawal*) of the ATM system state in Figure 3.3 are refined into the diagram shown on Figure 3.4. The state *transactionOption* is refined into sub-states *trans* and *waiting* while *performWithdrawal* state is refined into sub-states *processingBank* and *complete*.

The communication is modelled by adding the two new events namely, *request* and *response*, which represent a transaction request to the bank and also the response to

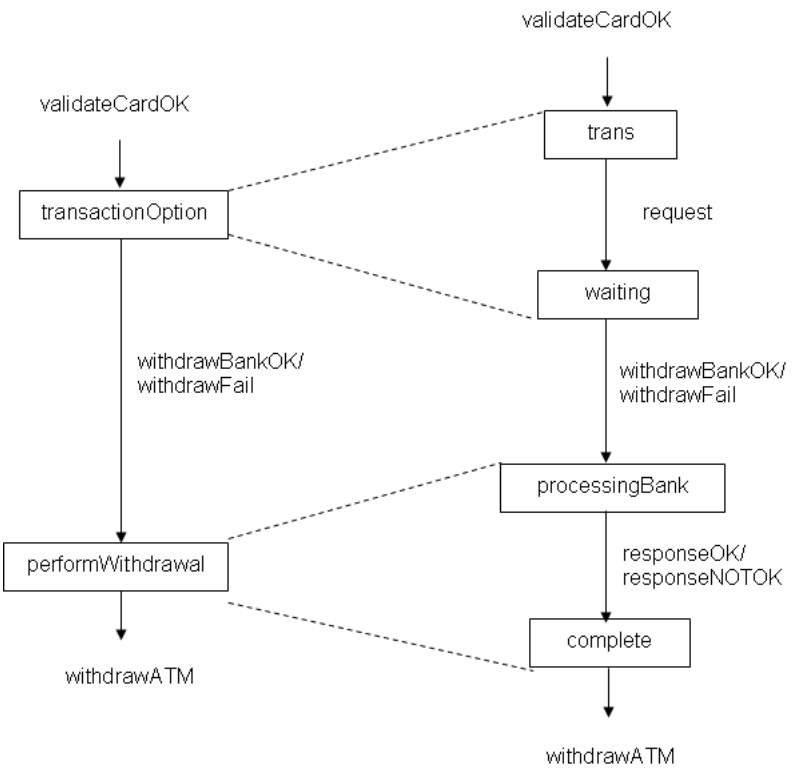


FIGURE 3.4: Sub-states of the States *transactionOption* and *performWithdrawal*

the ATM respectively. The sequence of the events is controlled by adding the ATM into the sets that represent states. The events are triggered by the state changes. The event *request* can be triggered when the ATM is in *trans* state and the ATM state is changed into the *waiting* state. The *withdrawBankOK* or *withdrawFail* event can then be triggered and the state is changed into *processingBank* state. When the request has been processed, the *responseOK/responseNOTOK* event can be triggered and the ATM state is moved into *complete* state.

3.3.4 Third Refinement

The third refinement refines the communication of the single system state into two separate states to distribute the control between ATMs and the bank. An ATM is responsible for most of the events including *request* and *response* events and the bank is responsible for processing the *withdrawBankOK* or *withdrawFail* event. An ATM is in the state *conversation* while the bank is processing (represented by state *bprocessing*) the withdrawal request. A new event *receiveRequest* is added in this refinement to model the bank receiving a request. The ATMs and the bank communicate via two new variables *req* and *rsp*.

The distributed states of the ATM and bank are shown in Figure 3.5. The *request* event

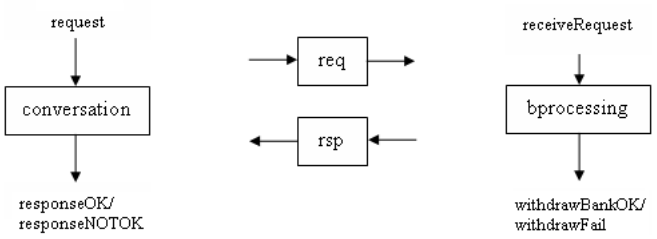


FIGURE 3.5: Distributed States

creates a new request and adds the request to the *req* variable. The *bprocessing* state can be entered when the *receiveRequest* event is triggered. When the request has been processed, a response is added to the *rsp* variable. When the response is in the *rsp* variable, the *response* event can be triggered and is received by the ATM.

3.3.5 Fourth Refinement

The fourth refinement introduces message passing as the form of communication. The messages sent within the system are modelled using a record style of specification using the syntax proposed in [34]. To model this record style, the context *ATMC_E1* is extended where a record type *MSG* is introduced to model the messaging medium between an ATM and bank as follows:

CONTEXT ATMC_E2
REFINES ATMC_E1
SETS
MSG::msg_atm:ATM (1)
END

The declaration (1) specifies a *MSG* record as having the field *msg_atm* of type *MSG* and links the messages with their corresponding ATMs. This syntax implicitly types the *msg_atm* as:

$$msg\_atm \in MSG \longrightarrow ATM$$

In this refinement level, two new variables, namely, *reqmsg* and *rspmsg* are introduced. These variables are of types *MSG* and they represent the two uni-directional channels for sending the request and the response messages respectively.

3.3.6 Fifth Refinement

The fifth refinement models the addition of information which needs to be sent during request to the bank and is received during response by the ATMs. The information includes the ATM card, the withdrawal amount, the status of transaction and the account balance. The information about the ATM card is send during both request and response events. The withdrawal amount is only sent during request event while the information about the transaction status and the account balance are sent only during response event. To model this additional information, an extension and specialisation to the record type *MSG* are done using the syntax proposed in [34] as follows:

CONTEXT ATMC_E3
REFINES ATMC_E2
SETS
EXTEND MSG **WITH** msg_card:$CARD$; (1)
REQ_MSG **SUBTYPES** MSG **WITH** $reqmsg\_wdAmount$:$\mathbb{N}$; (2)
RSP_MSG **SUBTYPES** MSG **WITH** $rspmsg\_status$:$STATUS$;$rspmsg\_bal$:$\mathbb{N}$ (3)
END

In the above declarations, the syntax (1) extends the *MSG* record as having another field, *msg_card* which is of type *CARD* and the record links messages and their cards. This syntax implicitly types *msg_card* as:

$$msg\_card \in MSG \longrightarrow CARD$$

The syntax (2) and (3) specialised *MSG* using subtyping. The syntax (2) declares *REQ_MSG* as a subtype of *MSG* to specify messages that are sent during request from the ATMs to the bank only. A record *REQ_MSG* has the field *reqmsg_wdAmount* which is of type $\mathbb{N}$. This field represents information about the withdrawal amount which is sent during request to the bank.

The syntax (3) declares *RSP_MSG* another subtype of *MSG* to specify messages that are received during response from the bank only. A record *RSP_MSG* has the field *rspmsg_status* which is of type *STATUS* and it contains the status information after the withdrawal request has been processed to indicate whether the withdrawal is successful or not. *STATUS* is an enumerated set with the elements OK and NOT_OK. The record RSP_MSG has another field *rspmsg_bal* which is of type $\mathbb{N}$ to represents the balance of account.

The fifth refinement refines the existing events related to processing the withdrawal request by adding more guards related to the additional fields to the *MSG* record.

3.3.7 Sixth Refinement

The sixth refinement models the communication as a single channel instead of two uni-directional channels. A new variable *msg* is introduced to represent the concrete communication channel. The abstract variables *reqmsg* and *rspmsg* are removed. These abstract variables represent the request and response messages. The gluing invariants are define as follows to differentiate between the request and response messages:

$$reqmsg = msg \cap REQ\_MSG \quad (1)$$
$$rspmsg = msg \cap RSP\_MSG \quad (2)$$

The invariant (1) says that in this refinement level, the messages in the variable *msg* which are of types *REQ_MSG* are the messages in the variable *reqmsg* of its abstract model. The invariant (2) says that in this refinement level, the messages in the variable *msg* which are of types *RSP_MSG* are the messages in the variable *rspmsg* of its abstract model.

Merging of the messages send and receive between ATM and bank is done to represent the actual implementation.

3.3.8 Experience modelling the ATM system in Event-B

This section gives an assessment of the experience of modelling using Event-B. Based on the experience, it is true, as stated by the guidelines in [16], that keeping a small difference between an abstract model and its refinement makes the proof of obligations easy to discharge.

Some of the invariants are constructed by using the provers in Rodin Event-B tool. One of them is the gluing invariant in the fifth refinement, that is *ATM_R5* in Appendix C. An attempt to construct the gluing invariant is done by using the interactive prover. The *ATM_R5* was run in a proving perspective without having any gluing invariant which result in three undischarge proof obligations. The interactive prover lists several hypotheses in order to achieve a goal. The hypotheses and the goal for one of those obligations are as follows:

Hypotheses:

$m \in reqmsg$
$msg\_atm(m)$=at
$msg\_card(m) = c$
$reqmsg\_wdAmount(m) = am$
$msg\_atm(m) \in conversation$
$msg\_card(m) \in cards$
$reqmsg\_wdAmount(m) \in \mathbb{N}$

The goal:

$$atm\_card(msg\_atm(m))=msg\_card(m)$$

From the hypotheses and the goal, it is deduced that a gluing invariant is needed to say that for all m where m is in *reqmsg*, the goal is implied. The goal only concerns with the parameter m. Therefore, from the list of hypotheses, the hypotheses $m \in reqmsg$ is selected forming the gluing invariant. The gluing invariant is represented in B as:

$$\forall\, m \cdot m \in reqmsg \Rightarrow atm\_card(msg\_atm(m))=msg\_card(m)$$

When this invariant is added in the model, the three previously undischarged proof obligations are proved without introducing any new proof obligations.

Another matter related to the ease of discharging a proof of obligation is using the disjoint sets representation rather that the state function representation in representing the states of the ATM system as described in Section 2.10.1. With the disjoint sets representation style, the constructed gluing invariants were simpler which makes the proof obligations easier to discharge. For example, in the third refinement, a gluing invariant is needed to specify that if an ATM is in the *bprocessing* state, then at the second refinement, the ATM must be in the *waiting* state. Using the disjoint sets representation, the gluing invariant is expressed simply as follows

$$bprocessing \subseteq waiting$$

With the state function representation, the gluing invariant is expressed as follows:

$$\forall\, a \cdot (a \in active\_atm \wedge rstatus(a)=bprocessing \Rightarrow astatus(a)=waiting)$$

where *rstatus* and *astatus* are functions from *active_atm* to a set of enumerate values. Clearly, the gluing invariant using the disjoint sets representation is simpler which helps to ease the proof effort.

All the proof obligations for all seven levels of refinements were generated and proved using Rodin provers. The statistics for all the refinement levels are outlined in Table 3.1. In the table, The *POs* column represents the total number of proof obligations generated for each level. The *inter POs* column represents the number of those proof obligations that had to be proved interactively. The *auto POs* column represents the number of those proof obligations that were proved automatically by the prover. Thus, we see that of almost 500 POs, most were discharged automatically using the prover.

LEVEL	POs	auto POs	inter POs
MM	4	4	0
R1	97	97	0
R2	82	82	0
R3	165	165	0
R4	61	60	1
R5	38	38	0
R6	50	50	0
Total	497	496	1

TABLE 3.1: Statistics from the Proof Effort

3.4 Event-B and UML-B Refinement

This section give an overview of refinement in Event-B and then based on that, a set of desirable requirements on UML-B are outlined.

In Event-B refinement, a machine that refines a more abstract machine may keep variables of an abstract machine, may drop some of the old variables and may introduce new variables. Also, a refinement machine may introduce new sets and constants. In a refinement machine, one or more events may refine an abstract event and may also introduce new events which refine *skip*. In Event-B refinement, invariants need to be devised to relate the refinement model and its abstract model. The Rodin interactive prover can help devise these invariants.

Based on refinement in Event-B we outline a set of desirable requirements for refinement in UML-B as follows:

1. Class refinement
   In order to reflect the refinement of variables in Event-B into UML-B refinement, class refinement is required. The keep and drop abstract variables and introduce new variables in Event-B refinement can be reflected in UML-B refinement by keeping the classes of an abstract machine, by dropping the abstract classes and introducing new classes. In addition, these Event-B refinements can be reflected in UML-B refinement by keeping the attributes of an abstract class, dropping the attributes and introducing new attributes.

2. State machine refinement
   State machine refinement is required in UML-B in order to reflect events refinement in Event-B. Events refinement in Event-B can be reflected in UML-B by having the transitions of a state machine refining corresponding abstract transitions of an abstract state machine. New events introduced in Event-B refinement can be reflected by introducing new transitions in UML-B refinement.

3. Classtype extension
   Classtype extension is required in UML-B refinement in order to reflect the additional constants of an extended context in Event-B where the constants are based on the sets that are introduced in the corresponding abstract context. Introducing new constants in Event-B refinement can be reflected in UML-B refinement by introducing new attributes.

We have outlined here a set of desirable requirements on UML-B. The next chapter (Chapter 4) explains the extensions made to the UML-B in order to meet these requirements.

3.5 Summary

This chapter presented the Event-B development of the ATM case study. The development provides understanding of modelling and refinement in Event-B. An overview of refinement in Event-B is provided in this chapter. A set of desirable requirements for refinement in UML-B are outlined based on the refinement in Event-B.

Chapter 4

Refinements in UML-B

4.1 Introduction

This chapter describes the extensions to the UML-B language and the translation scheme. An introduction to UML-B and the general translation of Event-B was given in Section 2.10. This chapter and Chapter 6 formed a proceeding [83] of the Formal Method 2009 international conference. First, the limitations with the previous UML-B are described. Then, we describe the notion of refined classes, refined state machines and extended classtypes. Several refinement techniques are also described with examples. The refinement techniques are listed below:

1. Add new attributes and associations to a refined class. This reflects adding variables in Event-B refinement in a pattern where the type of the variable is a relationship from a set of instances.
2. Add new classes in a refinement. This reflects adding variables in Event-B refinement in a pattern where the type of the variable is a set of instances.
3. State elaboration. This reflects adding variables and new events in Event-B refinement.
4. Transition elaboration. This reflects event refinement in Event-B.
5. Move event or transition to a refined class or a new class in a refinement. This reflects event refinement in Event-B.
6. Add new attributes and associations to an extended classtype. This reflects adding constants in Event-B refinement in a pattern where the type of the constants is a relationship from a set of instances.
7. Add new classtypes in a refinement. This reflects adding sets in Event-B refinement.

The techniques (1) and (2) are described in Section 4.3. The techniques (3) and (4) are described in Section 4.4 whereas technique (5) is described in Section 4.5. The techniques (6) and (7) are described in Section 4.6.

4.2 Limitations of the previous UML-B

Let's refer to the version of UML-B before the extensions as UML-B Version 1 and after the extension as UML-B Version 2. The UML-B Version 1 has limitations with modelling refinement. The limitations arise when refining classes, state machines and extending classtypes. The explanation of these limitations are described in the following subsections.

4.2.1 Limitations in Refining Classes

In UML-B Version 1, the requirement for class refinement as in Section 3.4 was done by having copies of classes in a refinement machine. The U2B translator generates Event-B implicit contexts and machines from classes in UML-B. The limitations when refining classes are as follows:

- In the generated Event-B implicit contexts, the carrier sets introduced in a context are repeated in its extended context causing static checker errors.
- In the generated Event-B machines, the type invariants which have been devised in a machine are repeated in its refinement machine. This does not cause any error but it is unnecessary complication.

For example, Figure 4.1(a) shows the machine *M1* is refined by machine *M2*. *M1* and *M2* have a class C with an attribute x of type natural number as in Figure 4.1(b). Each UML-B machine gives rise to both an implicit Event-B context and an Event-B machine. Figure 4.1(c) shows the generated Event-B implicit contexts *M1_ImplicitContext* and *M2_ImplicitContext*. For the given UML-B models, there is an error indicating there exist ambiguity of the carrier set *C_SET* in the refinement implicit context *M2_ImplicitContext*. This error is shown at the bottom of Figure 4.1(c). In the generated Event-B machine in Figure 4.2, the type invariants labelled $C.type$ and $x.type$ generated in *M1* are repeated in the refinement machine *M2*.

4.2.2 Limitations in Refining State Machines

In UML-B Version 1, the state machine refinement requirement as in Section 3.4 was done by having a copy of a state machine in a refinement machine. Then, the copied state

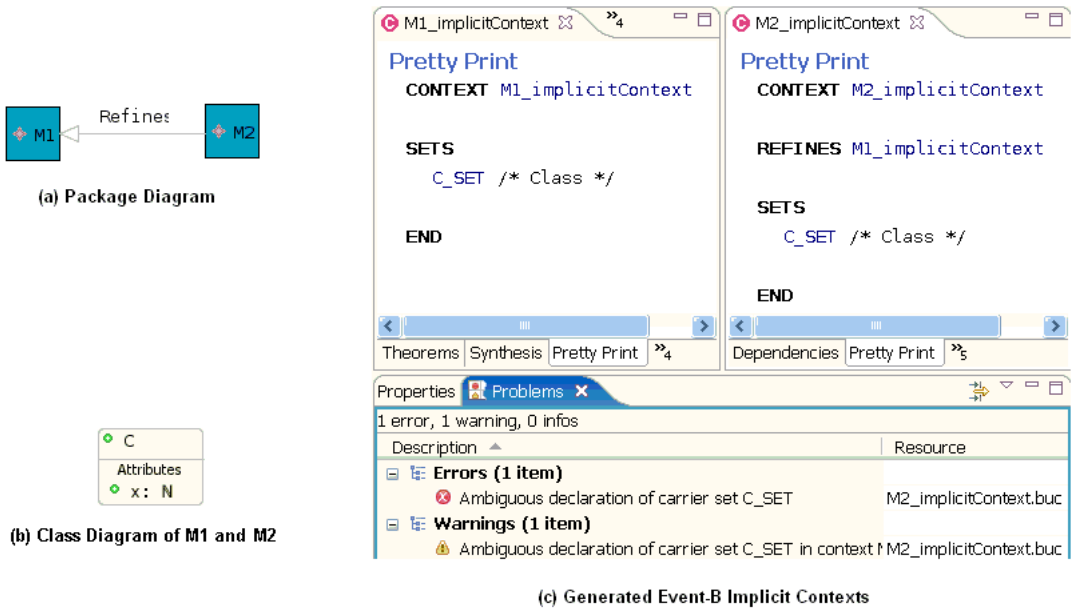


FIGURE 4.1: Package Diagram, Class Diagram and Generated Implicit Contexts

machine may be refined by refining its states into sub-states. This is done by adding a nested state machine to a state forming a hierarchy of state machines. UML-B Version 1 allows modelling nested state machine however refinement is not well supported. The limitations when refining state machines are as follows:

- For a disjoint set translation, the type invariants and disjointness invariants generated from the states are repeated in a generated Event-B refinement machine.
- For a state function translation, the carrier sets generated from the state machine and the constants generated from the states are repeated in the generated Event-B implicit context.
- The initial transition (transition with an initial state) and the final transition (transition with a final state) of a nested state machine were not properly supported since they could not be associated with the parent transitions i.e., the incoming and outgoing transitions of the super-state.

We give here an example of state machine refinement. Assume that the state machine *Csm* in Figure 4.3(a) is attached to the class *C* of *M1*. The state machine *Csm* in Figure 4.3(b) refines the state machine in Figure 4.3(a) where the state *B* is added a state machine *Bsm* that consists of three sub-states namely *B1*, *B2* and *B3* (Figure 4.3(c)). Figure 4.4 shows the generated Event-B invariants for *M1* and *M2* using the disjoint sets representation. The invariants labelled *A.type*, *B.type* and *disjointState B,A* in machine *M1* should not be repeated in the refinement machine *M2*. This is one of the limitations when performing state machine refinement. The other limitation is duplication of events

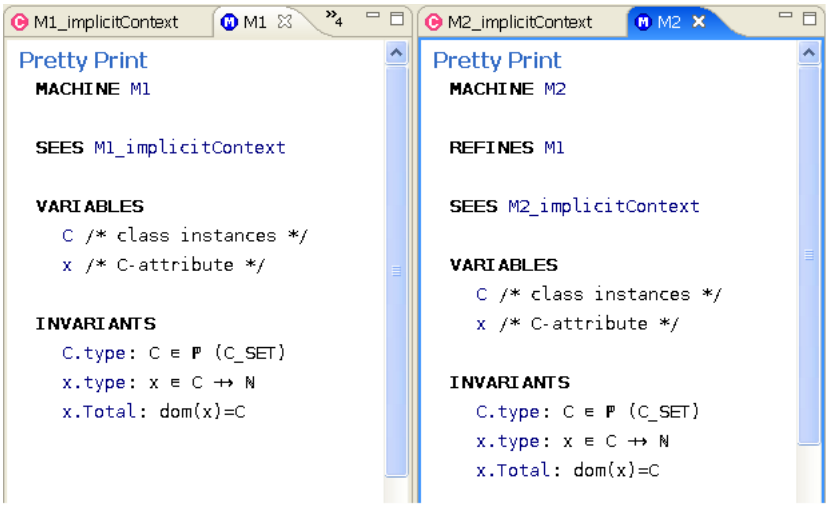


FIGURE 4.2: Generated Event-B Machines

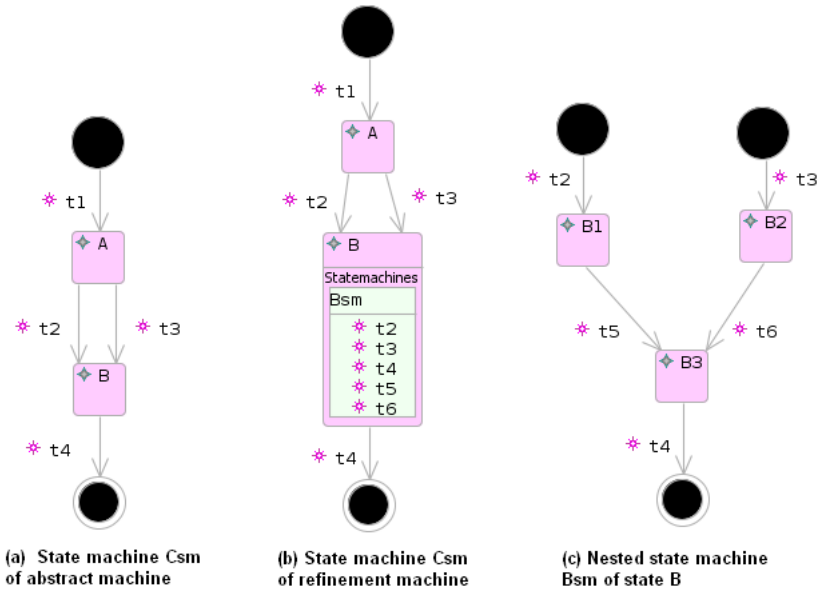


FIGURE 4.3: Example of State Machine Refinement

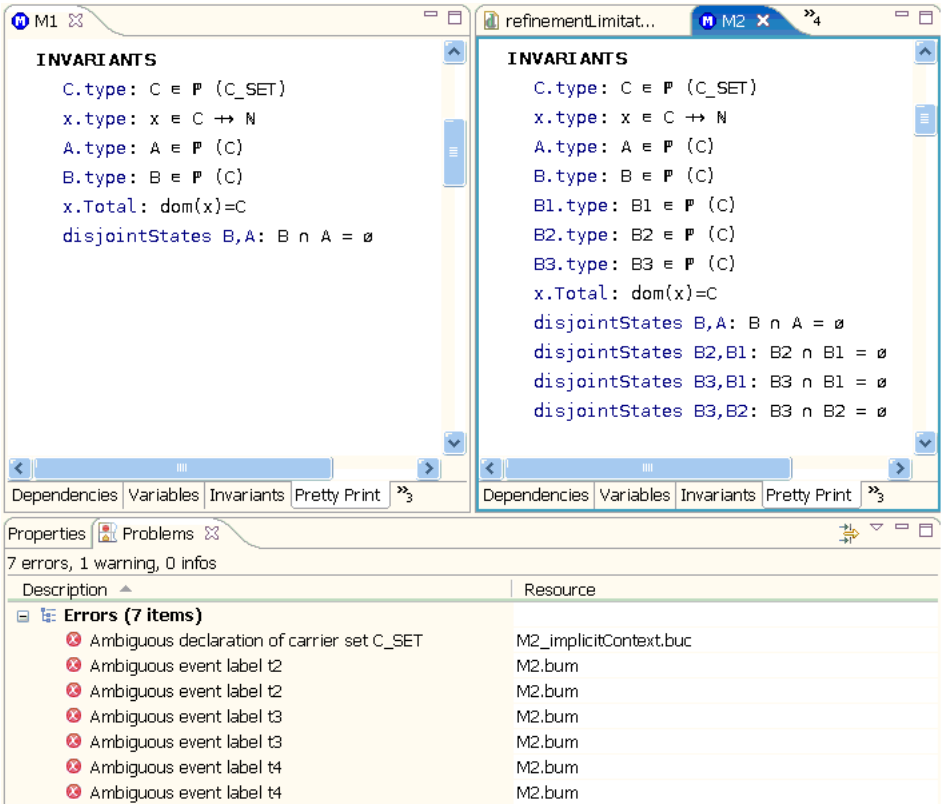


FIGURE 4.4: Generated Event-B Machine from UML-B State Machines

generated in a refinement machine. For example the events *t2*, *t3* and *t4* are duplicated in machine *M2*. Errors of event duplication, i.e., ambiguous event label are shown at the bottom of the Figure 4.4. These duplicated events are generated from the incoming and outgoing transitions of the state *B* in Figure 4.3(b) and also from the transitions with an initial source states i.e., the transitions *t2* and *t3* and the transition with a final target state i.e., the transition *t4* in the nested state machine *Bsm*. The errors are reported because the modeller attempts to associate the initial and final transitions of the nested state machine with the incoming and outgoing transitions of the super-state by giving them the same name. What is required in the translation is that only one each of *t2*, *t3* and *t4* events are generated. It means, we want to be able to specify in UML-B model that *t2* in Figure 4.3(c) is the same as *t2* in (b). Similarly for *t3* and *t4*. But this cannot be specified in UML-B Version 1.

4.2.3 Limitations in Extending Classtypes

In UML-B Version 1, the classtype extension as in Section 3.4 was done by having a copy of classtypes in an extended context. The U2B translator generates Event-B contexts from classtypes in UML-B. The limitations when extending classtypes are as follows:

- The carrier sets and constants introduced in a context are repeated in its extension

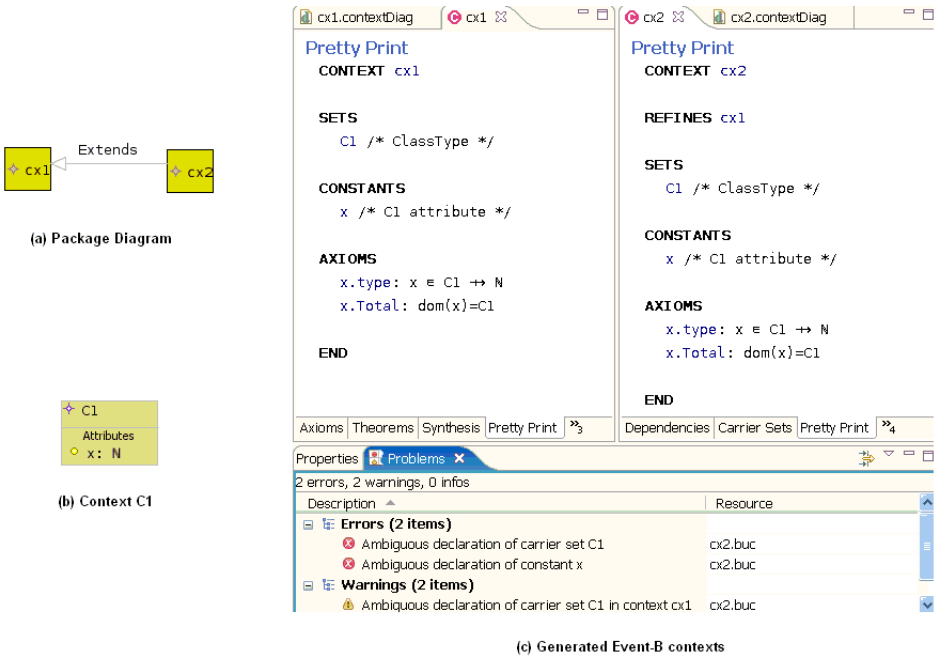


FIGURE 4.5: Package Diagram, Context Diagram and Generated Contexts

context causing errors to be reported by the static checker.

- The type axioms which have been generated in a context are repeated in its extension context. This does not cause any error but it is unnecessary complication.

For example, Figure 4.5(a) shows the context *cx1* is extended by context *cx2*. *cx1* and *cx2* has a classtype *C1* with an attribute x of type natural number as in Figure 4.5(b). Figure 4.5(c) shows the generated Event-B contexts *cx1* and *cx2*. For the given UML-B models, there are errors indicating there exist ambiguity of carrier set *C1* and constant x in the extension context *cx2*. These errors are shown at the bottom of Figure 4.5(c). In the generated Event-B contexts in Figure 4.5(c), the axioms labelled *x.type* and *x.total* generated in *cx1* are repeated in the extension context *cx2*.

Part of our work involved extending UML-B in order to overcome the limitations of class refinement, state machine refinement and classtype extension outlined here. Class refinement is described in Section 4.3, state machine refinement is described in Section 4.4 and classtype extension is described in Section 4.6.

4.3 Refinement of Classes in UML-B

In this section, the refinement techniques concerning the notion of refined classes and inherited attributes are described.

The motivation for refined classes and inherited attributes comes from performing refinement in Event-B. The notion of refined classes and inherited attributes in UML-B reflects the refinement of variables in Event-B. A refined class is one that refines a more abstract class and an inherited attribute is one that inherits an attribute of the abstract class. A notion of refined classes is needed in UML-B because some elements of an abstract UML-B model need to be retained by the refinement.

In Event-B refinement, a machine that refines a more abstract machine may keep variables of an abstract machine, may drop some of the old variables and may introduce new variables. In UML-B refinement, a machine that refines a more abstract machine may contain refined classes where each refined class refines a class of its abstract machine (i.e., keeps variables of its abstract machine). In UML-B refinement, a machine may drop some refined classes (i.e., drop some variables). Also in UML-B refinement, a machine may introduce new classes (i.e., new variables) in a class diagram. Hence, a refined class indicates that the variable representing the set of instances of the class will be retained in the refinement. However, it has a further use since it provides a mechanism for the modeller to indicate the refinement features that were (or are) based on that variable.

In UML-B refinement, a refined class may inherit attributes of its abstract class (i.e., keeps variables of its abstract machine). A refined class may drop some of the attributes of its abstract class (i.e., drop some variables of its abstract machine) and a refined class may introduce new attributes (i.e., new variables). The following schematic table illustrates a refined class that inherits and drops abstract attributes and introduces new attributes. The table lists out the attributes for class *C* and a refined class *C*. Class *C* contains attributes *a1*, *a2* and *a3*. In refinement, the refined class *C* inherits attributes *a1* and *a2*, drops attribute *a3* and has new attributes *a4* and *a5*. In the generated Event-B machine, both a class and a refined class give rise to variables. A type invariant is generated for an abstract class i.e., Class *C* but not for a refined class because its type is already defined in the abstract Event-B machine. Similarly both the inherited attributes and new attributes give rise to variables and a type invariant is generated for each new attributes but not for the inherited attributes.

Class C	**Refined Class C**
a1	a1 (*inherited*)
a2	a2 (*inherited*)
a3	a4 (*new*)
	a5 (*new*)

We describe here an example of performing refinement in UML-B using the notion of refined classes and inherited attributes. Consider a refinement where machine *M2* refines machine *M1*.

Figure 4.6 shows an example of a package diagram (a) that contains machine *M1* which has a class diagram(b) containing classes *CA* and *CB*. These classes give rise to the

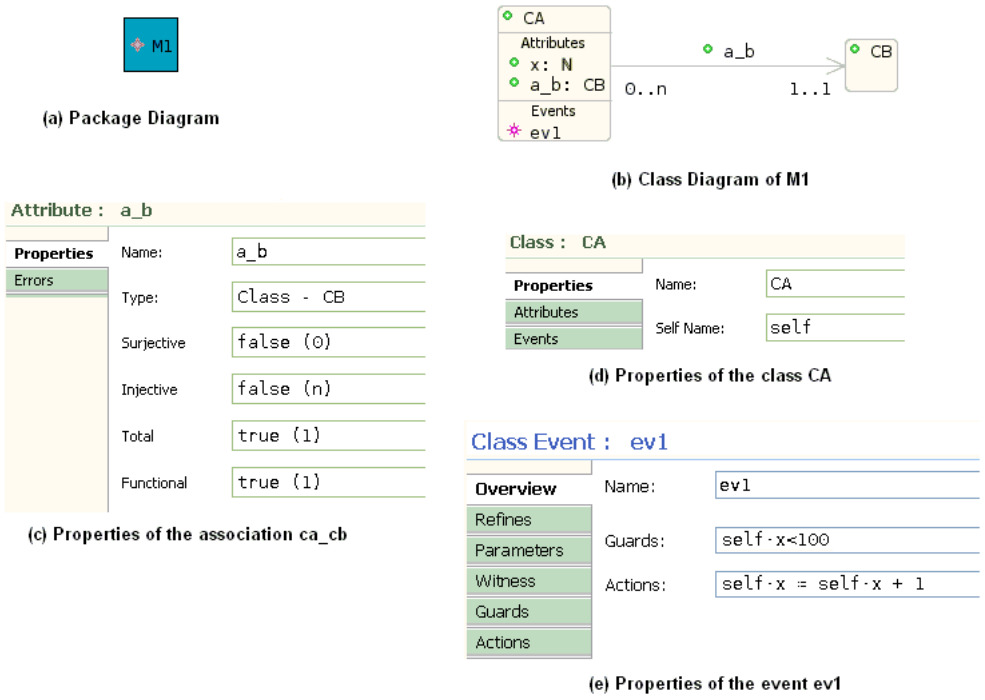


FIGURE 4.6: Package Diagram and UML-B Specification of Machine M1

sets *CA_SET* and *CB_SET* in the generated Event-B implicit context. In the generated Event-B machine the classes *CA* and *CB* give rise to variables. The class *CA* consists of the attribute x of type $\mathbb{N}$ and also the association *a_b* of type *CB*. The multiplicity property for the association *a_b* shown in Figure 4.6(c) specifies a many-to-one relationship (i.e., total function). The attributes x and *a_b* also represent variables in the generated Event-B machine. For each class, attribute and association, a type invariant will be generated in the Event-B machine. For example, the class *CA* corresponds to the type invariant which specifies that *CA* is a subset of *CA_SET* ($CA \in \mathbb{P}(CA\_SET)$). Attribute x corresponds to the type invariant $x \in CA \rightarrow \mathbb{N}$ that specifies x is defined for all *CA*. The self name property of the class *CA* is shown in Figure 4.6(d). The class *CA* has an event *ev1*. The event is executed when x is less than 100 and its action is to increase x by 1. The guard and action of the event is shown in Figure 4.6(e).

Figure 4.7(a) shows an example of a package diagram that manages a refinement relationship between machines. The package diagram shows that machine *M2* refines machine *M1*. The class diagram of *M2* is shown in Figure 4.7(b) where it consists of refined classes *CA* and *CB* which refine the classes *CA* and *CB* of machine *M1* respectively. The refined class *CA* of machine *M2* inherits attribute x and association *a_b* of the class *CA* of machine *M1*. The refined class *CB* of machine *M2* has a new association *cb_cc*. Machine *M2* has a new class, *CC* which corresponds to a new set (*CC_SET*) in

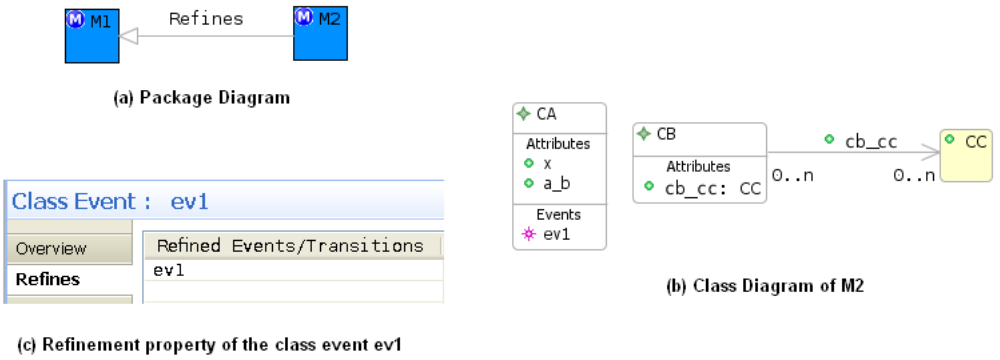


FIGURE 4.7: Package Diagram and Class Diagram of Machine M2

the generated Event-B implicit context (which extends the previous implicit context). In the generated Event-B machine for machine *M2*, the variables *CA*, *CB*, *x* and *a_b* are retained. The machine *M2* has new variables *CC* and *cb_cc* with their type invariants $CC \in \mathbb{P}(CC\_SET)$ and $cb\_cc \in CB \leftrightarrow CC$ respectively. The refined class *CA* has an event *ev1* which refines the abstract event *ev1* of machine *M1*. The refinement property is shown in Figure 4.7(c).

In Event-B refinement, a machine that refines another machine (i.e., abstract machine) must provide a refinement of each abstract event. This can be either that one event refines one abstract event, or many events refine one abstract event. New events may be introduced in the refinement. Similarly, in UML-B refinement, at least one concrete event must refine each abstract event and new events may be introduced. These concrete events can either be attached to refined classes or state machines of a refined classes. In UML-B refinement, we can also define additional invariants and theorems by attaching them to refined classes and states that reflect adding invariants and theorems in Event-B refinement.

4.4 Refinement of State Machines in UML-B

In this section, the refinement techniques concerning the notion of refined state machines and refined states are described.

The motivation for refined state machines and refined states come from combining the state machine hierarchy in UML-B with refinement in Event-B. The essential concept is that state machines are refined by elaborating an abstract state with nested sub-states. A refined state machine is one that refines a more abstract state machine and a refined state is one that refines a more abstract state.

In UML-B refinement, a refined machine may contain refined state machines and refined states of its abstract machine. We describe first an example of performing refinement

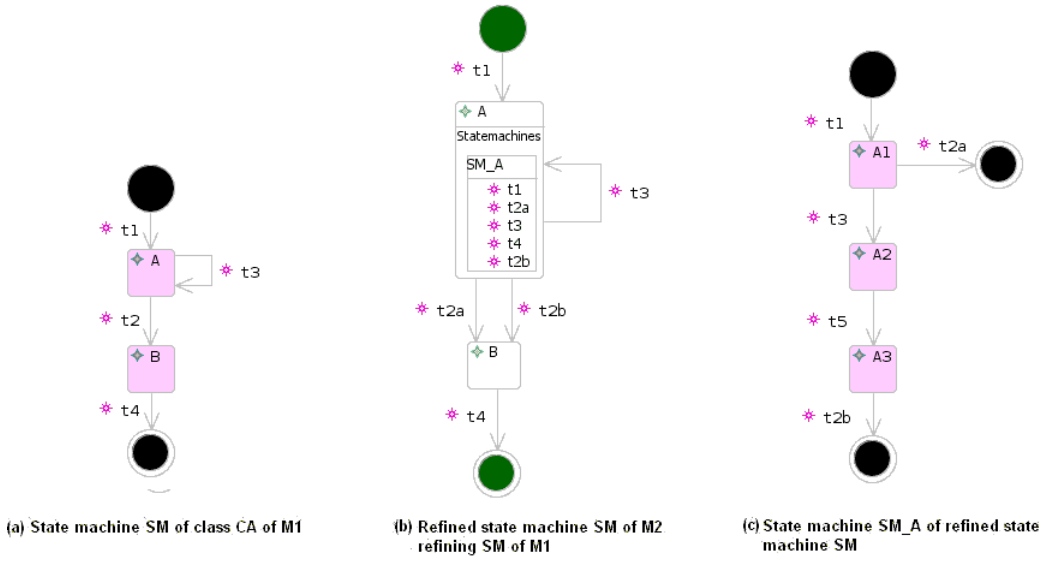


FIGURE 4.8: Refinement of State Machine (Machine M2 refines Machine M1)

in UML-B using the notion of refined state machines and refined states. We will then describe the general rules. Figure 4.8 shows an example of refinement of a state machine. Assume that the state machine SM of Figure 4.8(a) belongs to a class *CA* of machine *M1*. Similarly, assume that the state machine of Figure 4.8(b) belongs to refined class *CA* of machine *M2*. The state machine 4.8(b) refines that of 4.8(a). The states of refined state machine *SM* are the state *A*, that refines state *A* of machine *M1* and the state *B*, that refines state *B* of machine *M1*. The refined state machine *SM* contains the transitions *t1*, *t2a*, *t2b*, *t3* and *t4* which refine the corresponding abstract transitions of machine *M1*. In Figure 4.8(b), the abstract transition *t2* is replaced with transitions *t2a* and *t2b* which refine the abstract transition *t2* of machine *M1*. This refinement of transition *t*2 reflects the refinement in Event-B where many events refine one abstract event. The transitions *t2a* and *t2b* have different source sub-states (i.e., representing different guards in Event-B) which are defined in the nested state machine *SM_A*.

The nested state machine *SM_A* (Figure 4.8(c)) elaborates the refined state A (Figure 4.8(b)) of machine *M2*. The nested state machine, *SM_A* has three states *A1*, *A2* and *A3*. The transition *t1* of the nested state machine *SM_A* in Figure 4.8(c) elaborates the incoming transition *t1* of the refined super-state *A*. It means, in the refinement, the target state of the transition *t1* is the state *A1*. The transitions *t2a* and *t2b* of the nested state machine *SM_A* elaborate the outgoing transition *t2a* and *t2b* of the refined super-state *A*. In Figure 4.8(b) we do not see a distinction between transitions *t2a* and *t2b*. In Figure 4.8(c) we can see a distinction: *t2a* has sub-state *A1* as a source while *t2b* has *A3* as source. The transition *t3* of the nested state machine *SM_A* elaborates the self loop transition of the refined super-state *A* specifying its source state as the state *A1* and its target state as *A2*. In the nested state machine *SM_A*, the transition *t5* is a

new transition that represents a new event in the generated Event-B machine.

In the generated Event-B machine using a disjoint sets representation, type invariants are created for all sub-states, where their types are their super-state, for example $A1 \in \mathbb{P}(A)$ is a type invariant for the state *A1*. An additional invariant is generated to specify that all sub-states constitute their super-state. For example, $A = A1 \cup A2 \cup A3$. Other generated invariants are a number of disjointness invariants that specified all sub-states are disjoint.

In the next successive paragraphs, we give a general definition of state machine refinement based on the example given above. A refined state machine refines a more abstract state machine. The structure of a refined state machine is an elaboration of the structure of its abstraction in two possible ways:

- Each transition is replaced by one or more transitions.
- An abstract state may be elaborated by a nested state machine (see below).

In the given example, we used the techniques of state elaboration and transition elaboration. In UML-B refinement, a refined state may be elaborated to sub-states which are contained in a nested state machine forming a state machine hierarchy. State elaboration enables more transitions to be added to a nested state machine. Some of these transitions elaborate the incoming and outgoing transitions of the super-state (i.e., the abstract state). Some of these transitions are new transitions (i.e, reflects introducing new events in Event-B).

In UML-B, nested state machines are modelled in a separate state machine diagram from their parent state machine diagram. Therefore, the transition elaboration technique is needed so that transitions in a nested state machine can elaborate the incoming and outgoing transitions of the super-state. In a nested state machine, a transition with an initial source state elaborates at most one incoming transition to the super-state and a transition with a final target state elaborates at most one outgoing transition from the super-state. The restriction of the elaborate property is made to at most one so that it is explicit on the state machine diagram which sub-transitions are elaborating the respective parent transitions.

In future, we will explore a case where the same transition may occur when an object is in any sub-state. Let us refer to the example in Figure 4.8 to describe this case. Using UML-B Version 2, this case can be model by replacing the abstract parent transition $t2$ with three parent transitions $t2a$, $t2b$ and $t2c$ in the refined state machine (Figure 4.8(b)). The abstract transition will be replaced by three transitions because there are three sub-states in the nested state machine (Figure 4.8(c)). In the nested state machine, each sub-state will have an outgoing transition (to a final target state). Each outgoing transition will elaborate a different parent transition. However, this may clutter the

state machine diagram as there are too many transitions. An alternative way is to keep the parent transition $t2$ in the refinement and not model the outgoing transitions from all sub-states. This means it is implicit that the same transition may occur when an object is in any of the sub-states.

An abstract state may have a self loop transition. In UML-B refinement, when the state is elaborated into sub-states, the self loop transition may be elaborated as one of the sub-transitions between any two of the sub-states. The sub-transition defines the state changes from a sub-state to another sub-state when the transition fires. When refining a self loop transition, the occurrence of the transition can either be many times or can be restricted to once. Restriction to once means removing looping behaviour and this is a valid refinement since we focus on preserving safety, not liveness, in our work.

An alternative way of modelling a nested state machine is to have a nested state machine in the same editor with the parent diagram. An example of this is illustrated in Figure 4.9 which models an alternative of Figure 4.8(b) and (c). However we prefer not to use this alternative because it will result in scalability problems when there are many levels of state machine hierarchy and a nested state machine has many states. The advantage of this alternative way is that it does not need the transition elaboration technique which was described previously. Whereas when using different editors for nested state machine and its parent state machine, the transition elaboration technique is essential.

The general definitions described here of the refined classes and refined state machines are made more precise in Chapter 5 where we define the extension to the UML-B meta-model.

4.5 Event/Transition Movement

This section describes the technique of moving class events or state machine transitions in UML-B refinement. There are two methods of moving a class event or a state machine transition in a refinement. The methods are as follows:

1. Move a class event to a refined class as a state machine transition or, move a state machine transition to a refined class as a class event.
2. Move a class event to a new class in a refinement either as a class event or a state machine transition or, move a state machine transition to a new class in a refinement either as a class event or a state machine transition.

Method (1) does not need any new UML-B language feature. However, method (2) creates a motivation for the need to be able to change the default self name in UML-B.

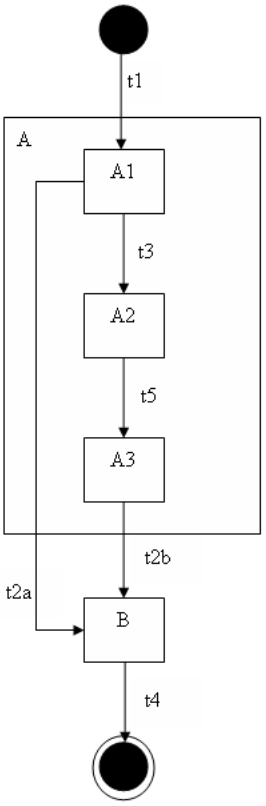


FIGURE 4.9: An Alternative of Modelling a Nested State Machine

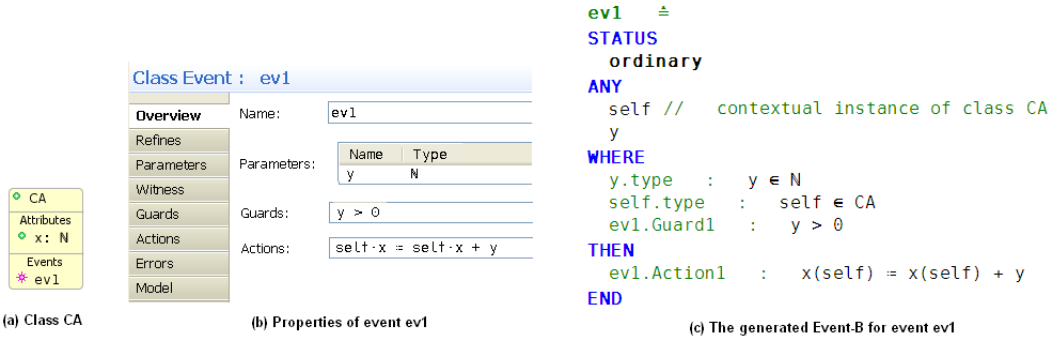


FIGURE 4.10: Example of the UML-B Specification with an Event in a Class CA of an Abstract Machine

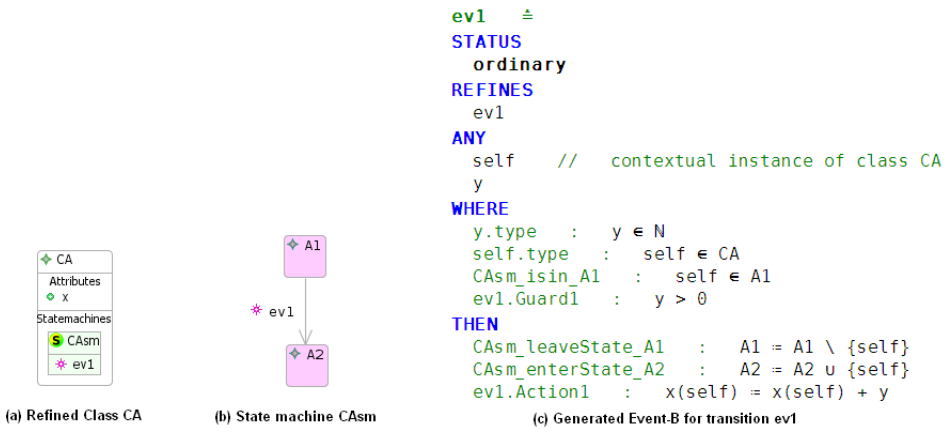


FIGURE 4.11: Example of the UML-B Refinement for the First Method

We describe both methods by giving first an example of an abstract machine upon which a refinement is based. Figure 4.10(a) shows a class *CA* with attribute x and event *ev1*. Figure 4.10(b) shows the properties of the event *ev1* showing its parameter y, a guard and an action. The action is defined using μB notation and uses a default identifier *self* i.e., the self name property which represents an instance of a class *CA*. The self name property becomes a parameter in the *e1* event in the corresponding Event-B machine. The generated Event-B specification for the event *ev1* is shown in the Figure 4.10(c).

Figure 4.11 is an example of a refinement for the first method of event movement. In the refinement, the event of the class *CA* is moved as a transition *ev1* between the states *A1* and *A2* of the state machine *CAsm* (Figure 4.11(b)) belong to the refined class *CA* (Figure 4.11(a)). The generated Event-B specification for the transition *ev1* is in Figure 4.11(c) showing additional guard specifying the current state *A1* for the transition to take place ($self \in A1$) and also additional actions specifying an instance move to the state *A2* ($A2 = A2 \cup \{self\}$) and an action specifying an instance leaves the current state *A1* ($A1 = A1 \setminus \{self\}$). The effect of this refinement is to constrain when the event can occur.

For the second method, in a refinement, a class event may be moved to a new class as a class event or as a transition in a state machine. We describe here the event movement technique when a class event is moved to a new class as a transition in a state machine. Assume that the UML-B specification in Figure 4.12 is a refinement of the abstract machine in Figure 4.10. In the refinement, a new class *CC* is introduced and the event *ev1* is moved to the class CC (Figure 4.12(a)). The event *ev1* becomes a transition between the two states *C1* and *C2* of state machine *CC_sm* (Figure 4.12(b)). In the refinement machine, a parameter, *ca*, of type *CA* is added to the *ev1* transition as shown in the properties view in the Figure 4.12(c). Also, a witness property is defined for the event *e1* which specified that *ca* in the refinement represents *self* of its abstract level (i.e., $ca = self$).

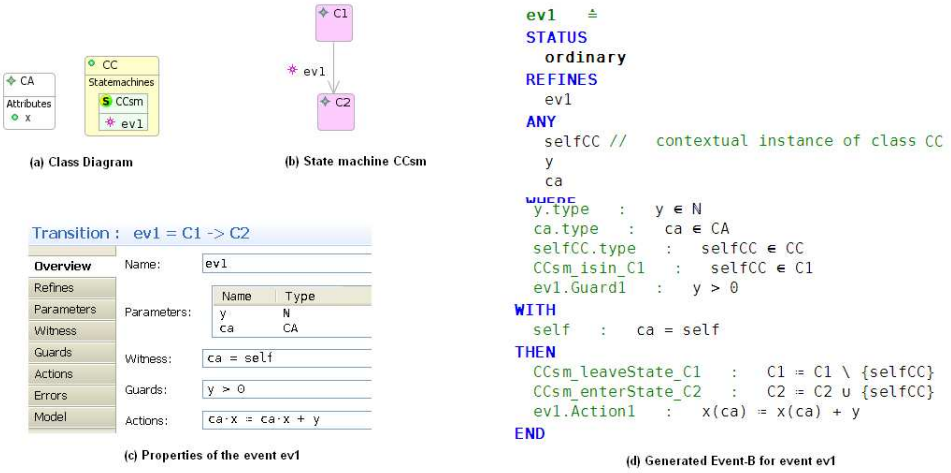


FIGURE 4.12: Example of the UML-B Refinement for the Second Method

The witness property is adapted from Event-B. In Event-B, a witness is used when replacing a parameter of an abstract event with a different parameter in a concrete event in the refinement. The witness is defined by a predicate involving the abstract parameter.

Figure 4.12(d) shows the generated Event-B specification for the event *ev1*. The parameter *selfCC* is generated from the self name property specified for the class *CC*. The self name property of the class *CC* is changed to *selfCC* from a default *self* to avoid conflicts with the default self name property of the refined class *CA* which is used in the witness. Figure 4.12(d) also shows an additional guard specifying the current state *C1* for the transition to take place and also additional actions specifying an instance move to the state *C2* and an action specifying an instance leaves the current state *C1*.

For both methods, the movement of state machine transitions as class events (or as state machine transitions for the second method) is also possible and the techniques are similar to the above examples.

Section 6.4 of the ATM case study will demonstrate the usefulness of moving events from one class to another in refinement. In the first refinement of the ATM case study, the *withdraw* event of the *Account* class is moved to the *ATM* class as a transition in a state machine of the class *ATM*. Section 8.7.3 will demonstrate the usefulness of moving transitions as refined class events in refinement. In the second refinement of the middleware component, the *request* and *response* transitions are moved as the *msg* refined class events.

4.6 Extension of Classtypes in UML-B

In this section, the techniques concerning the notion of extended classtypes are described.

The motivation for extended classes comes from performing record extension in Event-B. An extended classtype is one that extends a more abstract classtype. A notion of extended classtypes is needed in UML-B because some classtypes of an abstract UML-B context need to be retained by the extension when new attributes are required to be added to the classtypes.

In Event-B context extension, a context that extends a more abstract context will keep the sets and constants of an abstract context and may introduce new sets and constants. In UML-B context extension, a context that extends a more abstract contexts may also contain extended classtypes where each extended classtype extends a classtype of its abstract context. In UML-B context extension, a context may introduce new classtypes (i.e., new sets) with its attributes (i.e., new constants based on the sets) in a context diagram. Also, new attributes may be introduced in an extended classtype in UML-B context extension.

We describe here an example of performing context extension in UML-B using the notion of extended classtypes. Figure 4.13(a) shows an example of a package diagram that shows context *cx1* is extended by *cx2*. Figure 4.13(b) shows the context diagram of *cx1* that contains a classtype *C1* with attribute *x*. This classtype give rise to the sets *C1* in the generated Event-B context. Attribute *x* of *C1* give rise to constant *x* in the generated Event-B context. For attribute of a classtype, a type axiom is generated in Event-B. For example, the axiom $x \in C1 \rightarrow \mathbb{N}$ is generated for attribute *x*. Figure 4.13(c) shows context diagram of *cx2* that contains the extended classtype *C1* and new classtype *C2*. A new attribute *y* of type BOOL is added to the extended classtype *C1*. Classtype *C2* has an association *c1c2* with the extended classtype *C1*. Similar to associations in class diagram, associations in context diagram are special case attributes. The attributes *y* and *c1c2* give rise to constants *y* and *c1c2* in the generated Event-B context. In addition, the axioms $y \in C1 \rightarrow BOOL$ and $c1c2 \in C2 \rightarrow C1$ are generated from the attributes *y* and *c1c2* respectively.

4.7 Summary

This chapter introduced the notions of refined class, refined state machine and extended classtype for UML-B. These notions are used to describe the following seven refinement techniques:

1. Add new attributes and associations to a refined class.

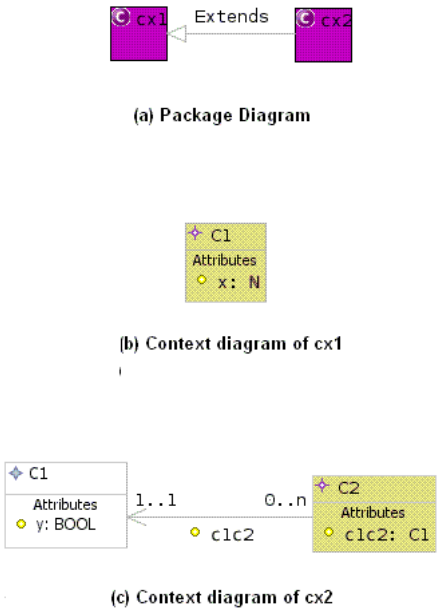


FIGURE 4.13: Package Diagram and Context Diagram

2. Add new classes in a refinement.
3. State elaboration.
4. Transition elaboration.
5. Move event or transition to a refined class or a new class in a refinement.
6. Add new attributes and associations to an extended classtype.
7. Add new classtypes in a refinement.

The UML-B metamodel has been extended to support these new techniques. Chapter 5 describes the extensions to the UML-B metamodel.

Chapter 5

UML-B Metamodel Extension

5.1 Introduction

This chapter contains a description of the extensions to the UML-B Version 1 metamodel and the implementation of the extensions. The description of the extensions is divided into four sections. Section 5.2 describes the metamodel extensions corresponding to the notion of refined classes in Section 4.3. Section 5.3 describes the metamodel extensions corresponding to the notion of refined state machines in Section 4.4. Section 5.4 describes the metamodel extensions corresponding to the notion of extended classtypes in Section 4.6. Section 5.5 describes the extensions to the UML-B drawing tools which use the metamodel extensions. These tasks were done in order to support refinement in UML-B.

5.2 Extending the UML-B Metamodel and Tool to Support Class Refinement

The limitations of refining a class in the UML-B Version 1 explained in Section 4.2.1, occur because the tool treats the refined class as another new class. This is because there are no metaclasses in the UML-B Version 1 metamodel which explicitly represent refined classes and inherited attributes.

Changes were made to the UML-B Version 1 metamodel described in Section 2.10.2. The changes to the metamodel in Figure 2.20 have been made to support class refinement. The changes involve additional classes, some changes in attributes of the existing metaclass and also changes to some of the association between existing and new classes.

In UML-B development, a class diagram consists of a number of classes where each class contains attributes. A class diagram of a refinement machine may contain classes

from its abstract class diagram. These classes should appear as refined classes that inherit attributes of their abstract classes. The extension of the UML-B metamodel involves adding a metaclass which represents these refined classes and a metaclass which represents these inherited attributes.

There are seven new metaclasses being introduced in the extensions which are *UMLBabstractClass*, *UMLBRefinedClass*, *UMLBabstractAttribute*, *UMLBInheritedAttribute*, *UMLBnamedElement*, *UMLBnameConstrainedElement* and *UMLBname*. The first four metaclasses are for the purpose of realizing the notion of refined classes. The other three metaclasses are for naming elements which will be described later.

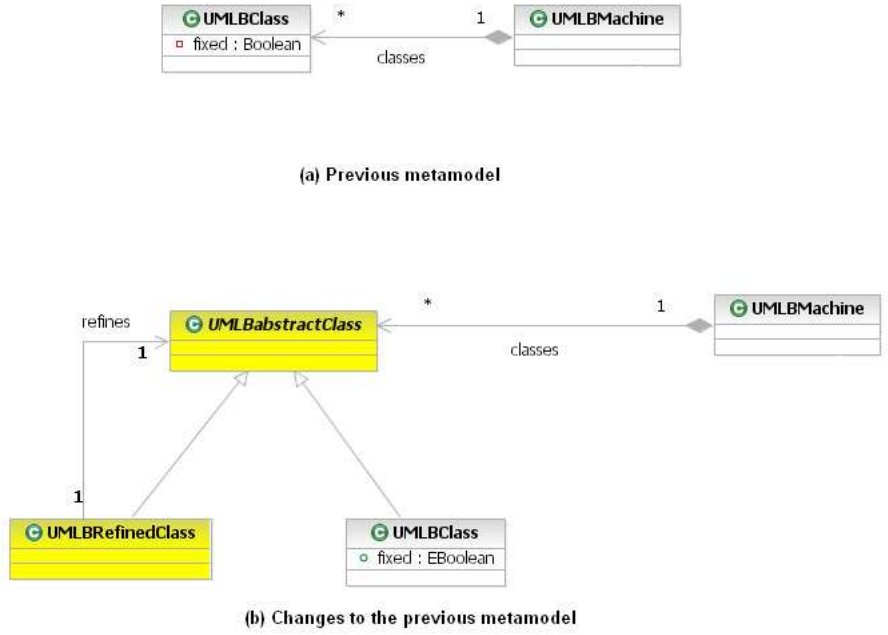


FIGURE 5.1: UML-B Metamodel Extensions for Classes

Figure 5.1 shows the changes to the metamodel corresponding to the notion of refined classes where Figure 5.1(a) is part of the UML-B Version 1 metamodel which specified that a UML-B machine may have a number of classes. We extend this part of the metamodel by replacing Figure 5.1(a) with Figure 5.1(b) that adds two new classes (highlighted by yellow colour). The metaclass *UMLBabstractClass* is an abstract metaclass which allows its common properties to be shared by its sub-metaclasses, i.e., *UMLBRefinedClass* and *UMLBClass* (existing class). The metaclass *UMLBabstractClass* does not represent any UML-B modelling object. *UMLBRefinedClass* represents a refined class whereas *UMLBClass* represents a class. Figure 5.1(b) specifies that a UML-B refinement machine may have a number of classes and/or refined classes. This corresponds to Event-B refinement where a refinement machine may have a number of new variables and/or old variables from its abstract machine. The one-to-one *refines* association be-

tween *UMLBRefinedClass* and *UMLBabstractClass* specifies that a refined class may refine exactly one class or refined class.

Figure 5.2 shows the changes to the metamodel corresponding to the notion of inherited attributes where Figure 5.2(a) is part of the UML-B Version 1 metamodel which specified that a UML-B class may has a number of attributes. We extend this part of the metamodel by replacing Figure 5.2(a) with Figure 5.2(b) that consists of two new classes. *UMLBabstractAttribute* is an abstract metaclass which does not represent any UML-B modelling object. *UMLBInheritedAttribute* represents an inherited attribute whereas *UMLBAttribute* represents an abstract attribute. Figure 5.2(b) specifies that a UML-B class or refined class may have a number of attributes or/and inherited attributes. Similar to the concept of class, this corresponds to Event-B refinement where a refinement machine may have a number of new variables or/and old variables from its abstract machine.

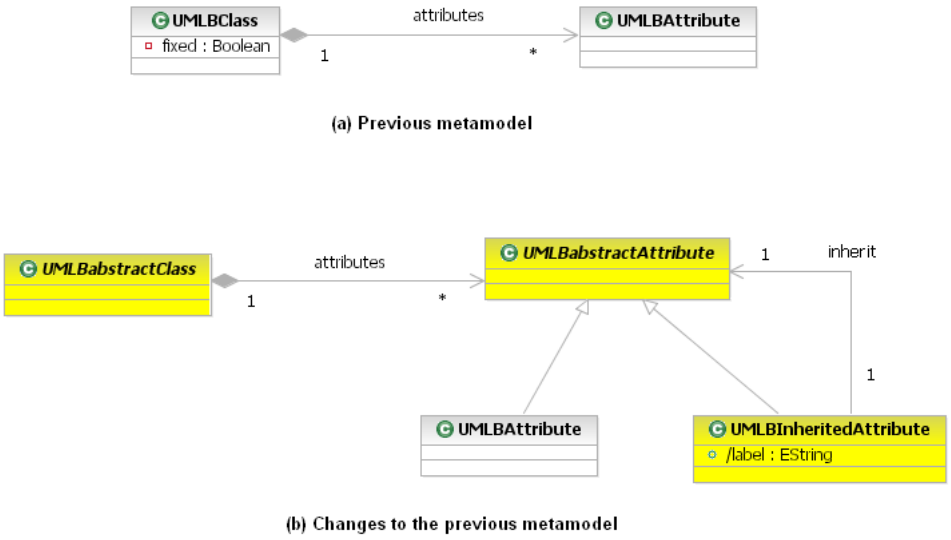


FIGURE 5.2: UML-B Metamodel Extensions for Attributes

A refined class and an inherited attribute do not have names since they refer to the abstract class and the abstract attribute which have names. Therefore, three metaclasses are added (and also attributes of the existing metaclass in the metamodel are changed) in order to support this naming feature.

The three added metaclasses are *UMLBnamedElement*, *UMLBnameConstrainedElement* and *UMLBname*. Since *UMLBRefinedClass* and *UMLBInheritedAttribute* should not inherit name but still needs to inherit error marking from the *UMLBelement*. The naming property was separated so that it can be inherited independently by those metaclasses that need it. Figure 5.3(a) shows the UML-B Version 1 metamodel that is replaced by Figure 5.3(b) .

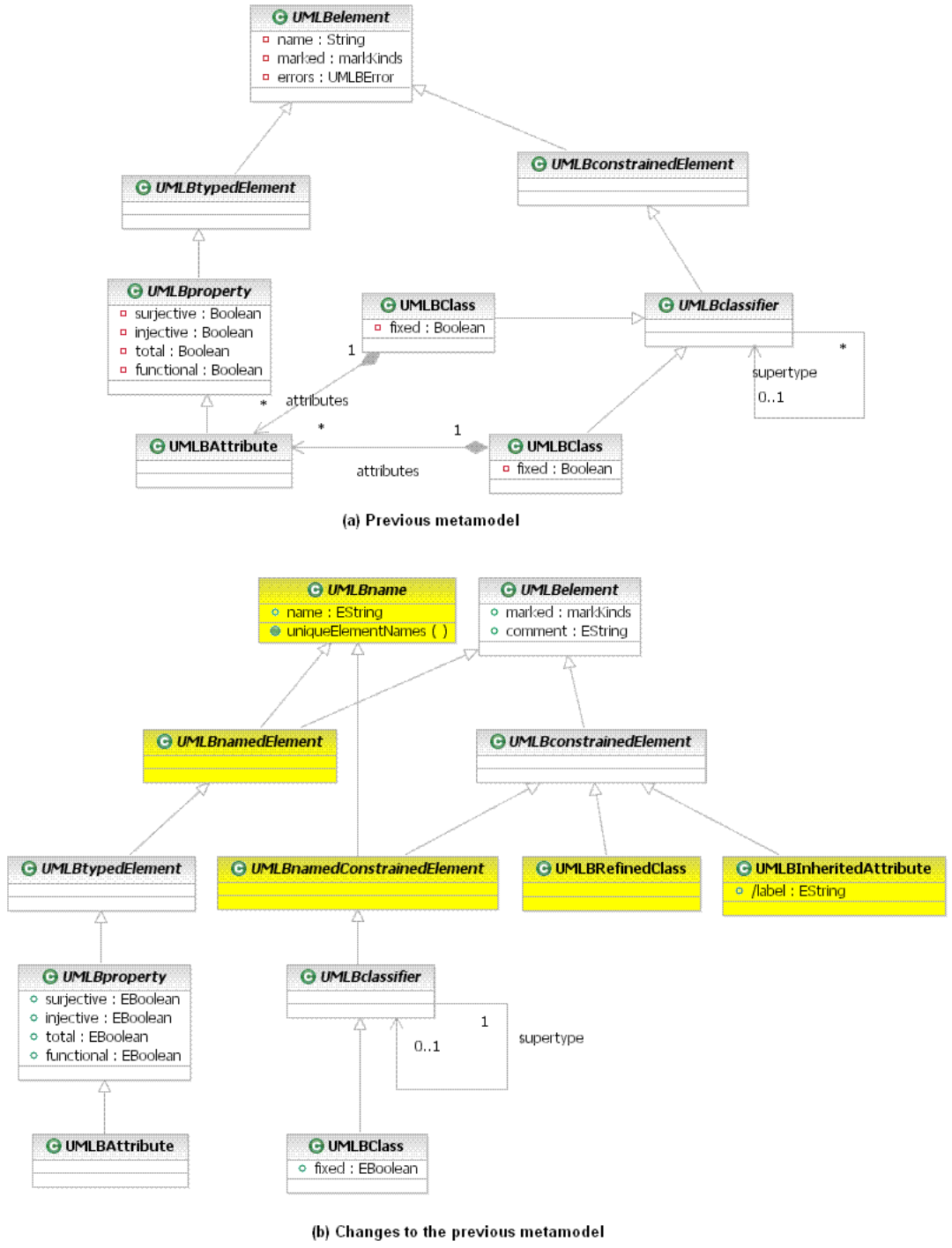


FIGURE 5.3: UML-B Metamodel Extensions for Name Elements

5.3 Extending the UML-B Metamodel and Tool to Support State Machine Refinement

The UML-B Version 1 also has limitations in refining a state machine. The limitations explained in Section 4.2.2 occur because the tool treats the refined state machine as

another new state machine which results in duplication of some elements in the generated Event-B specification.

Changes to the UML-B Version 1 metamodel in Figure 2.20 have been made to overcome the limitation of refining a state machine. The changes involve additional classes and changes to some of the associations between existing and new classes.

Figure 5.4(a) is part of the UML-B Version 1 metamodel that defines the abstract metaclass *UMLBstatemachineCollection* may own many state machines. This is replaced by Figure 5.4(b) that defines that the abstract metaclass *UMLBstatemachineCollection* consists of state machines and refined state machines. The new metaclasses added are *UMLBabstractStatemachine* and *UMLBRefinedStatemachine*. The metaclass *UMLBabstractStatemachine* is an abstract metaclass which allows its common property to be shared by its sub-metaclasses. The new class *UMLBRefinedStatemachine* represents refined state machines and the existing class *UMLBStatemachine* represents state machines. The new metaclass *UMLBabstractStatemachine* is only a generalization class which does not represents any UML-B modelling object.

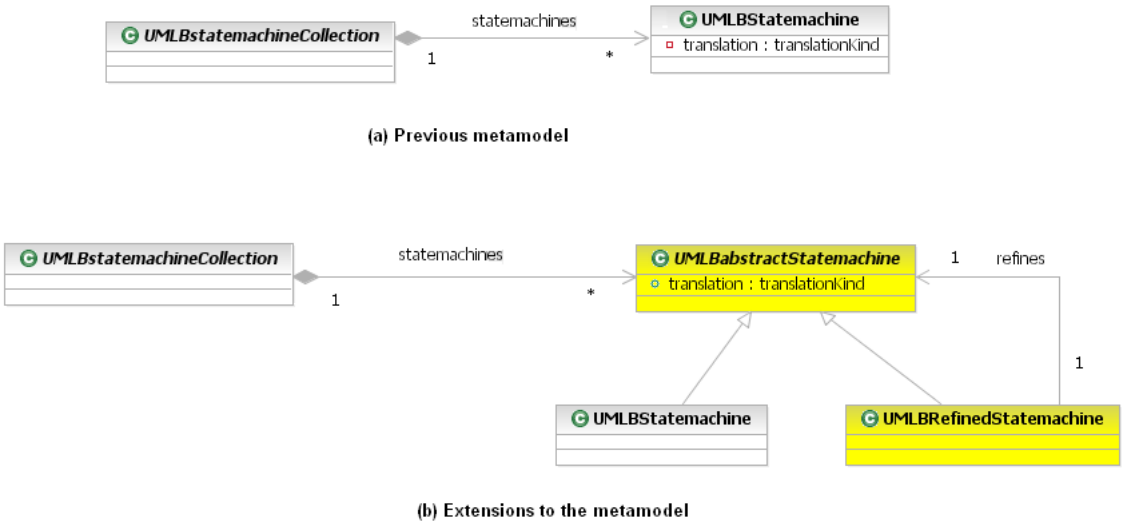


FIGURE 5.4: Changes to UML-B Metamodel for State Machine

Figure 5.5(a) is part of the UML-B Version 1 metamodel that defines a state machine may have many states. This is replaced by Figure 5.5(b) that defines a state machine may have many states which are of two kinds, i.e., state and refined state. The new metaclasses added are *UMLBabstractState* and *UMLBRefinedState*. Similar to the metaclass *UMLBabstractStatemachine*, the metaclass *UMLBabstractState* is an abstract metaclass which allows its common property to be shared by its sub-metaclasses. The new class *UMLBRefinedState* represents refined states and the existing metaclass *UMLBState* represents states. The new metaclass *UMLBabstractState* is only a generalization metaclass

which does not represent any UML-B modelling object.

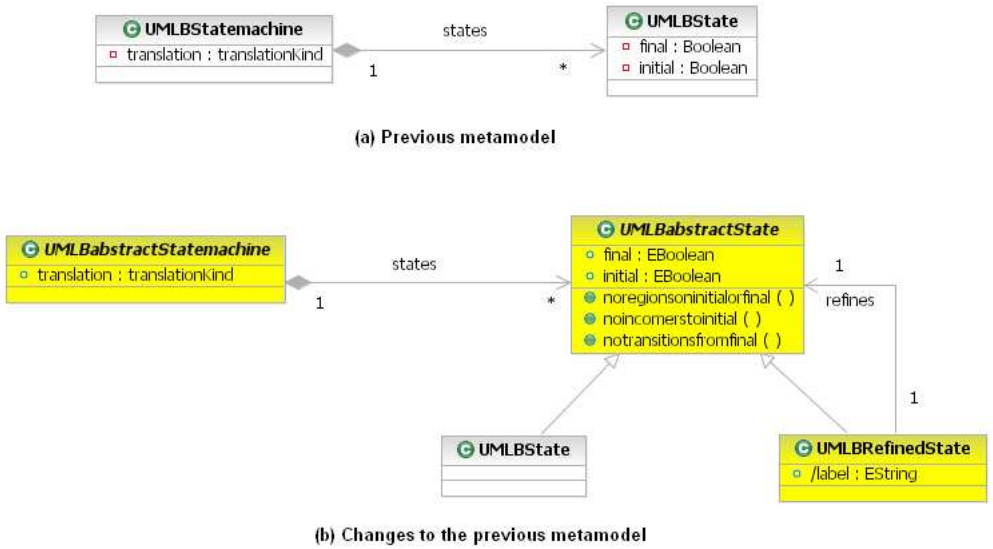


FIGURE 5.5: Changes to UML-B Metamodel for State

A state may have a nested state machine and in UML-B the nested state machine is modelled as a separate diagram from its super-state. A nested state machine will consist of transitions with an initial source state and a final target state. The transitions whose source states are initial states, elaborate the incoming transitions to the super-state while the transitions whose target states are final states, elaborate the outgoing transitions from the super-state. A nested state machine may contain a transition between two sub-states that may elaborate a self loop transition of the super-state. In order to model the elaboration property of a transition, the associations *elaborates* and *IsElaboratedBy* are added to the class *UMLBTransition* in the metamodel. These extensions can be seen in Figure 5.6.

Figure 5.6(a) is part of the UML-B Version 1 metamodel that defines how a state machine may contain many transitions. The metamodel also specified that a transition may refine one or more transitions. This metamodel is replaced by Figure 5.6(b) that defines how both state machines and refined state machines may contain many transitions and a refinement. The new metamodel also specifies that a transition may elaborate a transition and a transition may be elaborated by at most by two transitions. The maximum cardinality of the association *isElaboratedBy* is defined as two so that the source and target states of a transition can be strengthened to sub-states in two nested state machines. The four methods of the metaclass *UMLBTransition* are the model validator which prevent the translation to Event-B when any of the errors specifying the elaborates property occurs. The method *noElaborateIncomingToFinal* validates that a sub-transition with a final target state is not elaborating an incoming parent transition. The method *noElaborateOutgoingFromInitial* validates that a sub-transition with

an initial source state is not elaborating an outgoing parent transition. The methods *noElaborateIncomingByNotInitialNotFinal* and *noElaborateOutgoingByNotInitialNotFinal* validate that a parent transition which is not a self loop transition is not elaborated by a sub-transition where its source and target states are normal states i.e., neither initial nor final state.

FIGURE 5.6: Changes to UML-B Metamodel for Transition

5.4 Extending the UML-B Metamodel and Tool to Support Classtype Extension

The limitations of extending a classtype in UML-B refinement explained in Section 4.2.3 occur because the tool treats the extended classtype as another new classtype. This is because there are no metaclasses in the existing UML-B metamodel which explicitly represent extended classtypes.

Changes were made to the UML-B Version 1 metamodel described in Section 2.10.2. The changes to the previous metamodel have been made to overcome the limitation of classtypes extension. The changes involve additional metaclasses and changes to some

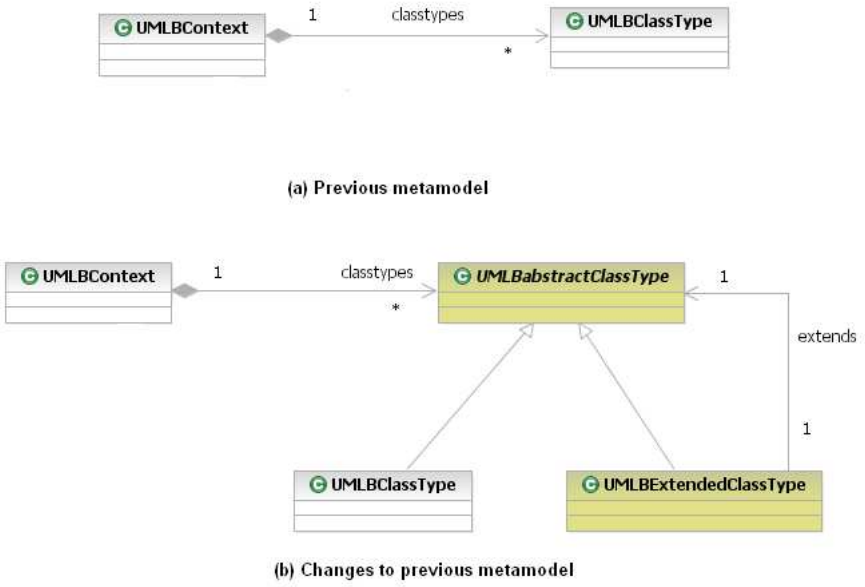


FIGURE 5.7: UML-B Metamodel Extensions for Classtypes

of the associations between existing and new metaclasses.

In UML-B development, a context diagram consists of a number of classtypes where each classtype may contain attributes. A context diagram of an extension context may contain all extended classtypes from its abstract context diagram. The extension of the UML-B metamodel involves adding a metaclass which represents these extended classes.

There are two new metaclasses that have been introduced in the metamodel, *UMLBabstractClassType* and *UMLBExtendedClassType*, for the purpose of realizing the notion of extended classes. Figure 5.7 shows the changes to the UML-B Version 1 metamodel corresponding to the notion of extended classtypes where Figure 5.7(a) is part of the UML-B Version 1 metamodel which specified that a UML-B context may have a number of classtypes. We extend this part of the metamodel by replacing Figure 5.7(a) with Figure 5.7(b) that consists of two new metaclasses. *UMLBabstractClassType* is the base metaclass for *UMLBExtendedClassType* and *UMLBClassType* (existing metaclass). *UMLBExtendedClassType* represents extended classtypes whereas *UMLBClassType* represents classtypes. *UMLBabstractClassType* is a generalization metaclass which does not represent any UML-B modelling object. Figure 5.7(b) specifies that a UML-B context may have a number of new classtypes and/or extended classtypes.

5.5 Extending the UML-B Drawing Tools

The drawing tools of the class diagram and state machine diagram editors were extended to support the refinement of classes and state machines. The extensions involved the

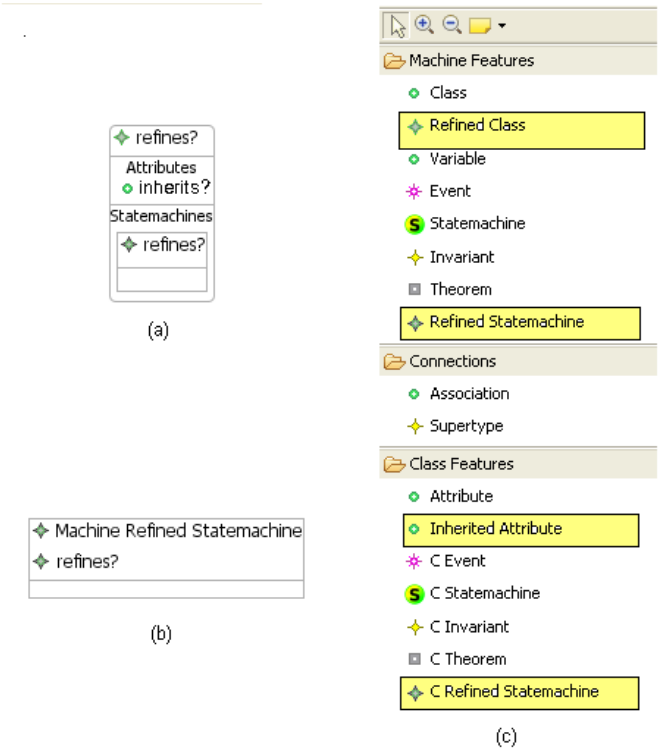


FIGURE 5.8: Drawing Tool Extensions for a Class Diagram Editor

following:

- adding figures for refined classes, refined state machines, inherited attributes and refined states.
- adding creation tools for the above figures to the diagram palette.
- adding properties views for the above figures.

Figure 5.8(a) shows a figure for refined classes which is a white round rectangle. Attached to it is a figure for a refined state machine which is a white rectangle. Also attached is an inherited attribute. The inherited attributes are simply text labels attached to refined classes. Figure 5.8(b) is a figure for a refined state machine which is not attached to a class. The figure in Figure 5.8(b) is to model behaviour of a system which does not involve a set of instances. Figure 5.8(c) is the tool palette which contains the creation tools corresponding to a class diagram editor. Those which are highlighted by yellow boxes are added to create the new figures on the drawing canvas of a class diagram editor.

The refined classes and refined state machines have a *refines* property. Modellers can specify this property by choosing from the combo box list provided by the tool. For the refined classes, the list contains all the classes of the abstract machine which have

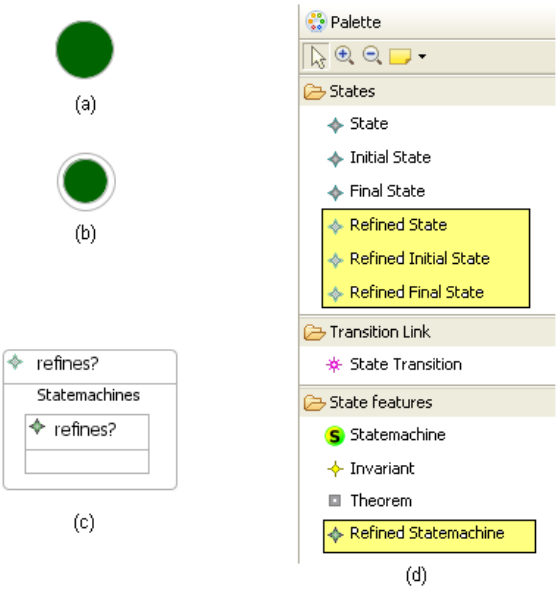


FIGURE 5.9: Drawing Tool Extensions for a State Machine Diagram Editor

not being refined so far. For the class-level refined state machines, the list contains all the state machines of the abstract class which have not been refined so far. For the machine-level refined state machines, the list contains all the state machines of the abstract machine which have not been refined so far. The inherited attributes of refined classes have an *inherited* property. The property can be specified by choosing from a combo box list which contains all the attributes of the abstract class which have not been inherited so far.

The extensions also involved adding figures for refined states in the state diagram editor. There are three figures that have been added to the drawing tool. These are for initial refined states (Figure 5.9(a)), final refined states (Figure 5.9(b)) and normal refined states (Figure 5.9(c)). A refined state machine figure may be attached to a refined state as in Figure 5.9(c).

Figure 5.9(d) shows the tool palette corresponding to a state machine diagram editor. The creation tools highlighted with yellow boxes are the added tools for creating the new figures on the drawing canvas. The three figures of refined states have a *refines* property which can be chosen from the combo box list. The list contains the states of the abstract state machine which have not been refined so far.

5.6 Summary

This chapter described the extensions to the UML-B metamodel which gives precise definitions of the notions of refined class, refined state machine and extended classtype. This chapter also described the extensions to the UML-B drawing tools which use the

metamodel extensions. The metamodel extensions also were used to extend the U2B translator. With the extensions, the UML-B Version 2 supports class refinements, state machine refinements and classtype extensions. Chapter 6 evaluates the extensions to the metamodel using the ATM case study which is modelled in UML-B.

Chapter 6

Modelling The ATM Case Study in UML-B

6.1 Introduction

This chapter and Chapter 8 contain a description of the ATM case study modelled using the UML-B Version 2 tool. This chapter contains the first five machine levels of the ATM development. These machine levels are an incremental development of the case study where a new requirements are added in each machine level. Later in the development, a refined machine is decomposed into three sub-machines. Chapter 8 will describe the development of the ATM case study regarding decomposition. The purpose of this chapter is to validate the notions of refined classes, refined state machines and extended classtypes in Chapter 4. In the case study, all the refinement techniques described in Chapter 4 are applied.

6.2 ATM Case Study: An overview

The UML-B development of the ATM system is based on the requirements in Appendix B. The requirements document contains more complete requirements than the requirements document in Appendix A which were modelled in the plain Event-B development of Chapter 3. The additional requirements are added to model closely the application of ATMs system. These additional requirements will be highlighted in the following sections. The package diagram in Figure 6.1 shows the contexts, the first five level of machines and their relationships where a machine sees a context, a context extends another context and a machine refines another machine. The summary of the first five machine levels are given here.

Abstract machine (ATM_A): Models bank accounts and operations on accounts.

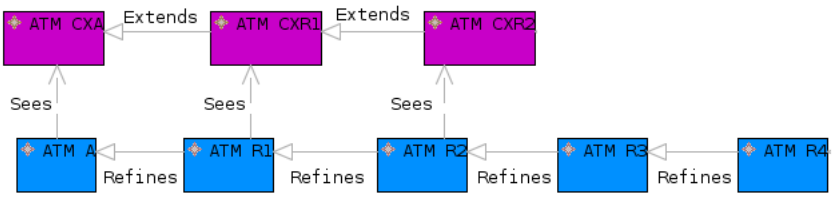


FIGURE 6.1: ATM Package Diagram

First Refinement (ATM_R1): Introduces ATMs as a medium to withdraw money and check balances.

Second Refinement (ATM_R2): Introduces explicit validation for cards.

Third Refinement (ATM_R3): Introduces the request and response communication between an ATM and the bank and splits withdrawal into a bank transition and an ATM transition.

Fourth Refinement (ATM_R4): Introduces the send and receive for the request and response events in the communication between an ATM and the bank and introduces instances of ATM being processed at the bank.

The abstract machine corresponds to the abstract machine of the Event-B development in Section 3.3.1. The first refinement does not correspond to any of the Event-B refinements. It is added in order to model the highest level state machine of an ATM. The second refinement of ATM modelled in UML-B is corresponds to the first refinement of the Event-B development in Section 3.3.2. The third refinement corresponds to the second refinement of the Event-B development in Section 3.3.3. The fourth refinement does not corresponds to any of the Event-B development.

The Event-B specifications for all machine levels are generated automatically when the UML-B models are saved. The generated Event-B specifications are in Appendix D.

6.3 Abstract Machine

The abstract machine models the accounts in a bank and a number of operations that may be performed on the accounts. Figure 6.2 shows a UML-B specification of the ATM abstract model. The abstract model is specified by a machine *ATM_A* which sees a context *ATM_CXA*. The context diagram (Figure 6.2(a)) consists of only one classtype named *Account*. The machine *ATM_A* consists of a class *account* (Figure 6.2(b)) with its attribute *bal* and four events namely, *createAccount*, *deposit*, *withdraw* and *checkBalance*. The class *account* is a subtype of the classtype *Account*. The *account* class represents

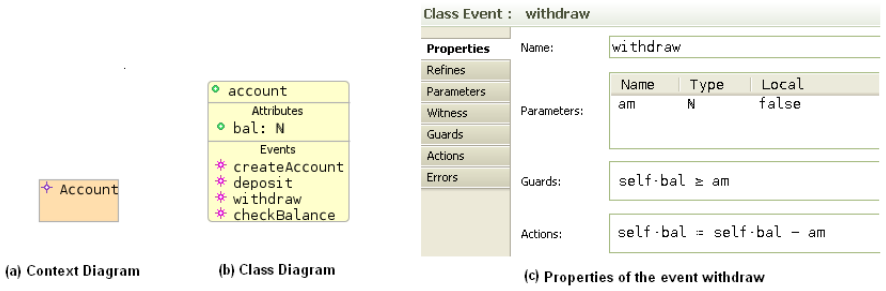


FIGURE 6.2: UML-B Specification of ATM Abstract Machine

the set of accounts that currently exist in the system. The attribute *bal* represents the balance of an account. The *withdraw* event has one added parameter, *am* of type natural number. The parameter is shown in the property view in Figure 6.2(c) including the guard and action. *self* is the self name property defined for the class *account*. The *withdraw* event can only occur if the amount, *am*, is less than or equal to the balance in the account. The *withdraw* event will result in decreasing the balance of the account by *am* amount.

Compared to the Event-B development in Section 3.3.1, the additional events being modelled in the UML-B development are the events *checkBalance*, *createAccount* and *deposit*. However, in the next refinement level, the events *createAccount* and *deposit* were not considered in the development. This is because in this work, we assumed the ATMs could not be used to create accounts or to deposit cash.

6.4 First Refinement

The first refinement introduces a set of ATMs which may be used for cash withdrawal or to check an account balance. A number of new events or transitions are also introduced which will be described later in this section.

This refinement introduces a new class *atm* which represents the sets of ATMs. The UML-B specification is shown in Figure 6.3. The context diagram (Figure 6.3(a)) of context *ATM_CXR1* contains four new classtypes namely *ATM*, *Card*, *ValidCard* and *InvalidCard* and the extended classtype *Account*. The classtypes *ValidCard* and *InvalidCard* subtyping the classtype *Card*. The classtype *ValidCard* is a set of valid card for ATMs whereas the classtype *InvalidCard* is a set of invalid cards. The classtype *ValidCard* has an association *card_account* with the extended classtype *account*. This association gives information about each valid card and a respective account in the bank that it represents. The context also contains two constants and one axiom. The constant *MIN_CASH* represents a constant value of the minimum cash in an ATM. The constant *MAX_CASH* represents a constant value of the maximum cash in an ATM. The axiom

constrains the constant *MAX_CASH* value to be greater than *MIN_CASH*.

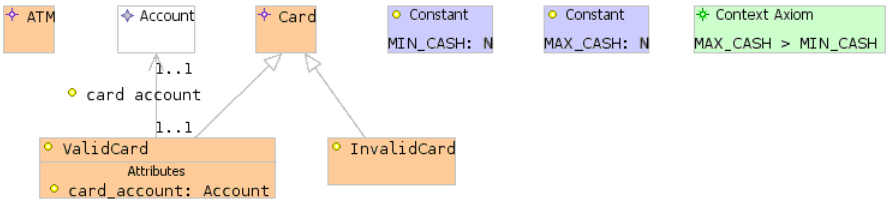
The class diagram (Figure 6.3(b)) of machine *ATM_R1* contains the new class *atm* and a refined class *account* that refines the *account* class of *ATM_A*. The class *atm* has three attributes which are *atm_acbal*, *atm_cash* and *atm_card*. The attribute *atm_acbal* represents an account balance after each withdraw cash or check balance transaction via an ATM. The attribute *atm_cash* represents a stock of cash in an ATM. The attribute *atm_card* represents a card in an ATM.

The refined class *account* inherits the *bal* attribute and refines the two events, namely, *createAccount* and *deposit* of the abstract *account* class of machine *ATM_A*. The other two events of its abstract class namely, *withdraw* and *checkBalance* are moved to the new class *atm* in this refinement level as transitions in the state machine *ATM_SM* of the class *atm*. At the abstract level, we specify the effect of a withdrawal on the account balance. In the refinement, we further specify that the withdrawal takes place via an ATM. At the abstract level it is natural to specify the withdrawal as an event of the *account* class while in the refinement it is natural to specify it as an event of the *atm* class.

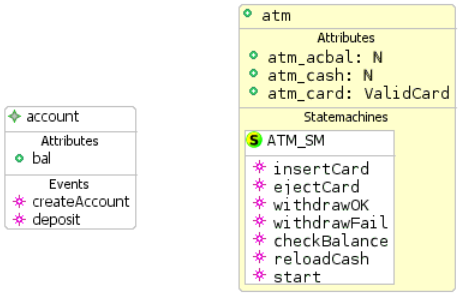
The state machine *ATM_SM* in Figure 6.3(c) partitions the behaviour of an ATM into either an *idle* state, (i.e., not being used/not active) or *active_atm* state (i.e., is being used). An ATM changes its state when it is triggered by a transition. The transition *start* creates an instance of ATM and adds it to the set *atm_card*, initialises its stock of cash as *MAX_CASH* and changes its state to *idle*. The *insertCard* transition can occur when an ATM is in the *idle* state and the card inserted is a valid ATM card. When it occurs it changes an ATM state from *idle* to *active_atm*. The *reloadCash* transition can occur when an ATM is in the *idle* state and the ATM cash amount is less than the *MAX_CASH*. The *reloadCash* transition will top up the ATM cash to the maximum amount *MAX_CASH*. The *ejectCard* transition changes an ATM state from *active_atm* to *idle* and removes the ATM from the set *atm_card*. While an ATM is in *active_atm* state, it means, an ATM user can use it for withdrawal or checking an account balance (i.e., *checkBalance* transition). The *withdrawOK* transition represents a successful withdrawal transaction, whereas, the *withdrawFail* transition represents a failure possibly because the withdrawal amount exceeds the account balance.

Compared to the Event-B development in Chapter 3, the additional transitions which were modelled in the UML-B development are the transitions *start*, *reloadCash* and *checkBalance*.

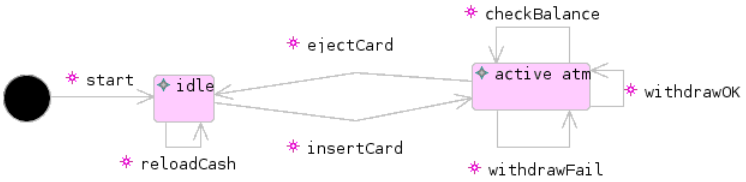
Figure 6.3(d) shows the properties of the *withdrawOK* transition with the parameters, witness, guards and action. The witness specifying that the parameter *ac* represents the *self* parameter of the abstract *withdraw* event. In this refinement, the guards are strengthened so that the *withdrawOK* transition can only occur when an ATM card is inserted (*selfATM.atm_card=c*), the card in the ATM is a valid card for the account



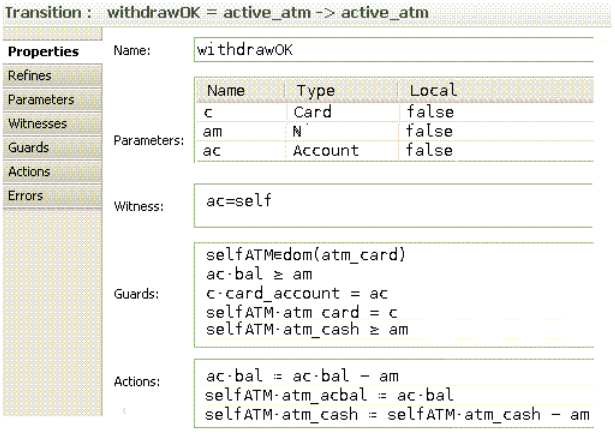
(a) Context Diagram of ATM_CXR1



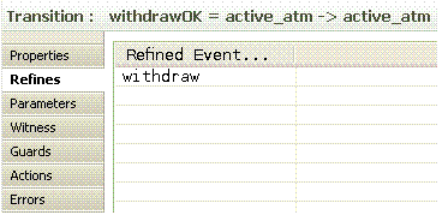
(b) Class diagram of ATM_R1



(c) State machine ATM_SM of class ATM of machine ATM_R1



(d) Properties of withdrawOK event



(e) Refines property of withdrawOK event

FIGURE 6.3: UML-B Specification of ATM First Refinement

whose balance is being modified ($selfATM \in dom(atm\_card)$ and $c.card\_account = ac$), a withdraw amount is less than or equal to the cash in the ATM ($selfATM.atm\_cash \geq am$) and the balance of the account is more than the withdraw amount ($ac.bal \geq am$). The actions of the *withdrawOK* transition are reducing the account balance by the withdraw amount ($ac.bal := ac.bal - am$), set the value of the variable account balance at an ATM to be the value of the account balance ($selfATM.atm\_acbal := ac.bal$) and reducing the ATM cash by the withdraw amount ($selfATM.atm\_cash := selfATM.atm\_cash - am$). Figure 6.3(e) shows the refines property of the *withdrawOK* transition.

6.5 Second Refinement

The second refinement introduces a concept of PIN number and models an explicit validation transition for cards. This is achieved by elaborating the *active_atm* state into sub-states. The context diagram (Figure 6.4(a)) of *ATM_CXR2* contains a new classtype *Pin* and the extended classtypes *ValidCard*. The classtype *ValidCard* has an association *card_pin* with the classtype *Pin* that represents a function that maps a set of valid cards to pin numbers. The class diagram (Figure 6.4(b)) of *ATM_R2* contains the two refined classes that refine the *account* and *atm* classes of *ATM_R1* machine. The refined class *atm* of *ATM_R2* contains the refined state machine *ATM_SM* which contains the two refined states that refine the states *idle* and *active_atm* of the state machine *ATM_SM* of *ATM_R1* (Figure 6.4(c)).

A new state machine named *active_atm_SM* is added to the refined state *active_atm* of *ATM_R2*. It contains four sub-states, namely, *validating*, *invalidCard*, *transOption* and *performTrans* (Figure 6.4(d)). The state machine has a transition *insertCard* which elaborates the incoming transition to the refined super-state *active_atm* of *ATM_R2*. The outgoing transitions *ejectCard1*, *ejectCard2* and *ejectCard3* from the states *invalidCard*, *transOption* and *performTrans* respectively elaborate the outgoing transitions of the refined super-state *active_atm* of *ATM_R2*. The transitions *withdrawOK*, *withdrawFail* and *checkBalance* elaborate the self loop transitions of the refined super-state *active_atm*. The transitions *validateCardOK*, *validateCardFail*, *retry* and *doAnother* are new transitions.

The refined state machine *ATM_SM* of machine *ATM_R2* represents a more detailed system state of an ATM. An ATM changes its state from *idle* to *validating* when an ATM card is inserted (represent by transition *insertCard*). From the *validating* state, an ATM may go to the *transOption* state on successful ATM card validation (represented by transition *validateCardOK*) or an ATM may move into the state *invalidCard* (represented by transition *validateCardFail*). The transition *validateCardOK* can occur if a user entered a correct pin number for a valid card. Otherwise, the transition *validateCardFail* will occur. An ATM may move back into the state *validating* from the state *val-*

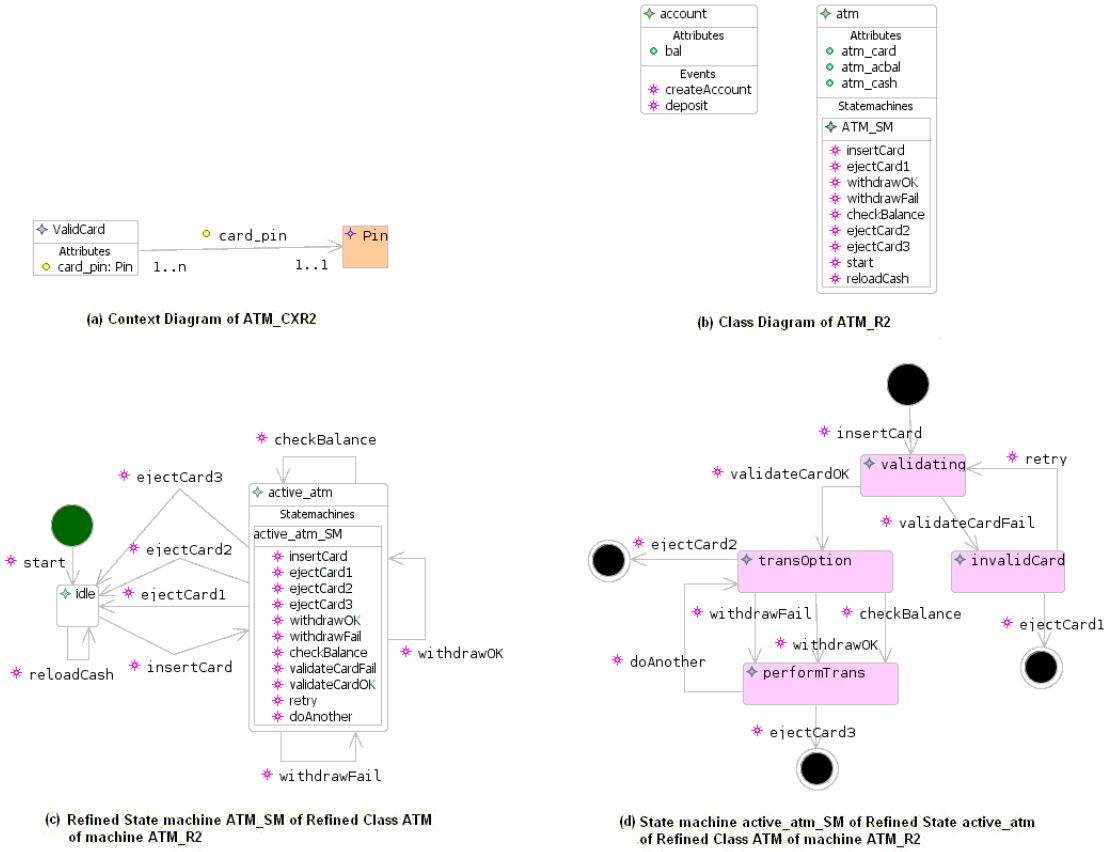


FIGURE 6.4: UML-B Specification of ATM Second Refinement

idateCardFail when a user re-enters the pin number (represent by the transition *retry*). From the *transOption* state, an ATM may go to the state $performTrans$ when either the transition *withdrawOK*, *withdrawFail* or *checkBalance* occurs. From $performTrans$ state, an ATM may go to *idle* state when the transition *ejectCard3* is triggered which will eject the ATM card from the ATM or an ATM may go back to *transOption* state when the transition *doAnother* occurs. Optionally, an ATM user can terminate an activity by ejecting the card while an ATM is in the state *invalidCard*(represent by transition *ejectCard1*) and *transOption* (represent by transition *ejectCard2*).

The refined state machine *ATM_SM* of machine *ATM_R2* in (Figure 6.4(c) and (d)) closely correspond to the ATM system states in Figure 3.3. In this level of UML-B development, the transition *withdrawATM* is not yet introduced. It is introduced in the next refinement machine so that the structure of state machine hierarchy becomes symmetric i.e., when introducing the transition *request* since the transition *withdrawATM* is an effect of the transition *request*. The additional state and transitions in the UML-B development includes the invalid state of an ATM when the event *invalidCard* occurs and also the UML-B development includes the transition for *checkBalance*, the *retry* transition for re-entering the pin number and the transition to repeat a request (tran-

sition *doAnother*). Another difference is that in the UML-B development, we model three options that an ATM user can use to terminate an operation (i.e., the transitions *ejectCard1*, *ejectCard2* and *ejectCard3*). These additional state and transitions were added to the UML-B development in order to closely model the application of ATM system which were not modelled in the plain Event-B development in Section 3.3.2.

6.6 Third Refinement

The third refinement models the transitions *request* and *response*, and splits a withdrawal into a bank transition and an ATM transition. Figure 6.5(a) is the class diagram of *ATM_R3* showing the refined classes and invariants. Three new attributes are introduced in the refined class *atm*. The attribute *atm_wdam* represents a requested withdrawal amount at each ATM. The attribute *atm_acbalA* represents an account balance at an ATM which is notified to a user. The attribute *atm_cashA* represents the stock of cash of an ATM which replaces the attribute *atm_cash* of *ATM_R2*. This is done because a new transition *withdrawATMOK* is introduced in this refinement. The action which set the attribute *atm_cash* by the transition *withdrawOK* of *ATM_R2* is moved to the new transition. The movement is done to separate the transitions happening at a bank and ATMs.

Figure 6.5(b) is a refined state machine *active_atm_SM* of machine *ATM_R3* which shows that the refined state *transOption* and the refined state *performTrans* have nested state machines. The request event is achieved by elaborating the refined states *transOption* into sub-states. The refined state *transOption* has a state machine *transOption_SM* (Figure 6.5(c)) which contains the states *trans*, *reqWD* and *reqCB*. The transitions *validateCardOK* and *doAnother* elaborate the incoming transitions to the refined super-state *transOption* of *ATM_R3*. The transitions *ejectCard2*, *withdrawOK*, *withdrawFail* and *checkBalance* elaborate the outgoing transitions from the refined super-state *transOption* of *ATM_R3*. The transitions *requestWD* and *requestCB* are new transitions which represent transition requests from an ATM to the bank.

The following table shows the explicit parameter, guards and action of the transition *requestWD*. These are added as the properties of the transition which give rise to the parameter, guards and action of the Event-B event. The implicit parameter *selfATM*, guard and actions which involve the source and target states of the transition are generated automatically from the class and state machine. The transition *requestWD* can be triggered when an ATM is in state *trans*, there is an ATM card in the ATM (*grd1*), the amount of cash in the ATM is greater than the constant minimum cash (*grd2*) and the withdrawal amount is less or equal to the minimum cash (*grd3*). The transition *requestWD* will move an ATM from the state *trans* into the state *reqWD* and it will set the value of the withdrawal amount variable (*act1*).

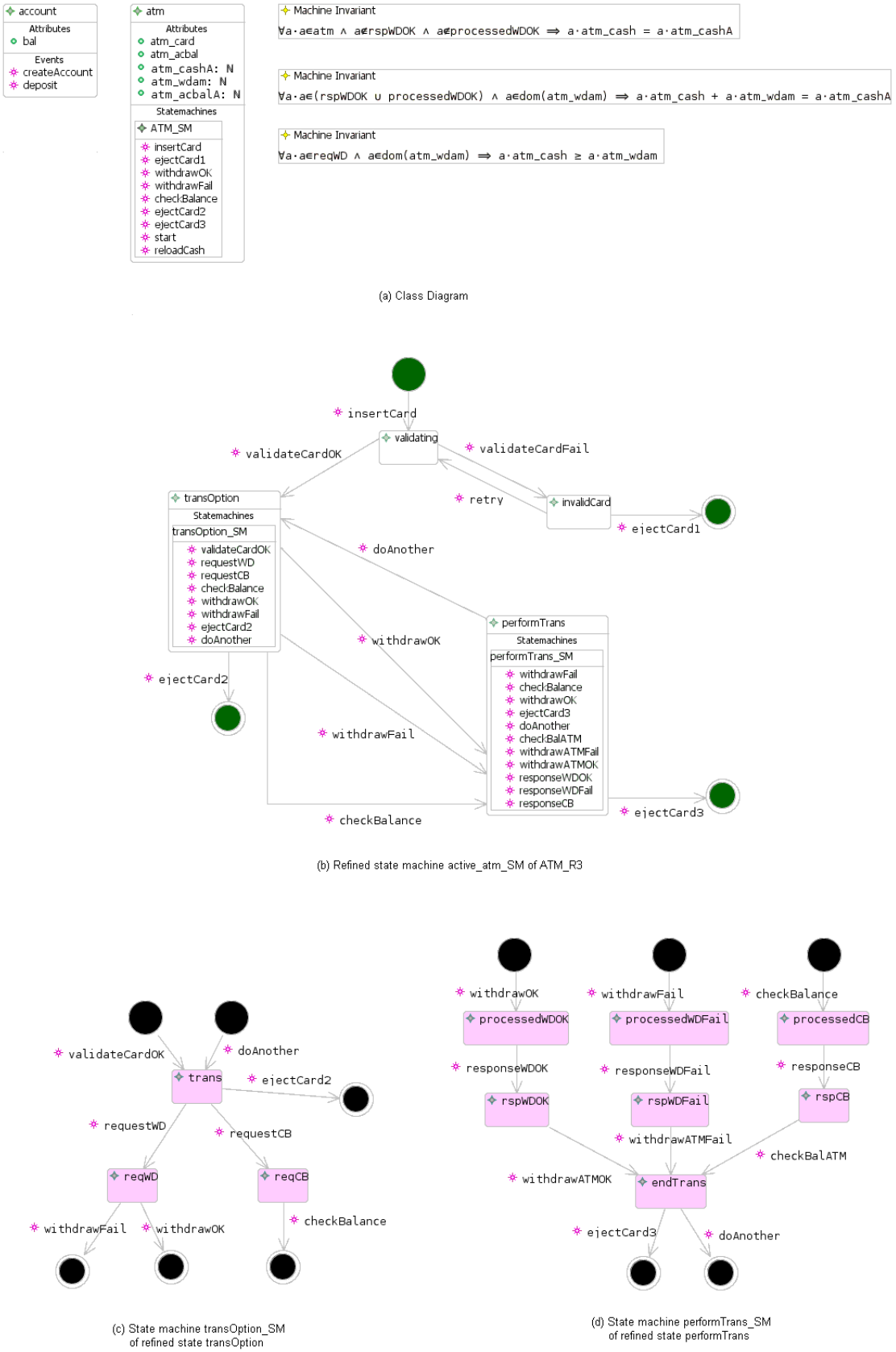


FIGURE 6.5: UML-B Specification of ATM Third Refinement

Transition: requestWD
Parameters: am (type: $\mathbb{N}$)
Guards: grd1: $selfATM \in dom(atm\_card)$ grd2: $atm\_cashA(selfATM) > MIN\_CASH$ grd3: $am \leq MIN\_CASH$
Actions: act1: $atm\_wdam(selfATM) := am$

The transition *requestCB* can be triggered when an ATM is in the state *trans* and there is an ATM card in the ATM. It will move an ATM from the state *trans* into the state *reqCB*.

The effect of the request is modelled in the refined state *performTrans*. The refined state *performTrans* has a state machine *performTrans_SM* (Figure 6.5(d)) which contains the states *processedWDOK*, *processedWDFail*, *processedCB*, *rspWDOK*, *rspWDFail*, *rspCB* and *endTrans*. The transitions *withdrawOK*, *withdrawFail* and *checkBalance* elaborate the incoming transitions to the refined super-state *performTrans* of *ATM_R3*. The transitions *ejectCard3* and *doAnother* elaborate the outgoing transitions from the refined super-state *performTrans* of *ATM_R3*. The transitions *responseWDOK*, *responseWDFail*, *responseCB*, *withdrawATMOK*, *withdrawATMFail* and *checkBalATM* are new transitions.

The transition *responseWDOK* can be triggered when an ATM is in the state *processedWDOK*, there is an ATM card in the ATM (*grd1*) and there is an account balance associated with the ATM (*grd2*). The transition *responseWDOK* will move an ATM from the state *processedWDOK* into the state *rspWDOK* and it will make a copy of the account balance (*act1*).

Transition: responseWDOK
Guards: grd1: $selfATM \in dom(atm\_card)$ grd2: $selfATM \in dom(atm\_acbal)$
Actions: act1: $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

The transitions *responseWDFail* and *responseCB* will behave similar to the transition *responseWDOK*. But the states when they occur and the state changes after their occurrences are different. These state changes can be seen in the state machine in Figure 6.5(d).

Transition: withdrawOK
Parameters: par1: c (type: $ValidCard$) par2: am (type: $\mathbb{N}$) par3: ac (type: $account$)
Guards: grd1: $selfATM \in dom(atm\_card)$ grd2: $selfATM \in dom(atm\_wdam)$ grd3: $atm\_card(selfATM) = c$ grd4: $bal(ac) \geq am$ grd5: $atm\_wdam(selfATM) = am$ grd6: $card\_account(c) = ac$
Actions: act1: $bal(ac) := bal(ac)$ - am act2: $atm\_acbal(selfATM) := bal(ac)$ - am

In this refinement, the abstract transition *withdrawOK* is split between the transition *withdrawOK* which refines the abstract transition and the new transition *withdrawATMOK*. This is done to separate the transitions at the bank and ATMs. The *withdrawOK* is a transition at the bank where it will deduct an account balance by a withdrawal amount (*act1*) and set the value of the account balance (*act2*). In this refinement *grd2* and *grd5* are added to the transition *withdrawOK* constraining that the withdrawal amount must exist in an ATM (*grd2*) and the withdrawal amount (*grd5*) is less or equal to an account balance (*grd4*).

The transition *withdrawATMOK* is a transition at an ATM where it will reduce the cash in the ATM (*act1*). The transition *withdrawATMOK* is triggered when an ATM is in a state *rspWDOK*, the ATM holds a valid card (*grd1*), *am* is a withdrawal amount (*grd2* and *grd5*) and is less or equal to the cash in the ATM (*grd6*), and the ATM has the value of an account balance to notify a user (*grd3*, *grd4* and *grd7*).

Similarly, the transitions *withdrawATMFail* and *checkBalATM* will notify the value of an account balance to an ATM user. However, there are no guard and action related to a withdrawal amount.

Transition: withdrawATMOK
Parameters: am (type: $\mathbb{N}$)
Guards: grd1: $selfATM \in dom(atm\_card)$ grd2: $selfATM \in dom(atm\_wdam)$ grd3: $selfATM \in dom(atm\_acbalA)$ grd4: $selfATM \in dom(atm\_acbal)$ grd5: $atm\_wdam(selfATM) = am$ grd6: $atm\_cashA(selfATM) \geq am$ grd7: $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$
Actions: act1: $atm\_cashA(selfATM) := atm\_cashA(selfATM)$ - am

This refinement relies on the three invariants in Figure 6.5(a). The invariant

$\forall\ a \cdot\ a \in (rspWDOK \cup processedWDOK) \wedge a \in dom(atm\_wdam) \Rightarrow atm\_cash(a) + atm\_wdam(a) = atm\_cashA(a)$

specifies that for all ATMs which are in a state $rspWDOK$ or $processWDOK$, the cash amount in the refinement is the same as the summation of the cash amount at the abstract level and the withdrawal amount. This invariant is needed because changes to account balances and cash in ATMs are split into separate events.

The invariant

$\forall\ a \cdot\ a \in atm \wedge a \notin rspWDOK \wedge a \notin processedWDOK \Rightarrow atm\_cash(a) = atm\_cashA(a)$

specifies that for all ATMs which are not in the states $rspWDOK$ or $processWDOK$, the cash amount in the refinement is the same as the cash amount at the abstract level.

The invariant

$\forall\ a \cdot\ a \in reqWD \wedge a \in dom(atm\_wdam) \Rightarrow atm\_cash(a) \geq atm\_wdam(a)$

is needed because when replacing the abstract variable *atm_cash* with the new variable *atm_cashA* to preserve the guard $atm\_cash(a) \geq am$ in the event *withdrawOK* of *ATM_R2*. This invariant specifies that when ATMs are in the state *reqWD*, the value of an ATM cash is more than or equal the value of a withdrawal amount.

The invariants are discovered by using the Rodin interactive provers. Examples of using the provers to construct invariants are given in Section 6.8.

The state machines *transOption_SM* (Figure 6.5(c)) and *performTrans_SM* (Figure 6.5(d)) correspond to the sub-states in Figure 3.4 of the Event-B development. The differences are the additional states and transitions in the UML-B development which model the

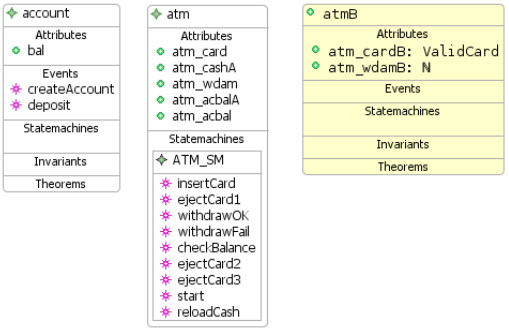


FIGURE 6.6: Class Diagram of the Fourth Refinement

check balance transaction that is not modelled in the Event-B development. Another difference is the introduction of the transitions *withdrawATMOK*, *withdrawATMFail* and *checkBalATM* in the refinement of the UML-B development.

6.7 Fourth Refinement

The fourth refinement models the send and receive events of the request and response communication between ATMs and the bank. This is done by adding a receive event for each request and adding a send event for each response. The send event for request refines the abstract request event. The receive event for response refines the abstract response event. The fourth refinement also introduces a set of requesting ATMs whose requests are being processed by the bank.

The class diagram of the fourth refinement can be seen on Figure 6.6. The two refined classes *account* and *atm* refine the abstract class of *ATM_R3*. The new class *atmB* represents ATMs as the bank. It has attributes *atm_wdamB* and *atm_cardB*. As mentioned in the introduction of this chapter, it is our intention to decompose the machine into three machine components i.e., ATM, bank and middleware which model the states and transitions at ATM, bank and middleware respectively. The transitions of these machines should not share attributes among them. The transition *withdrawOK*, *withdrawFail* and *checkBalance* are the transitions at the bank. In the third refinement, the transition *withdrawOK* is sharing the attributes *atm_card* and *atm_wdam* with the other transitions occur at ATMs. Therefore in this refinement a copy of these attributes is made. The attribute *atm_cardB* is a copy of the attribute *atm_card* made when a bank receives a request, i.e., *recvdReqWD* and *recvdReqCB* transitions. The attribute *atm_wdamB* is a copy of the attribute *atm_wdam* made when a bank receives a request.

The *receive request* events are achieved by elaborating the refined states *reqWD* and *reqCB* of the nested state machine *transOption_SM*. The *sending response* events are achieved by elaborating the refined states *processedWDOK*, *processedWDFail* and *pro-*

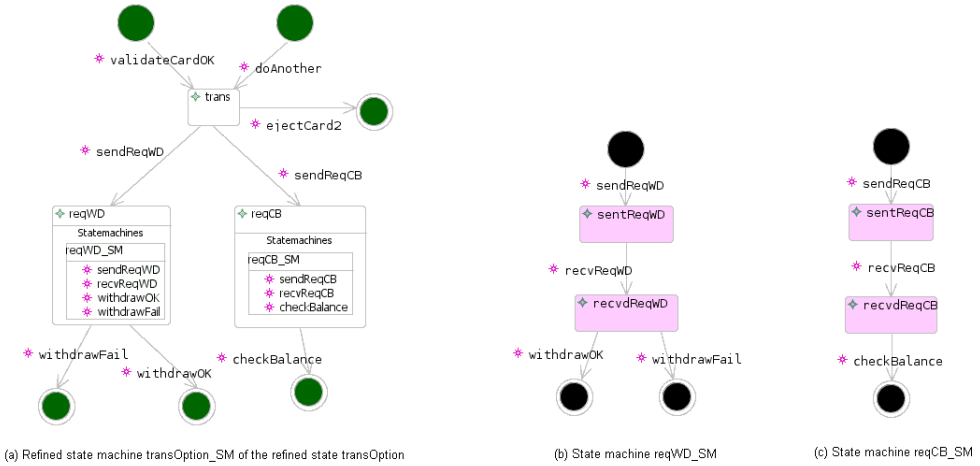


FIGURE 6.7: UML-B Specification of the Fourth Refinement: Request Event

cessedCB of the nested state machine *performTrans_SM*.

Figure 6.7(a) shows the refined state machine *transOption_SM*. The transitions *validateCardOK* and *doAnother* elaborate the incoming transition to the state *transOption* that own the state machine *transOption_SM*. The transitions *ejectCard2*, *withdrawOK*, *withdrawFail* and *checkBalance* elaborate the outgoing transitions from the state *transOption*. Attached to the state *reqWD* is the state machine *reqWD_SM*. The incoming transition *sendReqWD* refines the transition *requestWD* of *ATM_R3*. Attached to the state *reqCB* is the state machine *reqCB_SM*. The incoming transition *sendReqCB* refines the transition *requestCB* of *ATM_R3*.

Figure 6.7(b) shows the nested state machine *reqWD_SM* with a new transition *recvReqWD* which represents a withdrawal request receipt event at the bank. The transition *sendReqWD* elaborates the incoming transition of the refined super-state *reqWD*. It means the target state for the transition *sendReqWD* is the state *sentReqWD*. The transitions *withdrawOK* and *withdrawFail* elaborate the outgoing transitions of the refined super-state *reqWD*. It means the source state for the transitions *withdrawOK* and *withdrawFail* is the state *recvdReqWD*. Figure 6.7(c) shows the nested state machine *reqCB_SM* with a new transition *recvReqCB* that represents a check balance request receipt event at the bank. The transition *sendReqCB* elaborates the incoming transition of the refined super-state *reqCB*. It means the target state for the transition *sendReqCB* is the state *sentReqCB*. The transitions *checkBalance* elaborate the outgoing transition of the refined super-state *reqCB*. It means the source state for the transition *checkBalance* is the state *recvdReqCB*.

The new transition *recvReqWD* is triggered when an ATM is in the state *sentReqWD*, the ATM has a withdrawal amount (*grd1*) and holds a valid ATM card (*grd2*). The transition will move the ATM from the state *sentReqWD* into *recvdReqWD*, it will add the ATM into the set *atmB* (*act1*), make a copy of the ATM card (*act2*) and make a

copy of the withdrawal amount (*act3*).

Transition: recvReqWD
Guards: grd1: $selfATM \in dom(atm\_wdam)$ grd2: $selfATM \in dom(atm\_card)$
Actions: act1: $atmB := atmB \cup \{selfATM\}$ act2: $atm\_cardB(selfATM) := atm\_card(selfATM)$ act3: $atm\_wdamB(selfATM) := atm\_wdam(selfATM)$

Similar to the transition *recvdReqWD*, transition *recvdReqCB* will make a copy of the ATM card.

The transitions *withdrawOK*, *withdrawFail* and *checkBalance* occur at the bank and they refine their corresponding transition of *ATM_R3*. In this refinement, for the transitions *withdrawOK*, *withdrawFail* and *checkBalance*, the new attributes *atm_cardB* and *atm_wdamB* replace the attributes *atm_card* and *atm_wdam* respectively in the guards.

This refinement relies on the following four invariants. The invariant

$$\forall\, a\cdot\, a\in (recvdReqWD \cup recvdReqCB) \Rightarrow a\in dom(atm\_card)$$

specifies that each instance of ATMs which is either in the states *recvdReqWD* or *recvdReqCB*, has an ATM card associated with it.

The invariant

$\forall\, a\cdot\, a\in (recvdReqWD \cup recvdReqCB) \wedge a\in dom(atm\_card) \wedge a\in dom(atm\_cardB)$
$\Rightarrow atm\_card(a) = atm\_cardB(a)$

specifies that each instance of ATMs which is in the states *recvdReqWD* or *recvdReqCB*, the ATM card that is associated with the ATMs at the ATM is the same as the ATM card associated with the ATMs at the bank.

The invariant

$$\forall\, a\cdot\, a\in recvdReqWD \Rightarrow a\in dom(atm\_wdam)$$

specifies that each instance of ATMs which is in the state *recvdReqWD*, has a withdrawal amount associated with it.

The invariant

$\forall\, a \cdot a \in recvdReqWD \wedge a \in dom(atm\_wdam) \wedge a \in dom(atm\_wdamB) \Rightarrow atm\_wdam(a) = atm\_wdamB(a)$

specifies that each instance of ATMs which is in the state *recvdReqWD*, the withdrawal amount that is associated with the ATMs at the ATM is the same as the withdrawal amount associated with the ATMs at the bank. In practice, the above invariants are specified as machine invariants in the UML-B class diagram.

Figure 6.8(a) shows the refined state machine *performTrans_SM*. The transitions *withdrawOK*, *withdrawFail* and *checkBalance* elaborate the incoming transition to the refined super-state *performTrans*. The transitions *ejectCard3* and *doAnother* elaborate the outgoing transitions from the refined super-state *performTrans*. The transitions *recvRspWDOK*, *recvRspWDFail*, *recvRspCB*, *withdrawATMOK*, *withdrawATMFail* and *checkBalATM* refine their corresponding transitions of *ATM_R3*. Each of the states *processedWDOK*, *processedWDFail* and *processedCB* has a nested state machine *processedWDOK_SM*, *processedWDFail_SM* and *processedCB_SM* respectively.

Figure 6.7(b) shows the nested state machine *processedWDOK_SM* with a new transition *sendRspWDOK* which represents a send response event from a bank approving a withdrawal request. The transition *withdrawOK* elaborates the incoming transition of the refined super-state *processedWDOK*. It means the target state for the transition *withdrawOK* is the state *processWDOK*. The transitions *recvRspWDOK* elaborates the outgoing transition of the refined super-state *processedWDOK*. It means the source state for the transitions *recvRspWDOK* is the state *sentRspWDOK*. Figure 6.7(c) shows the nested state machine *processedWDFail_SM* with a new transition *sendRspWDFail* that represents a send response event from a bank disapproving a withdrawal request. The transition *withdrawFail* elaborates the incoming transition of the refined super-state *processedWDFail*. It means the target state for the transition *withdrawFail* is the state *processWDFail*. The transition *recvRspWDFail* elaborates the outgoing transition of the refined super-state *processedWDFail*. It means the source state for the transition *recvRspWDFail* is the state *sentRspWDFail*. Figure 6.7(d) shows the nested state machine *processedCB_SM* with a new transition *sendRspCB* which represents a send response event from a bank notifying an account balance. The transition *checkBalance* elaborates the incoming transition of the refined super-state *processedCB*. It means the target state for the transition *checkbalance* is the state *processCB*. The transition *recvRspCB* elaborates the outgoing transition of the refined super-state *processedCB*. It means the source state for the transition *recvRspCB* is the state *sentRspCB*.

The new transition *sendRspWDOK* is a transition at the bank which occurs when a requesting ATM is in the state *processWDOK*, the ATM is in the class *atmB* (*grd1*) and there is an account balance associated with the ATM (*grd2*). The transition will move the ATM from the state *processWDOK* into *sentRspWDOK* and it will remove the ATM from the class *atmB* (*act1*) and from the domain of the attributes *atm_cardB*

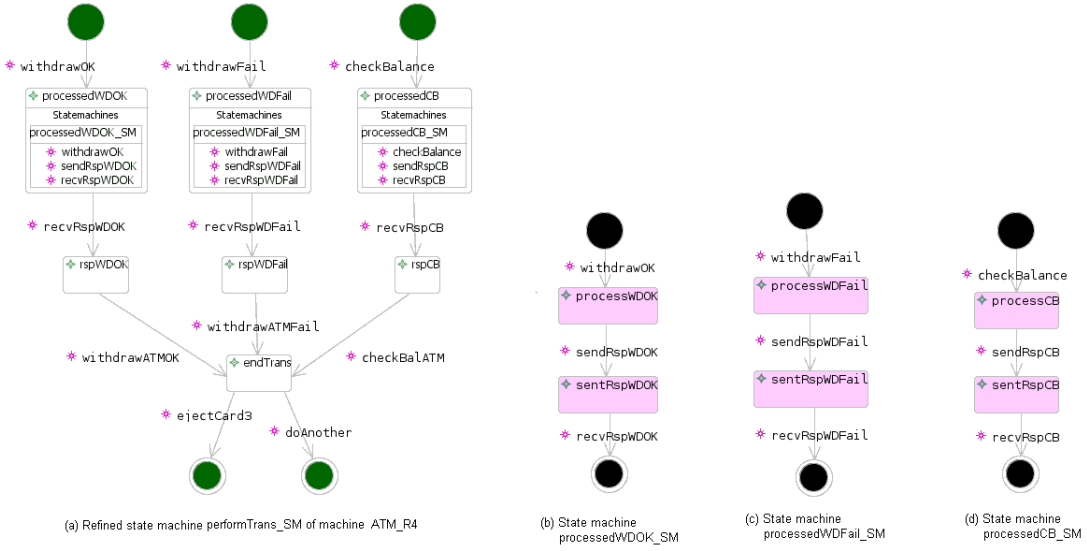


FIGURE 6.8: UML-B Specification of the Fourth Refinement: Response Event

(*act2*) and *atm_wdamB* (*act3*). In this level, *act1*, *act2* and *act3* are explicitly added as the actions. Later in the development when applying decomposition as in Section 8.6.2, the transition *sendRspWDOK* is modelled as a destructor. In this case, the three actions are generated automatically.

Transition: sendRspWDOK
Guards:
grd1: $selfATM \in atmB$
grd2: $selfATM \in dom(atm\_acbal)$
Actions:
act1: $atmB := atmB \setminus \{selfATM\}$
act2: $atm\_cardB := \{ selfATM\} \lhd atm\_cardB$
act3: $atm\_wdamB := \{ selfATM\} \lhd atm\_wdamB$

The guards of the transitions *sendRspWDFail* and *sendRspCB* are similar to the transition *sendRspWDOK* except the guard and action regarding the states of the ATMs are different. The transition *sendRspWDFail* occurs when an ATM is in the state *processWDFail*. This transition will move an ATM into the state *sentRspWDFail*. The transition *sendRspCB* occurs when an ATM is in the state *processCB*. This transition will move an ATM into the state *sentRspCB*.

Compared to the Event-B development in Chapter 3, the additional transitions which were modelled in the UML-B development are the transitions *recvReqCB*, *sendRspCB*, *sendRspWDOK* and *sendRspWDFail*.

Machines	**POs**	**aPOs**	**iPOs**
ATM_A	4	4	0
ATM_R1	47	47	0
ATM_R2	68	68	0
ATM_R3	167	160	7
ATM_R4	149	142	7
Total	435	421	14

TABLE 6.1: Statistics from the Proof Effort

6.8 Experiences from the ATM system in UML-B

This section gives an assessment of the experience of modelling using UML-B Version 2.

All the proof obligations (POs) for the five machines of the ATM case study were generated and proved using the Rodin tool provers [4]. The statistics are outlined in Table 8.1 showing the total POs for each level (POs), the number of POs which are automatically discharged (aPOs) and the number of POs which are interactively discharged (iPOs).

In *ATM_R3*, there are seven interactively discharged POs. Three POs are discharged manually by proving that two related states are disjoint and another four are proved by rewriting the partition invariant into its definition. A similar way is used to prove the seven interactively POs in ATM_R4. Two POs are discharged by manually proving that two states are disjoint and the other five POs are discharged by rewriting the partition invariant.

The state machine refinement in the second (*ATM_R2*), third (*ATM_R3*) and fourth (*ATM_R4*) refinements introduces additional levels in the state machine nesting hierarchy. This supports modular reasoning, since refinement invariants are only required for the states that are being elaborated, so it localizes proof effort. For example, for the fourth refinement (*ATM_R4*), the refined states *reqWD*, *reqCB*, *processedWDOK*, *processedWDFail* and *processedCB* are elaborated into sub-states. The gluing invariants required are only for these states and their sub-states. Thus, the generated proof obligations only concern with the events that occur during these states.

Similar to the experience in Event-B, some of the invariants are constructed by using the provers of the Rodin tool. One of them is the gluing invariant in the third refinement (*ATM_R3*). An attempt to construct the invariant is done by using the interactive prover. The *ATM_R3* was run in a proving perspective without having any gluing invariant which result in a number of undischarge proof obligations. The first undischarged PO is given here as an example. The prover cannot discharge the guard $atm\_cash(selfATM) \geq am$ of the event *withdrawOK*. The hypotheses and the goal are as follows:

Hypotheses:

$selfATM \in atm$
$selfATM \in reqWD$
$selfATM \in dom(atm\_card)$
$atm\_card(selfATM) = c$
$selfATM \in dom(atm\_wdam)$
$atm\_wdam(selfATM) = am$
$atm\_card(selfATM) \in ValidCard$
$card\_account(msg\_card(m))=ac$
$card\_account(msg\_card(m))\in account$
$atm\_wdam(selfATM) \in \mathbb{N}$
$bal(card\_account(atm\_card(selfATM)) \geq atm\_wdam(selfATM)$

The goal:

$atm\_cash(selfATM) \geq atm\_wdam(selfATM)$

From the hypotheses and the goal, it is deduced that an invariant is needed to say that for all *selfATM* where *selfATM* is in *reqWD*, the goal is implied. The invariant represented in Event-B as:

$\forall selfATM \cdot selfATM \in reqWD \wedge selfATM \in dom(atm\_wdam) \Rightarrow atm\_cash(selfATM) \geq atm\_wdam(selfATM)$

6.9 Summary

This chapter presented the use of seven techniques of Chapter 4 in the ATM case study which was modelled using the UML-B tool. The approach of elaborating states with sub-states in refinement supports an incremental refinement approach. The hierarchical structure of nested state machines also supports modular reasoning by localising the invariants required for refinement proofs into the relevant state and its sub-states. The ATM case study has evaluated that the extensions of the metamodel in Chapter 5 are working as expected.

Chapter 7

Decomposition and Composition in UML-B

7.1 Introduction

In a UML-B development, we may want to decompose a refined machine into a number of machines (or components). These decomposed machines can then be refined individually. This is done to reduce the complexity of modelling and proving a system.

Our goal is to support the decomposition of a single UML-B model into a distributed system which consists of requesting components, responding components and middleware component. The middleware synchronizes the communication between the requesting and responding components. Figure 7.1 illustrates an example of a decomposition of a distributed system where a UML-B machine is decomposed into three machines. In the example, the state machine *SM* of machine *M* is decomposed into four state machines: state machine *SMa* of machine *M1*, state machines *SMb* and *SMc* of machine *M*2 and state machine *SMd* of machine *M3*. *M1* represents a requesting component that sends a request to the responding component *M3*. *M2* represents a middleware and it has two state machines, one is for synchronizing requests and another is for synchronizing responses.

This chapter introduces two techniques of state machine refinement preceding a decomposition of a refined machine as in Figure 7.1. The two techniques of refining state machines are

- Flattening state machines
- State grouping

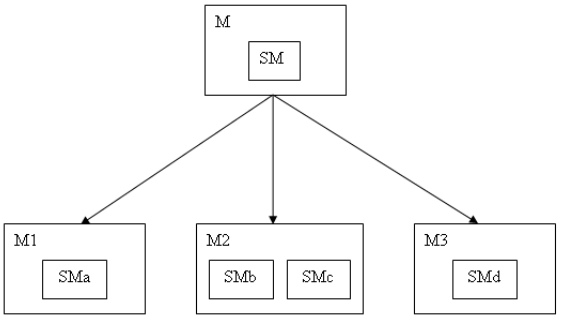


FIGURE 7.1: Decomposition of State Machines

The decomposition in this work is based on the event-based or shared-event decomposition. The shared-event decomposition approach splits an event between two components. This kind of decomposition is suitable for use in a distributed system using message passing.

Silva and Butler have introduced the notion of *composed machine* in [89] for composing several machines (decomposed machines) to support the event-based decomposition in Event-B. Their work is described in Section 7.4. The advantage of the composition is that the decomposed machines can then be refined individually. Proving that the composed machine is a refinement of the abstract machine is performed by discharging all the proof obligations of the composed machine.

The decomposed machines *M1*, *M2* and *M3* in Figure 7.1 are abstract machines which are not refinements of other machines. Our intention is to have a composed machine which refines machine *M* and composes the machines *M1*, *M2* and *M3* to ensure that their composition are valid refinement of *M*. Therefore, the decomposed machines may be refined individually and this should reduce the complexity of modelling and proving a complex system.

UML-B Version 2 did not have support for composing several machines. This chapter describes the notions of *composed machines*, *included machines*, *composed events* and *combined events* in UML-B. These notions are the extension to the UML-B language to support composition and decomposition. This chapter also describes the extensions to the UML-B metamodel which give a precise definition of the notions. These notions correspond to the notion of a composed machine and its structure in Event-B. A section on the extensions to the UML-B drawing tool to support composition is also included in this chapter.

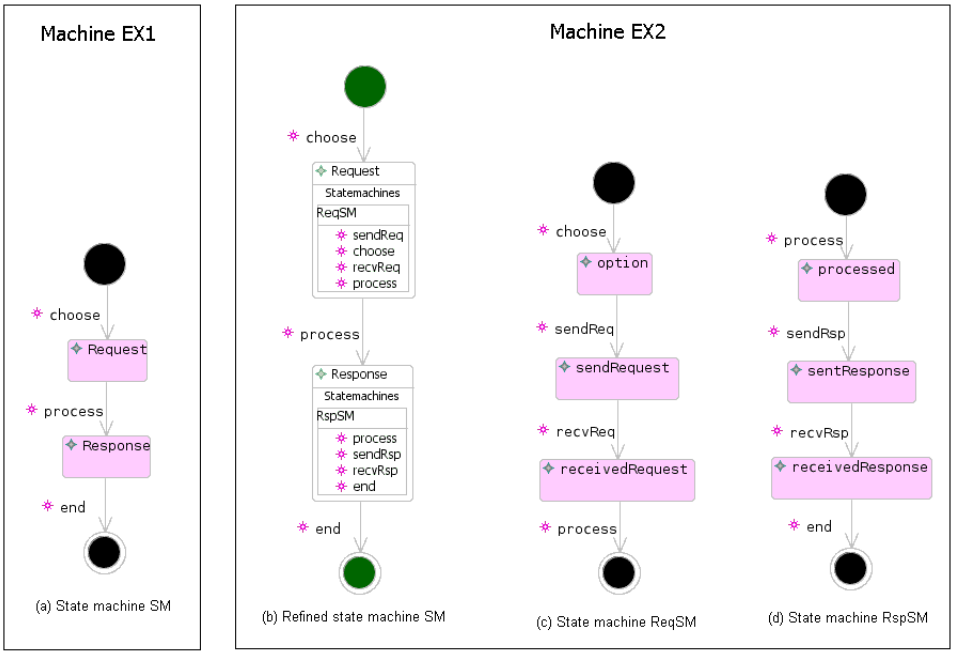


FIGURE 7.2: Example of State Machine Refinement (Nested State Machine)

7.2 Flattening State Machines

This section introduces the technique of state machine flattening in order to decompose a UML-B machine. This technique is introduced as part of a decomposition in order to use the state grouping technique later.

A UML-B model may consist of several machines that are linked by a refinement relationship. As described in Section 4.4, a state machine refinement may be performed by elaborating the structure of a state machine to have nested state machines in any of the super-states. This means, there are a number of nested state machines in a particular machine level. Flattening a state machine means to refine the structure of a state machine that consists of nested state machines into a state machine without any nested state machines.

The techniques of flattening and grouping are first described based on a simple example of a client-server system. The UML-B model consists of four machines where *EX1* is refined by *EX2*, *EX2* is refined by *EX3* and *EX3* is refined by *EX4*. In the example, the flattening of state machines is done at the third machine level i.e., machine *EX3*. State grouping is performed at the fourth machine level i.e., machine *EX4* and will be described in Section 7.3. The state machines of machines *EX1* and *EX2* are shown in Figure 7.2. We describe first an example of flattening state machines in UML-B. We will then describe the general rules.

Figure 7.2(a) is the state machine *SM* of a machine *EX1*. The state machine *SM* consists of two states, *Request* and *Response* and three transitions *choose*, *process* and *end*. Machine *EX2* refines machine *EX1* by refining the state machine *SM*. The state machine refinement in machine *EX2* is done by extending the state machine hierarchy of *SM* into nested state machines in the states *Request* and *Response* (Figure 7.2(b)). Figure 7.2(c) is the nested state machine *ReqSM* contained in the state *Request*. Figure 7.2(d) is the nested state machine *RspSM* contained in the state *Response*.

Figure 7.3(a) shows the refined state machine *SM* of machine *EX3*. Machine *EX3* refines machine *EX2* by flattening the state machines of *EX2* (Figure 7.2(b,c,d)). By flattening, the super-states *Request* and *Response* of the state machine *SM* are removed together with their nested state machines *ReqSM* and *RspSM* respectively. The sub-states of the super-states *Request* and *Response* are lifted to the top level. The transitions between the lifted sub-states also are lifted together with the states to the top level. The target state of the incoming transition *choose* to the super-state *Request* is replaced by the sub-state *option*. The source state of the outgoing transition *process* from the super-state *Request* is replaced by the sub-state *receivedRequest*. The target state of the incoming transition *process* to the super-state *Response* is replaced by the sub-state *processed*. The source state of the outgoing transition *end* from the super-state *Response* is replaced by the sub-state *receivedResponse*.

In this paragraph, we give an informal general definition of the flattening technique. Later in this section we define the technique more formally. The structure of a hierarchy of state machines may be flattened by:

- removing the super-state structures and lifting their sub-states and the transitions between the sub-states to the top level.
- the target state of each incoming transition to the super-state is replaced by the sub-state which has the corresponding incoming transition in the abstract machine.
- the source state of each outgoing transition from the super-state is replaced by the sub-state which has the corresponding outgoing transition in the abstract machine.

The flattening technique has been automated in the UML-B tool. The simple model of Figure 7.2 and 7.3 has been modelled and proved using the Rodin provers. All the proof obligations are proved. This is because in Event-B terms, a super-state is redundant w.r.t. its sub-states as it is a union of all sub-states. For example, the following predicate

$$Request = option \cup sentRequest \cup receivedRequest$$

specifies that the state *Request* is a union of the sub-states *option*, *sentRequest* and *receivedRequest*. Thus, the flattening technique is a valid refinement. Our work deals

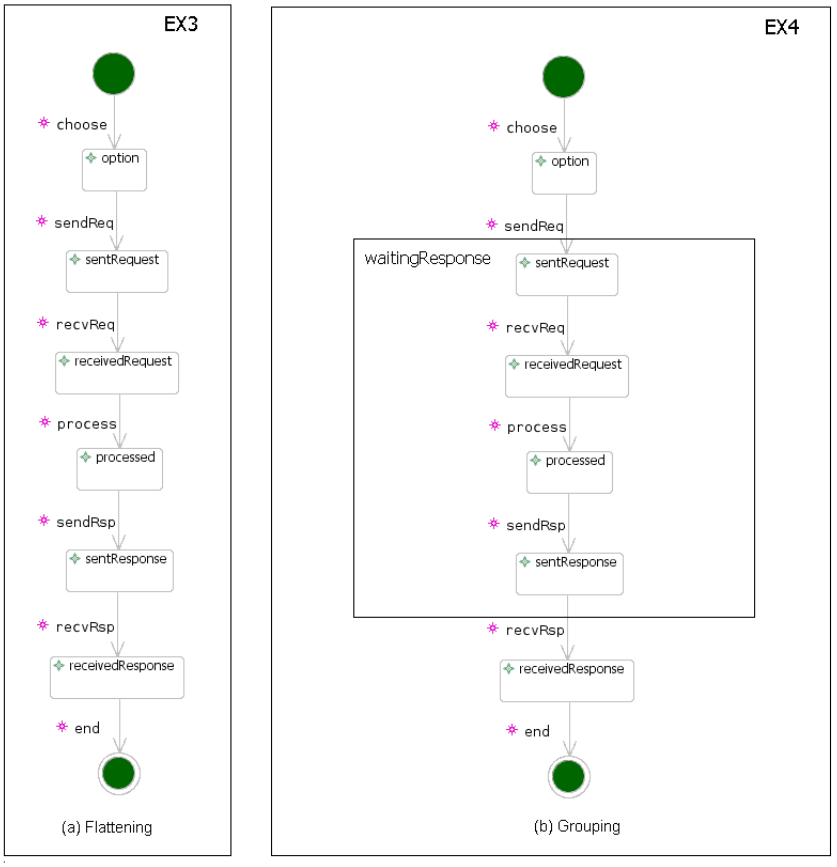


FIGURE 7.3: Example of Refinement by Flattening and Grouping State Machines

with the state set representation and does not deal with the state function representation of state machines translation. In future, the use of this technique with the state function representation will be explored.

7.2.1 Formal Definition of Flattening

In this section, we define the flattening technique using Event-B notation. We refer to the state machine model to be flattened as *M1* and the resulting flattened state machine as *M2*.

First, we define the data structures to be used in both *MI* and *M2*. Let *ELEMENT* be the set of UML-B elements. UML-B elements include machines, classes and state machines (including states and transitions).

STATES represents the set of states in both *M1* and *M2* and it is a subset of *ELEMENT*:

$$STATES \subseteq ELEMENT$$

The set *STATES* is partitioned into three subsets. The subset *initial* represents the initial states, *final* represent the final states and *normal* represent the states which neither initial nor final:

$$partition(STATES, initial, final, normal)$$

TRANSITIONS represents the set of transitions in both *M1* and *M2* and it is a subset of *ELEMENT*:

$$TRANSITIONS \subseteq ELEMENT$$

Next, we define the components of *M1* which are states, transitions and their containment relationships.

S1 represents the set of states of *M1*:

$$S1 \subseteq STATES$$

containmentS1 is the function which maps the states *S1* of *M1* into their container or owner of type *ELEMENT*:

$$containmentS1 \in S1 \rightarrow ELEMENT$$

Let s be the *S1* state whose sub-states are to be flattened:

$$s \in S1$$

In order for state s to be flattened it must be a UML-element that contains sub-states:

$$s \in ran(containmentS1)$$

$containmentS1(s)$ is the UML-B element in which the super-state s is itself contained.

$containmentS1^{-1}$ [{ s}] is the set of sub-states contained in s.

T1 represents the set of transitions of *M1*:

$$T1 \subseteq TRANSITIONS$$

The transitions *T1* are partitioned into two subsets; *elaborateT1* represents a set of elaborating transitions and *nonelaborateT1* represents a set of transitions which do not elaborate any transition:

$$partition(T1, elaborateT1, nonelaborateT1)$$

containmentT1 is the set which maps each transition of *M1* into its container of type *ELEMENT*:

$$containmentT1 \in T1 \rightarrow ELEMENT$$

transSourceStateT1 maps each transition of *M1* into its source state:

$$transSourceStateT1 \in T1 \rightarrow S1$$

transTargetStateT1 maps each transition of *M1* into its target state:

$$transTargetStateT1 \in T1 \rightarrow S1$$

elaborateOutgoingT1 maps elaborating transitions of *M1* into the outgoing transitions of super-states:

$$elaborateOutgoingT1 \in elaborateT1 \rightarrowtail T1$$

elaborateIncomingT1 maps the elaborating transitions of *M1* into the incoming transitions of super-states:

$$elaborateIncomingT1 \in elaborateT1 \rightarrowtail T1$$

Note: The functions *elaborateOutgoingT1* and *elaborateIncomingT1* are partial and injective. They are injective because each elaborating transition can elaborate at most one parent transition. They are partial because the domains contain both incoming and outgoing elaborating transitions. If an elaborating transition *t'* elaborates the transition *t* as both incoming and outgoing, this means *t* is a self loop transition. In this case, the domains of the functions are not disjoint. Otherwise, the domains are disjoint.

Elabs is the set of transitions of the super-state *s* which elaborate the parent transitions:

$$Elabs = containmentT1^{-1}\ [\{\ s\}\ ] \cap elaborateT1$$

NonElabs is the set of transitions of the super-state *s* which are not elaborating any parent transition:

$$NonElabs = containmentT1^{-1}\ [\{\ s\}\ ] \cap nonelaborateT1$$

Now that we have defined the components of *M1*, we now define the components of *M2*, representing the result of flattening super-state s in *M1*.

S2 represents the set of states of *M2*:

$$S2 \subseteq STATES$$

containmentS2 maps each state of *M2* into its container:

$$containmentS2 \in S2 \rightarrow ELEMENT$$

T2 represents the set of transitions of *M2*:

$$T2 \subseteq TRANSITIONS$$

The transitions *T2* are partitioned into two subsets; *elaborateT2* represents a set of elaborating transitions and *nonelaborateT2* represents a set of transitions which do not elaborate any transition:

$$partition(T2,elaborateT2,nonelaborateT2)$$

containmentT2 maps each transition of *M2* into its container.

$$containmentT2 \in T2 \rightarrow ELEMENT$$

transSourceStateT2 maps each transition of *M2* into its source state.

$$transSourceStateT2 \in T2 \rightarrow S2$$

transTargetStateT2 maps each transition of *M2* into its target state:

$$transTargetStateT2 \in T2 \rightarrow S2$$

elaborateOutgoingT2 maps elaborating transitions of *M2* into the outgoing transitions of super-states:

$$elaborateOutgoingT2 \in elaborateT2 \rightarrowtail T2$$

elaborateIncomingT1 maps elaborating transitions of *M2* into the incoming transitions of super-states:

$$elaborateIncomingT2 \in elaborateT2 \rightarrowtail T2$$

The rules defining the elements of *M2* components are as follows:

Rule 1: States of *M2*

M2 has all the states of *M1* but excluding the super-state s and excluding the initial and final sub-states of s:

$$S2 = (S1 \setminus \{\ s\}\ ) \setminus (containmentS1^{-1}\ [\{\ s\}\ ] \setminus normal)$$

Rule 2: Containment of the states of *M2*

The states containment *containmentS2* of *M2* has all the elements of *containmentS1* but excluding the elements associated with the super-state s. In addition, the container of s becomes the container of each normal sub-state of s:

$$\begin{aligned} containmentS2 = {} & (\{s\} \lhd containmentS1 \rhd \{s\}) \cup \\ & ((containmentS1^{-1}[\{s\}] \cap normal) \times \{containmentS1(s)\}) \end{aligned}$$

Rule 3: Transitions of *M2*

M2 has all the transitions of *M1* but excluding the elaborating transitions of the super-state s:

$$T2 = T1 \setminus Elabs$$

Rule 4: Containment of the transitions of *M2*

The transitions containment *containmentT2* has all the elements of *containmentT1* but excluding the elements associated with the elaborating transitions of s. Also, for each transition of *containmentT1* whose container is the super-state s, its container is changed to container of s:

$$containmentT2 = (Elabs \lhd containmentT1) \lhd\!\!+ (\ NonElabs \times \{\ containmentS1(s)\}\ )$$

Rule 5: Source states of *M2* transitions

The source states of each transition of *M2* are the same as those of *M1* but excluding the elaborating transitions of the super-state s. In addition, in *M2*, each of the transition t whose source state is s (i.e., t is an outgoing transition of s), the source state of t is replaced by the source state of the sub-transition which elaborates t:

$$\begin{aligned} transSourceStatesT2 = {} & (Elabs \lhd transSourceStatesT1) \lhd\!\!+ \\ & \{\ t\cdot\ t \in T1 \land transSourceStatesT1(t) = s\ \mid \\ & (\ t \mapsto (transSourceStatesT1(elaborateOutgoingT1^{-1}\ (t))))\} \end{aligned}$$

Note: From well definedness of UML-B model,

$$transSourceStatesT1(t) = s \Rightarrow t \in dom(elaborateOutgoingT1^{-1})$$

Rule 6: Target states of *M2* transitions

The target states of each transition of *M2* are the same as those of *M1* but excluding the elaborating transitions of the super-state s. In addition, in *M2*, each of the transition t whose target state is the super-state s (i.e., t is an incoming transition of s), the target state of t is replaced by the target state of the sub-transition which elaborates t:

$$\begin{aligned} transTargetStatesT2 = (Elabs \lhd transTargetStatesT1) \lhd\!\!+ \\ \{\ t\cdot\ t \in T1 \land t \in \land\ transTargetStatesT1(t) = s \mid \\ (\ t \mapsto (transTargetStatesT1(elaborateIncomingT1^{-1}\ (t))))\} \end{aligned}$$

Note: From well definedness of UML-B model,

$$transTargetStatesT1(t) = s \Rightarrow t \in dom(elaborateIncomingT1^{-1})$$

Rule 7: Elaborating transitions of *M2*

The elaborating transitions of *M2* are all the elaborating transitions of *M1* but excluding the elaborating transitions of the super-state s:

$$elaborateT2 = elaborateT1 \setminus Elabs$$

Rule 8: Non-elaborating transitions of *M2*

The transitions of *M2* which do not elaborate any transition are all the non-elaborating transitions of *M1*:

$$nonelaborateT2 = nonelaborateT1$$

Rule 9: Outgoing elaborating transitions of *M2*

The outgoing elaborating transitions of *M2* are the outgoing elaborating transitions of *M1* but excluding the elaborating transitions of s:

$$elaborateOutgoingT2 = Elabs \lhd elaborateOutgoingT1$$

Rule 10: Incoming elaborating transitions of *M2*

The incoming elaborating transitions of *M2* are the incoming elaborating transitions of *M1* but excluding the elaborating transitions of s:

$$elaborateIncomingT2 = Elabs \lhd elaborateIncomingT1$$

7.3 State Grouping

This section introduces the technique of state grouping in the refinement chain preceding a machine decomposition. This technique may be used after flattening a state machine. Grouping states means adding a new structure (state) in a state machine and nesting some of its states within the new structure.

We describe first an example of grouping states in UML-B. We will then describe the general rules. Figure 7.3(b) shows an example of grouping the flattened state machine in Figure 7.3(a). In the figure, a new super-state *waitingResponse* is added and the refined states *sentRequest*, *receivedRequest*, *processed* and *sendResponse* together with the transitions between them are grouped and nested in the state machine of the state *waitingResponse*. The target state *sentRequest* of the incoming transition *sendReq* is replaced with the new super-state *waitingResponse*. In the nested state machine of the super-state, the state *sentRequest* has an incoming transition which elaborates the transition *sendReq* and initiates from an initial state. The source state *sentResponse* of the outgoing transition *recvRsp* is replaced with the new super-state. In the nested state machine of the super-state, the state *sentResponse* has an outgoing transition which elaborates the transition *recvRsp* and terminates at a final state.

The grouping technique is used to facilitate the decomposition by partitioning the states of a requesting component, i.e., the client, the states of a middleware and the states of a responding component i.e., the server. The states which are grouped in the state *waitingResponse* are the states of middleware and responding component. The states which are not in the new state *waitingResponse* are the states of the requesting component.

In this paragraph, we give an informal general definition of the grouping technique. Later in this section we define the technique more formally. The structure of a state machine may be grouped by:

- adding a super-state structure and bringing some states and the transitions between them to a lower level by nesting them in the super-state structure.
- the target state of each incoming transition (the source state of the transition is outside the super-state) to the sub-state which is nested in the new super-state structure is replaced by the super-state. The sub-state has an incoming transition (the source state of the transition is an initial state) which elaborates the corresponding incoming transition to the super-state.
- the source state of each outgoing transition (the target state of the transition is outside the super-state) from the states which are nested in the new super-state structure is replaced by the super-state. The sub-state has an outgoing transition (the target state of the transition is a final state) which elaborates the corresponding outgoing transition to the super-state.

The grouping technique has been automated in the UML-B tool. The simple model of Figure 7.3 has been modelled and proved using the Rodin provers. All the proof obligations are proved. This is because in Event-B terms, adding a super-state structure is a superposition refinement [13]. Superposition refinement allows a development of a system by incrementally adding new requirements in refinements. In Event-B, adding new variables and (or) new events which modify only new variables is a superposition refinement. Thus, the grouping technique is a valid refinement. Similar to the flattening technique, at the moment the grouping technique can be used with the state set representation. The use of grouping technique with the state function representation will be explored in future.

The refined state machine in Figure 7.3(a) has a different structure from its abstract state machine (Figure 7.2(b)). Similarly the refined state machine in Figure 7.3(b) has a different structure from its abstract state machine (Figure 7.3(a)). This is contrary to the hierarchical refinement where the structure of the abstract state machine is retained in a refinement. However, refactoring a state machine by removing a super-state structure of an abstract state machine and lifting the sub-states to a top level in a refinement can facilitate the decomposition of UML-B machines. Similarly, adding a super-state structure and nesting some states at the top level in the super-state structure can facilitate the decomposition of UML-B machines. In future, we will enforce in the metamodel that a structure of abstract state machines can be refactored in the refinement only when using the flattening and grouping techniques.

The movement of the transition *process* to a different hierarchy as in Figure 7.3(b) illustrates why re-grouping is required. Hierarchical refinement (Section 4.4) would not give rise to this structuring from Figure 7.2(b). Is the refinement using the flattening or grouping technique a bad idea? No, because it facilitates easy refinement, and flattening and grouping have a negligible proof overhead. For the example, all the proof obligations (POs) are discharged automatically when using the flattening technique. When using the grouping technique, only two from sixteen POs are manually discharged. These two POs are discharged by manually rewriting the partition invariants in the hypotheses and goal. These proving steps are simple and could be easily automated in the prover.

7.3.1 Formal Definition of Grouping

In this section, we define the grouping technique using Event-B notation. Let us refer a state machine model to be grouped as *M1* and the resulting grouped state machine as *M2*.

The sets defined in Section 7.2 of the flattening technique are used in defining the elements of *M2* for grouping technique.

First, we assume the following states and transitions of *M1* to be grouped:

Let *subStates* be the states of *M1* to be grouped:

$$subStates \subseteq S1$$

Let *container* be of type *ELEMENT*:

$$container \in ELEMENT$$

and let all the elements of *subStates* share the same parent *container*:

$$containmentS1[subStates] = container$$

Let *subTransitions* be the transitions of *M1* to be grouped:

$$subTransitions \subseteq T1$$

The set *subTransitions* contains the transitions whose source and target states are the elements of the set *subStates*.

$$subTransitions = \{t | t \in T1 \wedge transSourceStateT1(t) \in subStates \wedge transTargetStateT1(t) \in subStates\}$$

We now define the components of *M2*, the result of grouping the states *subStates* in *M1*.

When grouping, a new super-state will be added in *M2*. Let assume that *ns* is the new super-state of *M2* i.e., *ns* is a 'fresh' state and not an element of *S1*:

$$ns \notin S1$$

The *container* (i.e., the parent of *subStates* in *M1*) is defined to be the container of *ns*.

When grouping, the *subStates* and *subTransitions* are nested in the new super-state *ns*. Additionally, a number of initial and final states, and their associated transitions will be added in *ns*. In the following paragraph, we define these initial states, final states, outgoing transitions of the initial states and incoming transitions of the final states:

Let *subInitialFinalStates* be the set of initial and final states of the super-state *ns*:

$$subInitialFinalStates \subseteq STATES$$

The set *subInitialFinalStates* contains a set of 'fresh' states and they are disjoint from *S1*:

$$subInitialFinalStates \cap S1 = \{\}$$

The states of *subInitialFinalStates* are partitioned into *subInitialSt* and *subFinalSt*. *subInitialSt* is the set of initial states of *ns*. *subFinalSt* is the set of final states of *ns*:

$$partition(subInitialFinalStates, subInitialSt, subFinalSt)$$

Let *inTransNs* be the set of all incoming transitions into the new super-state *ns*, i.e., those transitions of *M1* whose source state is not in *subStates* and whose target state is in *subStates*:

$$inTransNs = \{\ t \mid t \in T1 \wedge transSourceStateT1(t) \notin subStates \wedge transTargetStateT1(t) \in subStates\}$$

Let *outTransNs* be the set of all outgoing transitions from the new super-state *ns*, i.e., those transitions of *M1* whose target state is not in *subStates* and whose source state is in *subStates*.

$$outTransNs = \{\ t \mid t \in T1 \wedge transTargetStateT1(t) \notin subStates \wedge transSourceStateT1(t) \in subStates\}$$

The number of initial states of *ns* corresponds to the number of incoming transitions into *ns*:

$$card(subInitialSt) = card(inTransNs)$$

The number of final states corresponds to the number of outgoing transitions from *ns*:

$$card(subFinalSt) = card(outTransNs)$$

Let *elabTransNs* be the transitions of *ns* which elaborate parent transitions (i.e., the incoming and outgoing transitions of *ns*):

$$elabTransNs \subseteq TRANSITIONS$$

The set *elabTransNs* contains a set of 'fresh' elaborating transitions and they are disjoint from *T1*:

$$elabTransNs \cap T1 = \{\}$$

The elaborating transitions of *ns* are partitioned into two sets. *subInTransNs* is the set of incoming transitions and *subOutTransNs* is the set of outgoing transition:

$$partition(elabTransNs, subInTransNs, subOutTransNs)$$

Each incoming elaborating transition in the set *subInTransNs* elaborates one incoming parent transition in the set *inTransNs*:

$$elaborateInNs \in subInTransNs \rightarrowtail\!\!\!\!\twoheadrightarrow inTransNs$$

Each outgoing elaborating transition in the set *subOutTransNs* elaborates one outgoing parent transition in the set *outTransNs*:

$$elaborateOutNs \in subOutTransNs \rightarrowtail\!\!\!\!\twoheadrightarrow outTransNs$$

Each incoming elaborating transition in the set *subInTransNs* has a distinct source state from the set *subInitialSt*:

$$subInTransSourceStateNs \in subInTransNs \rightarrowtail\!\!\!\!\twoheadrightarrow subInitialSt$$

Each outgoing elaborating transition in the set *subOutTransNs* has a distinct target state from the set *subFinalSt*:

$$subOutTransTargetStateNs \in subOutTransNs \rightarrowtail\!\!\!\!\twoheadrightarrow subFinalSt$$

The rules defining the elements of the resulting regrouped model *M2* are as follows.

Rule 1: States of *M2*
M2 has all the states of *M1*, the super-state *ns* and a number of initial and final sub-states:

$$S2 = S1 \cup \{ ns\} \cup subInitialFinalStates$$

Rule 2: Containment of the states of *M2*
The states containment of *M2* has all the elements of *containmentS1* but the container of each regrouped states *subStates* is changed to the new super-state *ns*. *containmentS2* also has *ns* and its container as its element. The other elements of *containmentS2* are the initial and final sub-states associated with their container *ns*:

$$\begin{aligned} containmentS2 = {} & (containmentS1 \mathbin{\lhd\!\!\!-} (subStates \times \{ ns\})) \cup \\ & \{ ns \mapsto container\} \cup (subInitialFinalStates \times \{ ns\}) \end{aligned}$$

Rule 3: Transitions of *M2*
M2 has all the transitions of *M1* as well as the elaborating transitions of the new state *ns*:

$$T2 = T1 \cup elabTransNs$$

Rule 4: Containment of the transitions of *M2*

The transitions containment *containmentT2* of *M2* has all the elements of *containmentT1* but the container of each regrouped transition *subTransitions* becomes the new super-state *ns*. Also, *containmentT2* includes the containment elements of the elaborating transitions whose container are *ns*:

$$\begin{aligned} containmentT2 = (containmentT1 \lhd (subTransitions \times \{\ ns\}\ )) \cup \\ (elabTransNs \times \{\ ns\}\ ) \end{aligned}$$

Rule 5: Source states of *M2* transitions

The source states of the transitions of *M2* has all the elements of *transSourceStateT1*. But the source states of the outgoing transitions *outTransNs* are *ns*. Additionally, *transSourceStateT2* includes the elements that associate each new incoming sub-transitions of *ns, subInTransNs* with an initial state as its source state.

$$\begin{aligned} transSourceStateT2 = (transSourceStateT1 \lhd\!\!\!- outTransNs \times \{\ ns\}) \cup \\ subInTransSourceStateNs \end{aligned}$$

Rule 6: Target states of *M2* transitions

The source states of the transitions of *M2* has all the elements of *transTargetStateT1*. But the target states of the incoming transitions *inTransNs* are *ns*. Additionally, *transTargetStateT2* includes the elements that associate each new outgoing sub-transitions of *ns, subOutTransNs* with a final state as its target state.

$$\begin{aligned} transTargetStateT2 = (transTargetStateT1 \lhd\!\!\!- inTransNs \times \{\ ns\}) \cup \\ subOutTransSourceStateNs \end{aligned}$$

Rule 7: Elaborating transitions of *M2*

The elaborating transitions of *M2* are all the elaborating transitions of *M1* and including the elaborating transitions of the super-state *ns*:

$$elaborateT2 = elaborateT1 \cup elabTransNs$$

Rule 8: Non-elaborating transitions of *M2*

The transitions of *M2* which do not elaborate any transition are all the non-elaborating transitions of *M1*:

$$nonelaborateT2 = nonelaborateT1$$

Rule 9: Outgoing elaborating transitions of *M2*

The outgoing elaborating transitions of *M2* are the outgoing elaborating transitions of *M1* and including the outgoing elaborating transitions of *ns*:

$$elaborateOutgoingT2 = elaborateOutgoingT1 \cup elaborateOutNs$$

Rule 10: Incoming elaborating transitions of *M2*
The incoming elaborating transitions of *M2* are the incoming elaborating transitions of *M1* and including the incoming elaborating transitions of *ns*:

$$elaborateIncomingT2 = elaborateIncomingT1 \cup elaborateInNs$$

An alternative formalisation would be to model the structures in Section 7.2.1 and 7.3.1 with machine variables and the flattening and grouping (i.e., the elements of the *M2* components) as events. The constraints could be modelled as invariants giving rise to proof obligations for verifying that the flattening and grouping maintain the constraints. This approach could be investigated in future work.

7.4 Composed Machines in Event-B

The extension to Event-B to support event-based composition and decomposition has been introduced by Silva and Butler [89, 90]. They have introduced the *composed machine* notion. A plug-in extension to the Rodin to support the composed machine notion has been developed. Based on this Event-B composed machine notion, the notion of composed machine and related notions are introduced in UML-B and are described in Section 7.5.

The composed machine notion allows a number of Event-B machines to be composed. The composed machine may be used in an Event-B development as a refinement of an abstract machine. Figure 7.4 shows an example of the structure of a composed machine named *CM*.

```
COMPOSED MACHINE CM
REFINES M
INCLUDES
        mC1
        mC2
INVARIANT inv1
EVENTS
        ev refines M.ev
                Combines Events mC1.ev || mC2.ev
END
```

FIGURE 7.4: A Structure of Event-B Composed Machine

The composed machine has six clauses. The *refines* clause indicates a more abstract machine which is refined by a composed machine. The *includes* clause indicates the

machines which are being composed. For the included machine, there is an *invariant option* property which allows the modeller either to include its invariants or not. The *invariant* clause contains the invariants of the composed machine. The invariants can refer to variables of all included machines. The *events* clause represents the events of the composed machine which are being composed from the included machines and they must refine all the events of the abstract machine. For each of these events, the *combines events* clause needs to be defined. The *combines events* clause indicates which event(s) of the included machine(s) is(are) being composed.

In Figure 7.4 the composed machine *CM* refines the machine *M*. The composed machine *CM* includes the two machines *mC1* and *mC2*. It has only one composed event *ev* and this event refines the abstract event *ev* of machine *M*. The event is defined by combining the events *ev* of the included machine *mC1* and *ev* of the included machine *mC2*. Combining two events means forming a single event from the guards and actions of both events.

7.5 Composed Machine, Included Machine, Composed Event and Constituent Event

This section introduces the notions of composed machine, included machine, composed event and constituent event into UML-B. The motivation for introducing these notions comes from performing event-based decomposition in Event-B [24] as described in Section 7.4.

In general, a *composed machine* is similar to a UML-B machine in terms of its relationships with other UML-B elements. In particular, a composed machine may refine a machine, may see a number of contexts and it may contain invariants and theorems. In contrast to a machine, a composed machine may include a number of included machines and a number of composed events. An *included machine* represents some machines in the UML-B model.

The *composed events* of a composed machine are a composition of transitions that are contained in all the included machines. Each composed event *refines* a transition of the machine being refined by the composed machine. A composed event may contain a number of constituent events. A *constituent* event is a transition of an included machine.

We describe an example of modelling a composed machine. The following example is based on the client-server system in Section 7.2 and in particular the machine *EX4* in Figure 7.3(b). We want to decompose the machine *M*4 into three machines which model the client, middleware and server components. Figure 7.5 shows the partitioning of the states in the state machines of machine *EX4* . The states *option* and *receivedResponse* which are not grouped in the state *waitingResponse* are the states of the client component

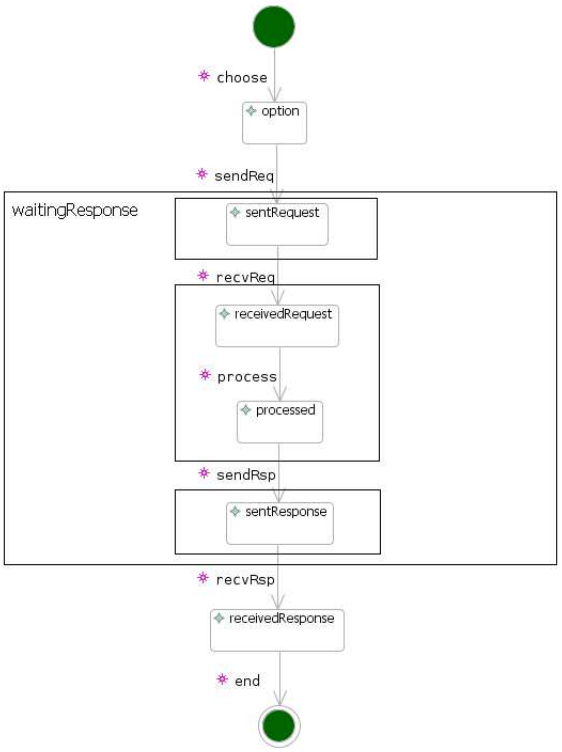
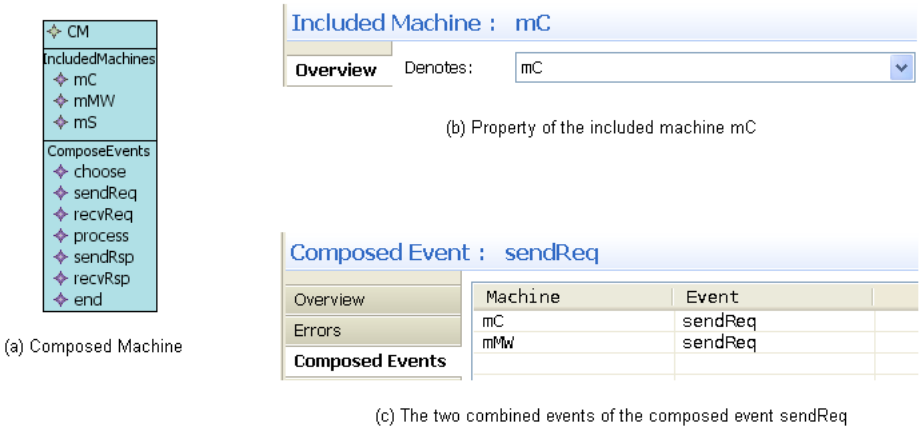


FIGURE 7.5: Partitioning of the State Machines for Decomposition

i.e., machine *mC*. The nested state machine in the state *waitingResponse* is partitioned into three state machines which are highlighted by three boxes. We want to place the states *sentRequest* and *sentResponse* in the middleware component i.e., machine *mMW* and into two state machines. The states *receivedRequest* and *processed* are the states of a state machine of the server component i.e., machine *mS*. Figure 7.5 has additional grouping compared with Figure 7.3(b).

Figure 7.6 shows a UML-B model for a composed machine. The composed machine *CM* (a) consists of three included machines: *mC*, *mS* and *mMW* and seven composed events: *choose*, *sendReq*, *recvReq*, *process*, *sendRsp*, *recvRsp* and *end*. Figure 7.6 (b) shows the *denotes* property for the included machine *mC*. This property indicates that the included machine is representing the machine *mC*. Figure 7.6 (c) shows a table listing the two constituent events of the composed event *sendReq*. The first row is the transition *sendReq* of *mC*. The second row is the transition *sendReq* of *mMW*.

The included machines denote the machine *mC*, *mS* and *mMW* respectively. The machine *mC* models the client component. The machine *mS* models the server component. The machine *mMW* is the *middleware* component which synchronizes the communication between the client and server components. The UML-B models of these machines are shown in Figure 7.7, 7.8 and 7.9. These state machines are derived by partitioning the state machine in Figure refgroupingDecom.



(a) Composed Machine

(b) Property of the included machine mC

(c) The two combined events of the composed event sendReq

FIGURE 7.6: UML-B Model of the Composed Machine

Figure 7.7 contains the state machine of a client component showing its states and transitions. The states and transitions of a server component can be seen in Figure 7.8. Figure 7.9 shows the two state machines of the middleware component. The transition *sendReq* of the middleware is synchronized with the transition *sendReq* of the client component. The transition *recvReq* is synchronized with the transition *recvReq* of the server component. The middleware component also contains the transitions *sendRsp* and *recvRsp* for synchronization with the client and server components.

Figure 7.10 shows the Event-B specification for the event *sendReq*. Each guard and action has a label indicating its container machine. For example, the first guard is labelled as *mC/self.type* indicates that the guard is contained in the machine *mC*. The Event-B specification is generated by the Event-B composition plug-in. This plug-in is an extension made to the Rodin Event-B tool and supports the Event-B composed machine described in Section 7.4. The Event-B specification is generated from the Event-B composed machine. In this case, this Event-B composed machine is created by the UML-B composed machine *CM*.

7.6 Extending the UML-B Metamodel and Tool to Support Composition

This section describes the extension to the UML-B Version 2 metamodel to support composition. The extensions involve adding a number of metaclasses and the relationships between them and the existing metaclasses.

There are four metaclasses which are added to the metamodel namely *UMLBComposedMachine*, *UMLBIncludedMachine*, *UMLBComposedEvent* and *UMLBCombinedEvent*. Two enumerations are also added namely *extendClause* and *invariantOption*.

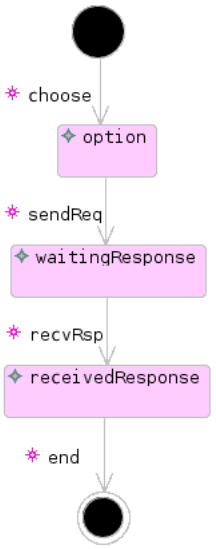


FIGURE 7.7: State Machine of the Client Component

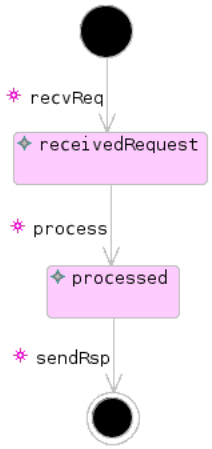


FIGURE 7.8: State Machine of the Server Component

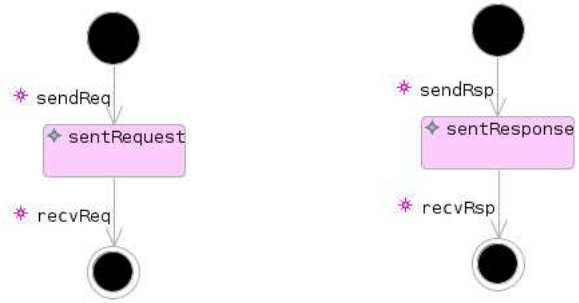


FIGURE 7.9: State Machines of the Middleware Component

```
sendReq   ≙
STATUS
  ordinary
REFINES
  sendReq
ANY
  self      //   contextual instance of class client
WHERE
  mC/self.type   :   self ∈ client
  mC/clientSM_isin_option   :   self ∈ option
  mMW/self.type   :   self ∈ clientM_SET \ clientM
THEN
  mC/clientSM_leaveState_option   :   option ≔ option \ {self}
  mC/clientSM_enterState_waitingResponse   :   waitingResponse ≔ waitingResponse ∪ {self}
  mMW/clientM_constructor   :   clientM ≔ clientM ∪ {self}
  mMW/req_SM_enterState_sentRequest   :   sentRequest ≔ sentRequest ∪ {self}
END
```

FIGURE 7.10: Generated Event-B Machine for Composed Event

Figure 7.11 shows part of the extension that focused on the metaclass *UMLBComposedMachine* and its relationship to other new metaclasses (*UMLBIncludedMachine* and *UMLBComposedEvent*) and the existing metaclasses (*UMLBMachine* and *UMLBContext*). The association *refines* from the metaclass *UMLBComposedMachine* to *UMLBMachine* means a composed machine may refine a machine. The association *contexts* with the metaclass *UMLBContext* means a composed machine may see many contexts. The containment *includedMachines* with the metaclass *UMLBIncludedMachine* means a composed machine may contain a number of included machines. The containment *composedEvents* with the metaclass *UMLBComposedEvent* means a composed machine may contain a number of composed events.

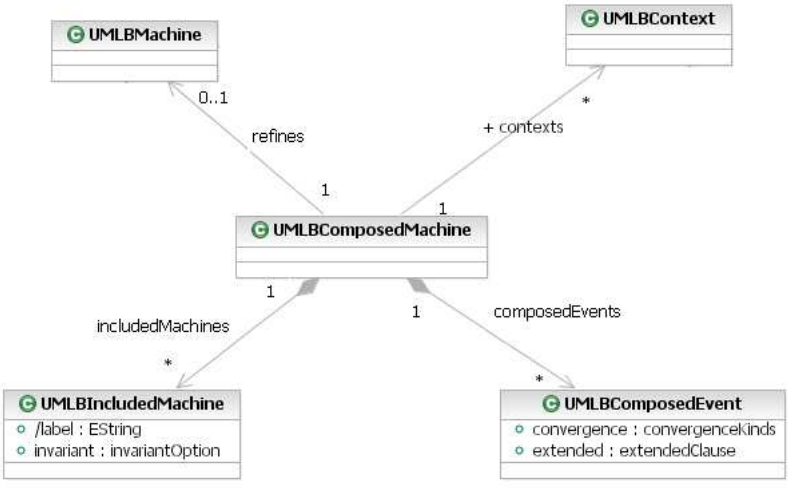


FIGURE 7.11: UML-B Metamodel Extensions for Composed Machine

Figure 7.12 shows part of the extension that focused on the metaclass *UMLBIncludedMachine* and its relationship to the existing metaclass *UMLBMachine*. The association *denotesMachine* specifies that an included machine must denotes a machine (Note that

UMLBIncludedMachine represents a placeholder for a reference to a machine, rather than an actual machine). The metaclass *UMLBIncludedMachine* has two attributes namely *label* and *invariant*. The attribute *label* is derived from the name of the machine being denoted. The attribute *invariant* is of type *invariantOption*. The type is given by the new enumeration *invariantOption* which has two enumeration literals namely, *no_invariant* for not including the invariants of the denoted machine and *include_invariant*, for including the invariants.

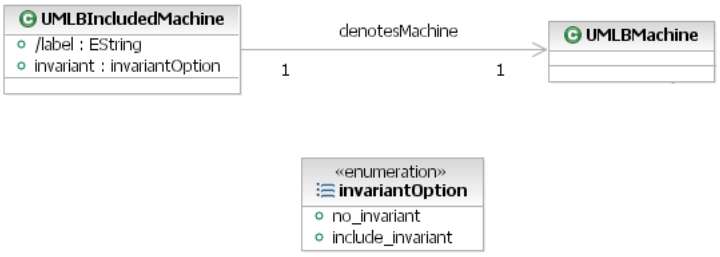


FIGURE 7.12: UML-B Metamodel Extensions for Included Machine

Figure 7.13 shows part of the extension that focused on the metaclass *UMLBComposedEvent* and its relationship with the new metaclass *UMLBCombinedEvent* and the existing metaclass *UMLBguardedAction*. The association *refines* with the metaclass *UMLBguardedAction* specifies that a composed event refines a number of events. The containment *combinedEvents* with the metaclass *UMLBCombinedEvent* specifies that a composed event may contain a number of constituent events. Similar to the metaclass *UMLBguardedAction*, the metaclass *UMLBcomposedEvent* has an attribute *convergence* which is either *ordinary*, *convergent* and *anticipated*. Another attribute of a composed event is *extended* which is either *not_extended*, for not extending an abstract event, or *extended* for extending an abstract event. Both the *extended* and *not_extended* options will retain the guards and actions of an abstract event in the refinement. The difference is, the *extended* option will not allow the guards and actions to be modified therefore, they are hidden in the Rodin tool editor. In contrast, the *not_extended* option will allow the guards and actions to be modified.

Figure 7.14 shows part of the extension that focused on the metaclass *UMLBCombinedEvent* and its relationship with the new metaclass *UMLBIncludedMachine* and the existing metaclass *UMLBguardedAction*. The metaclass *UMLBCombinedEvent* represents the constituent events of a composed event(Note that *UMLBCombinedEvent* represents a placeholder for a reference to an event, rather than an actual event). The association *includedmachine* specifies that a constituent event refers to an included machine. The association *denotesEvent* specifies that a constituent event denotes an event. The drawing tool (Section 7.7) restricts this event to be one of the events of the included machine.

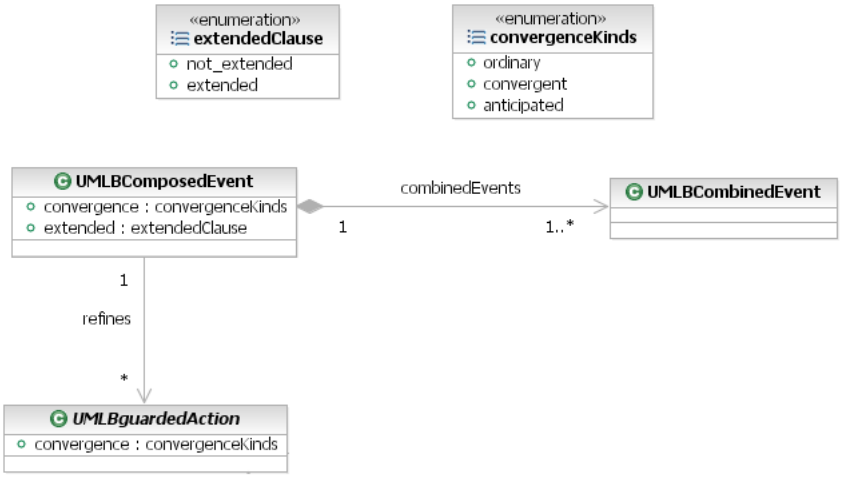


FIGURE 7.13: UML-B Metamodel Extensions for Composed Event

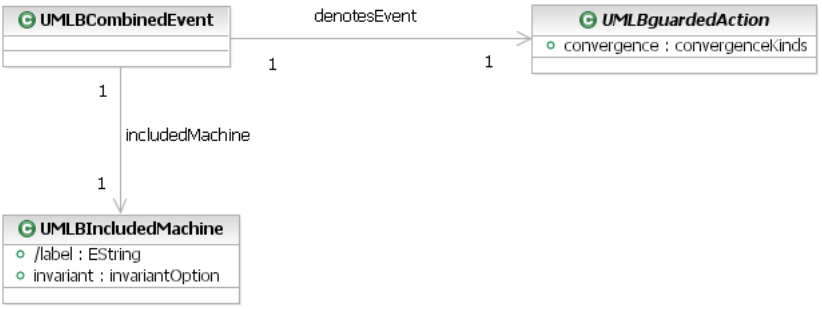


FIGURE 7.14: UML-B Metamodel Extensions for Combined Event

7.7 Extending the UML-B Drawing Tool

The drawing tool of the package diagram editor was extended to support composition. The extensions involved the following:

- adding figures for composed machine, included machines, composed events, sees links and refines links.
- adding creation tools which enable creation of the above figures and links.
- adding properties of the above figures.

Figure 7.15(a) shows a figure for composed machines which is a blue rectangle. Attached to it are included machines and composed events which are simply text labels. *Machine1* and *Machine2* are examples of included machines and *ev* is an example of composed event. Another added figure is the *sees* figure which is an arrow linking the composed

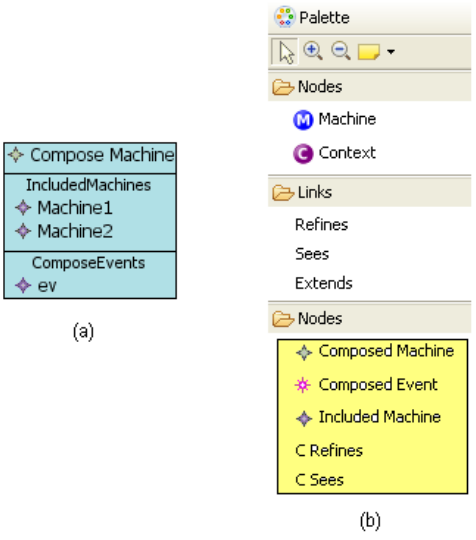


FIGURE 7.15: Drawing Tool Extensions for a Package Diagram Editor

machines and contexts. Also added is the *refines* figure which is an arrow linking the composed machines and machines.

The tool palette of UML-B is extended with additional creation tools for creating composed machines, composed events, included machines, sees links and refines links on the drawing canvas. The tool palette is shown in Figure 7.15(b). The yellow box highlights the new creation tools.

Each included machine has a *denotes* property. This property is specified by choosing from a combo box provided by the tool. The list contains all the machines in a UML-B package diagram. Each composed event has a *refines* property. This refines property can be chosen from a list box containing all the class events and transitions of the abstract machine. Each composed event is a combination of events (constituent events) of the included machines. These constituent events are added using a table.

7.8 Summary

This chapter has described the extensions to the UML-B language to support event-based composition and decomposition. The techniques of flattening and grouping state machines which facilitate decomposition in UML-B have been described. This chapter then introduced the notions of composed machines, included machines, composed events and combined events. These correspond to the notion of composed machine and its structures in Event-B. The extension to the UML-B metamodel which gives precise definitions of these notions are also described. The metamodel extensions were used to extend the UML-B drawing tool to support composition. The Event-B translator was

extended to support the flattening and grouping techniques. Chapter 8 evaluates the extensions to the metamodel in the ATM case study.

Chapter 8

Modelling The ATM Case Study in UML-B ((De)composition)

8.1 Introduction

This chapter contains a description of the ATM case study modelled using the UML-B Version 2 tool which incorporates the composition and decomposition. The purpose of the task is to validate the techniques of flattening state machines and state grouping defined in Chapter 7. Also, the case study is done to validate the notions of composed machine, included machine, composed event and constituent event in Chapter 7.

8.2 ATM Case Study: An overview

The UML-B development of the ATM system describes in this chapter is a continuation of the development in Chapter 6. There are thirteen machines (including decomposed machines) and one composed machine in the ATM UML-B development. The package diagram in Figure 8.1 shows the contexts, machines and a composed machine and their relationships. Chapter 6 describes the first five machines. This chapter describes the rest of the machines that involve composition and decomposition. The summaries for each machine are as follows:

Fifth Refinement (*ATM_R5*): Flattening the state machines hierarchies by removing the super-state structures.

Sixth Refinement (*ATM_R6*): Grouping states by adding a super-state structure and nesting some of the states in the super-state.

Seventh Refinement (*ATM_R7*): Introducing instances of ATM as a middleware.

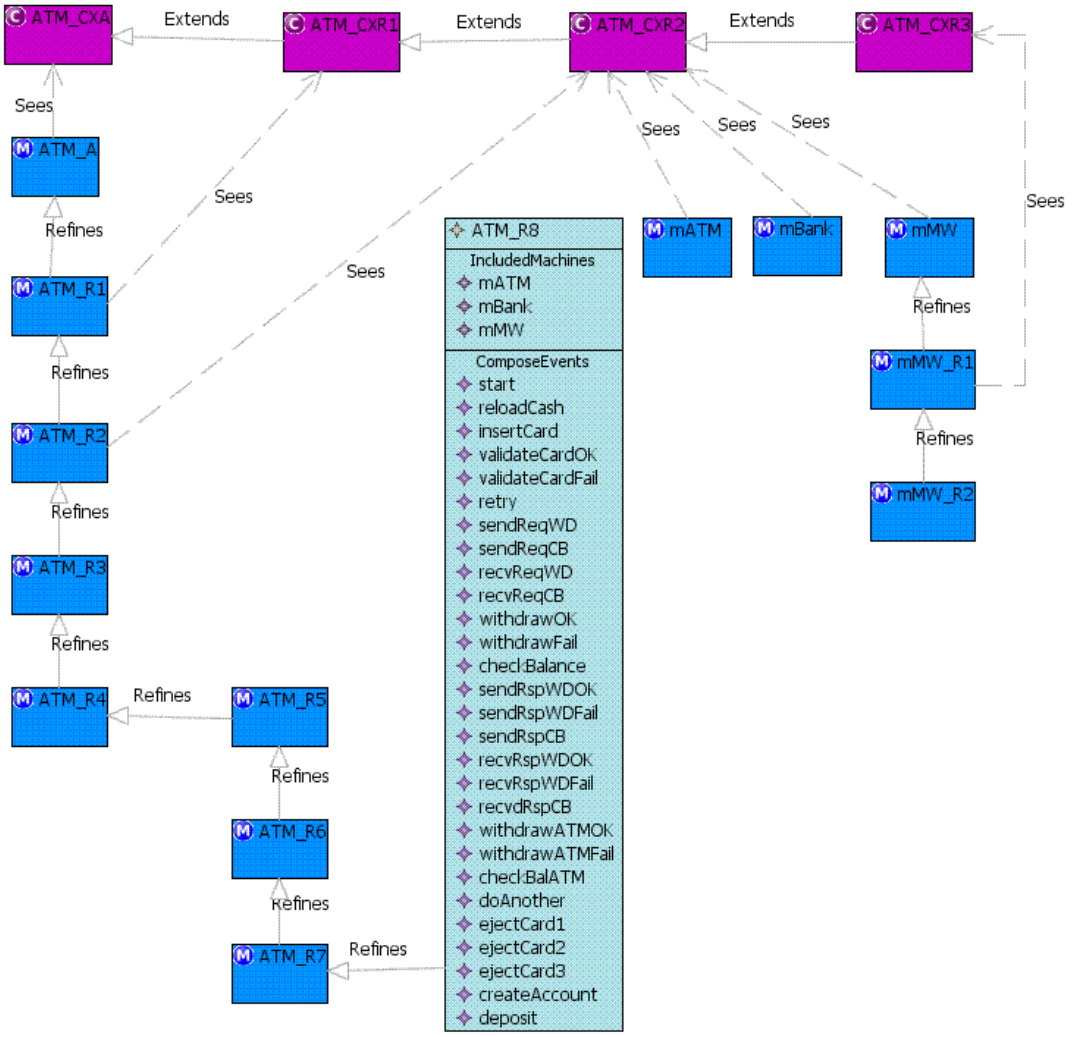


FIGURE 8.1: ATM Package Diagram

Eighth Refinement (*ATM_R8*): Introducing a composed machine that composes the machine *mATM*, *mBank* and *mMW*. The machines *mATM* and *mBank* are the decomposed machines based on the seventh refinement (*ATM_R7*).

Abstract ATM component (*mATM*): Models the behaviour of ATMs.

Abstract Bank component (*mBank*): Models the behaviour of a bank.

Abstract Middleware component (*mMW*): Models the behaviour of a middleware. The middleware sychronizes the sending and receiving communication between ATMs and the bank.

The machine *mMW_R1* introduces a form of communication using message passing via five channels corresponding to different requests and responses (withdrawal request, check balance request, successful withdrawal response, unsuccessful withdrawal response

and check balance response). The machine *mMW_R2* refines the five communication channels into one channel in order to reflect the actual implementation.

The fifth and sixth refinement are the refinement machines preceding the decomposition. In the Event-B development of Chapter 6, decomposition is not being modelled, therefore these two machine levels do not correspond to any of the Event-B refinements. Similarly, the seventh and eight refinements do not correspond to any of the Event-B refinements as they relate to decomposition. The machine *mMW_R1* which models the message passing partly corresponds to the fourth and the fifth refinements of the Event-B development in Sections 3.3.5 and 3.3.6. The middleware machine *mMW_R2* partly corresponds to the sixth refinement of the Event-B development in Section 3.3.7.

The generated Event-B specifications are in Appendix D.

8.3 Fifth Refinement

The fifth refinement changes the state machines of *ATM_R4* by flattening the nested state machines in the fourth refinement. This is done in order to systematically decompose events into events at an ATM and events at a bank. This refinement retains all the three classes of the machine *ATM_R4*. Which means the class diagram of *ATM_R5* consists of three refined classes *atm*, *atmB* and *account*. All these classes have the inherited attributes that inherit their corresponding classes in *ATM_R4*.

Figure 8.2 is the refined state machine *ATM_SM* that has been flattened. All the super-states and their nested state machines have been removed. The highest level of super-state in the hierarchy which has been removed is the super-state *active_atm*. The second level of super-states which are removed are *transOption* and *performTrans*. At the third level hierarchy, within the super-state *transOption*, the super-states *reqWD* and *reqCB* are removed. Within the super-states *performTrans*, the super-states *processedWDOK*, *processedWDFail* and *processedCB* are removed. The sub-states of these super-states are lifted to the top level. The transitions between the lifted sub-states are also lifted together with the states to the top level. The transitions refine the respective abstract transitions of *ATM_R4*. It can be seen from the figure that only the inner-most sub-states are retained with all the transitions. The target states of the incoming transitions to the super-states which are removed are replaced by the corresponding innermost sub-states. The source states of the outgoing transitions from these super-states are replaced by the innermost sub-states.

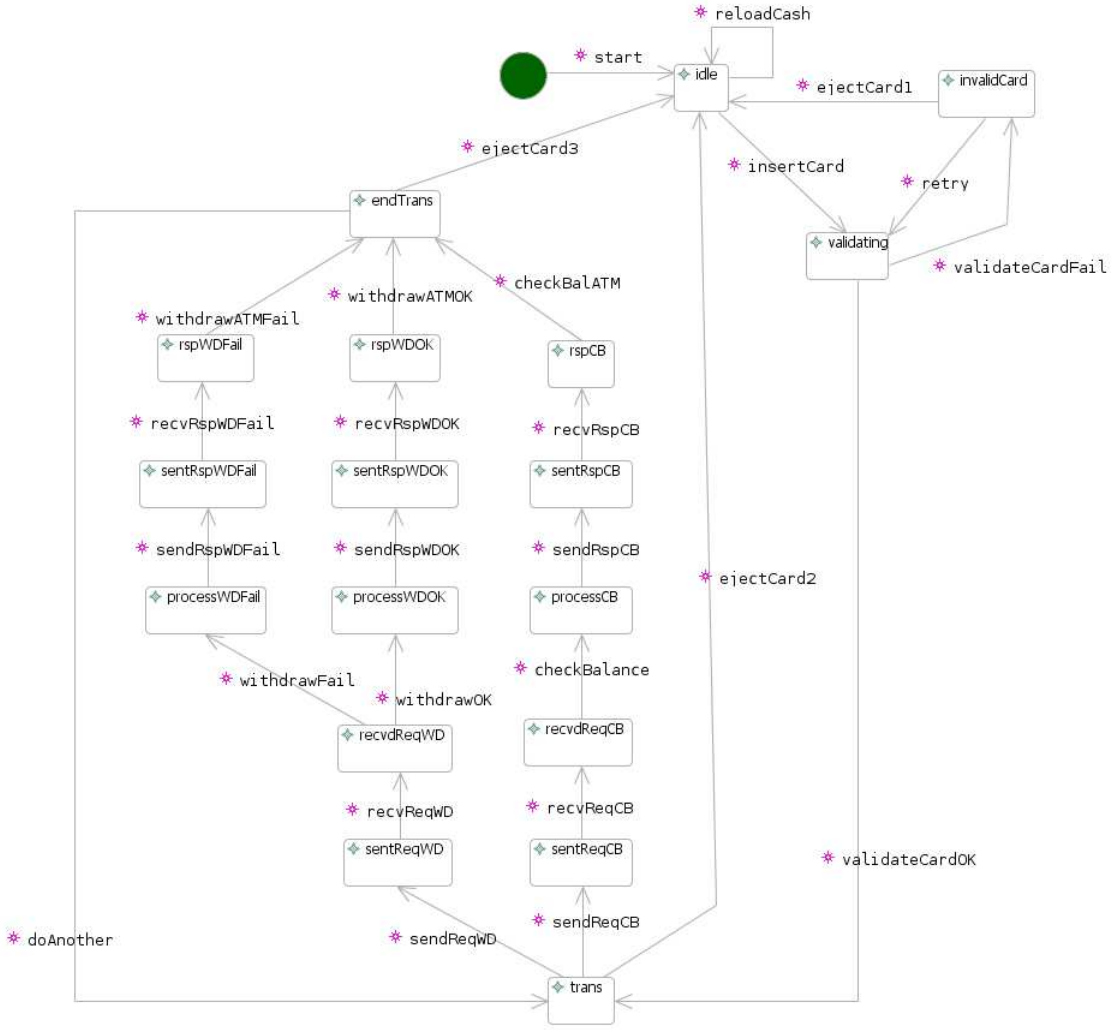


FIGURE 8.2: Flattening the State Machine ATM_SM in the Fifth Refinement

8.4 Sixth Refinement

Similar to the fifth refinement, this refinement does not introduces any new classes or new attributes. The sixth refinement refines the fifth refinement by using the state grouping technique. The grouping technique is used to partition the states and transitions between ATM, middleware and bank components. In this refinement, a new state *waitingResponse* is introduced in the refined state machine *ATM_SM*. The states and transitions at the top level hierarchy of the refined state machine *ATM_SM* including the new state *waitingResponse* are the states and transitions occur at ATMs. The states and transitions occur at the bank and middleware are nested in the state machine of the new state.

Figure 8.3 is the refined state machine *ATM_SM* showing the transitions, the refined states and the new state *waitingResponse*. Attached to the state *waitingResponse* is the

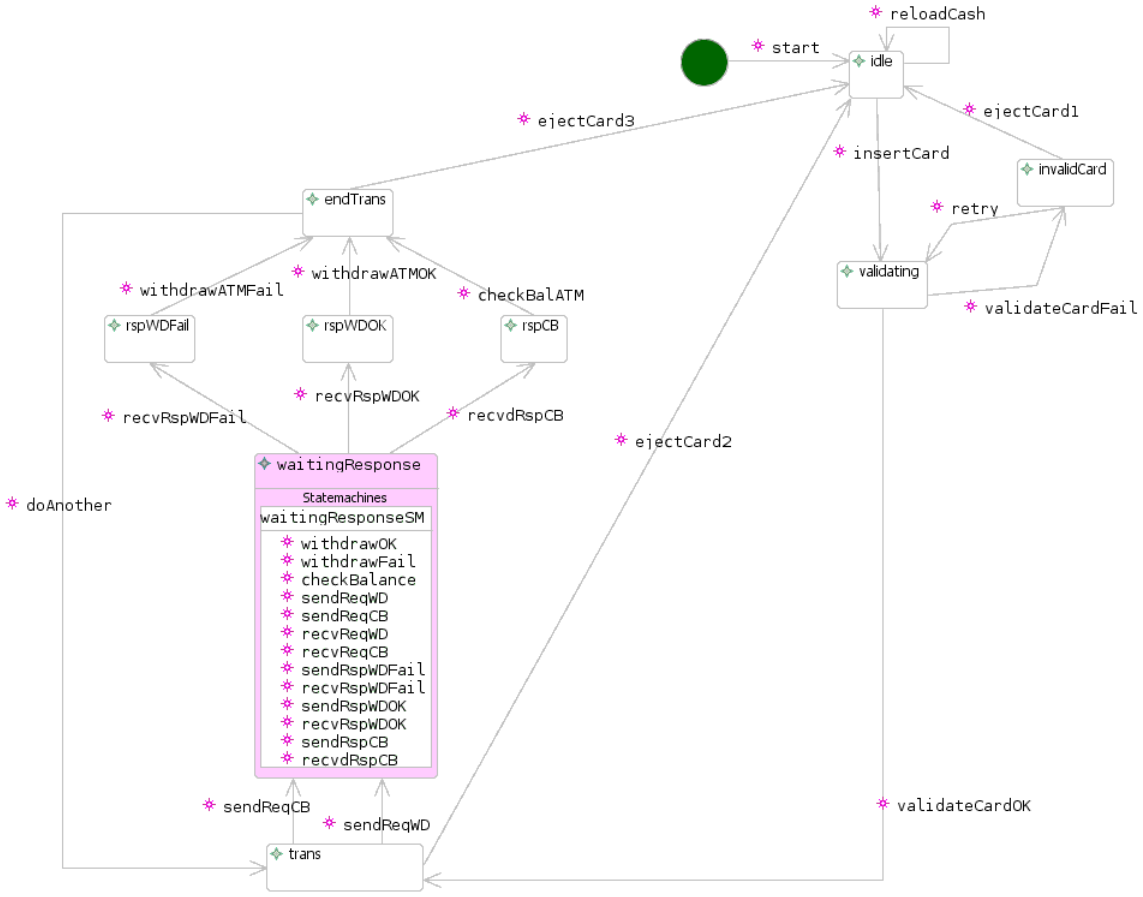


FIGURE 8.3: The Refined State Machine ATM_SM of the Sixth Refinement

nested state machine *waitingResponseSM*. In this refinement, the transition *sendReqCB* will change an ATM state from *trans* into the state *waitingResponse*. Similarly, the transition *sendReqWD* will change an ATM into the state *waitingResponse*. While in the state *waitingResponse*, either the transition *recvdRspWDOK*, *recvdRspWDFail* or *recvdRspCB* may occur. These three transitions will place an ATM into the state *rspWDOK*, *rspWDFail* or *rspCB* respectively.

The states and transitions of the nested state machine *waitingResponseSM* can be seen in Figure 8.4. The transitions from initial states and end at final states elaborate the incoming and outgoing transitions of the super-state *waitingResponse*. All other transitions refine the respective abstract transitions of *ATM_R5*.

8.5 Seventh Refinement

The seventh refinement introduces instances of ATM as a middleware. We intend to decompose the machine into three machine components i.e., ATM, bank and middleware.

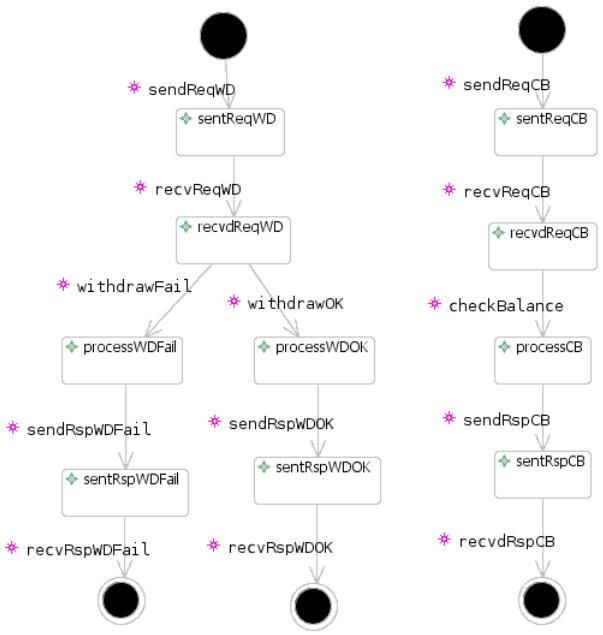


FIGURE 8.4: The State Machine *waitingResponseSM* of the Sixth Refinement

The middleware will synchronize the communication between ATM and bank by sharing the transitions send and receive for both request and response. These components cannot share variables. Therefore attributes which duplicate the attributes sent by ATMs to the bank and the attribute sent by the bank to ATMs are introduced in the middleware.

Figure 8.5 shows the classes of the seventh refinement. The classes are the refined classes *account*, *atm*, *atmB* and a new class *atmM*. The refined class *atm* represents a set of ATMs as the requesting part. The refined class *atmB* represents the ATMs set as the bank, which is the responding part. The refined class *atmB* has a new attribute *atm_acbalB* and the attribute *atm_acbal* is removed from the refined class *atm*. The attribute *atm_acbalB* represents an account balance at the bank. Sending a request involves copying request attributes (e.g., withdrawal amount) from ATM to middleware. The class *atmM* is used to model this. Similarly sending a response involves copying response attributes (e.g., account balance) from bank to middleware. Again, *atmM* is used to model this. The class *atmM* has three attributes *atm_cardM*, *atm_wdamM* and *atm_acbalM* which represent copies of the set of ATM cards, withdrawal amounts and accounts balance of ATMs at the middleware respectively.

In this refinement the attribute *atm_acbal* is replaced by *atm_acbalB* and moved to the class *atmB*. This is because the *atm_acbal* represents the account balance which is updated at the bank by the event *withdrawOK*, *withdrawFail* or *checkBalance*. The movement of the account balance attribute to *atmB* is postponed until the middleware is introduced. This is because when sending a response to an ATM, the account balance is removed from the set *atm_acbal* at the bank. However, a copy of the updated account

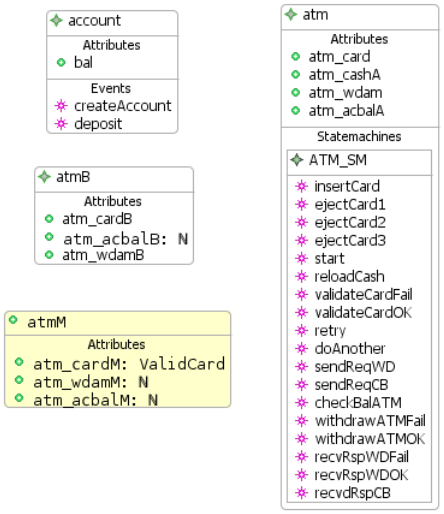


FIGURE 8.5: Classes of the Seventh Refinement

balance by bank is needed at an ATM to display to the user. Therefore a copy of the account balance needs to be made at the middleware so that it can be send to the ATM.

In this refinement, only the refined class *atm* has state machines (including the nested state machine). This means all the transitions belong to the refined class *atm* until the decomposition is done. Decomposition will be explained later. The explicit parameter, guards and actions of the transition *sendReqWD* are shown in the following table. These are added as the properties of the transition which give rise to the properties of the Event-B event. The implicit parameter *selfATM*, guard and actions relating to the source and target states of the transition are generated automatically. The transition *sendReqWD* will add an instance of ATM to the class *atmM* (*act2*), copy the withdrawal amount at the ATM to the attribute *atm_wdamM* (*act3*) and copy the identity of the card in the ATM to the attribute *atm_cardM* (*act4*).

Transition: sentReqWD
Parameters: am (type: $\mathbb{N}$)
Guards: grd1: $selfATM \in dom(atm\_card)$ grd2: $atm\_cashA(selfATM) > MIN\_CASH$ grd3: $am \leq MIN\_CASH$
Actions: act1: $atm\_wdam(selfATM) := am$ act2: $atmM := atmM \cup \{ selfATM \}$ act3: $atm\_wdamM(selfATM) := am$ act4: $atm\_cardM(selfATM) := atm\_card(selfATM)$

Similarly the transition *sendReqCB* behaves like the transition *sendReqWD* except it will not copy a withdrawal amount.

For the transition *recvReqWD*, the attributes *atm_cardM* and *atm_wdamM* replace the attributes *atm_card* and *atm_wdam* in the guards (*grd1* and *grd2*) and actions (*act1* and *act4*). The transition *recvReqWD* will remove an ATM instance from the class *atmM* (*act3*) and from the domain of all attributes of *atmM* (*act5*, *act6* and *act7*).

In this level, *act2*, *act3* and *act5..act7* are explicitly added as the actions. Later in the development when applying decomposition, the transition *recvReqWD* is shared by the bank and middleware components. In the bank component described in Section 8.6.2, the transition *recvReqWD* is modelled as a constructor. In this case, *act2* is generated automatically. In the middleware component described in Section 8.6.3, the transition *recvReqWD* is modelled as a destructor. In this case, *act3* and *act5..act7* are generated automatically. Similarly for the above transition *sentReqWD*, the send response and receive response transitions are added which will be described later in this section. All these transitions have some actions which are constructors and destructors which will be generated automatically in the decomposition.

Transition: recvReqWD
Guards: grd1: $selfATM \in dom(atm\_cardM)$ grd2: $selfATM \in dom(atm\_wdamM)$
Actions: act1: $atm\_wdamB(selfATM) := atm\_wdamM(selfATM)$ act2: $atmB := atmB \cup \{ selfATM\}$ act3: $atmM := atmM \setminus \{ selfATM\}$ act4: $atm\_cardB(selfATM) := atm\_cardM(selfATM)$ act5: $atm\_acbalM := \{ selfATM\} \lhd atm\_acbalM$ act6: $atm\_cardM := \{ selfATM\} \lhd atm\_cardM$ act7: $atm\_wdamM := \{ selfATM\} \lhd atm\_wdamM$

The specification of the transition *recvReqCB* is similar to the transition *recvReqWD* except that *grd2* and *act1* are irrelevant. This refinement relies on the following invariants.

The invariant

$\forall\, a \cdot\, a \in (sentReqWD \cup sentReqCB) \Rightarrow a \in dom(atm\_card)$

specifies that each instance of ATMs which is either in the states *sentReqWD* or *sentReqCB*, has an ATM card associated with it.

The invariant

$\forall\, a \cdot\ a \in sentReqWD \Rightarrow a \in dom(atm\_wdam)$

specifies that each instance of ATMs which is in the state *sentReqWD*, has a withdrawal amount associated with it.

The invariant

$\forall\, a \cdot\ a \in (sentReqWD \cup sentReqCB) \wedge a \in dom(atm\_card) \wedge a \in dom(atm\_cardM) \Rightarrow atm\_card(a) = atm\_cardM(a)$

specifies that for all ATMs which are in the states *sentReqWD* or *sentReqCB*, the ATM card associated with the ATMs at the ATM is the same as the ATM card associated with the ATMs at middleware.

The invariant

$\forall\, a \cdot\ a \in sentReqWD \wedge a \in dom(atm\_wdam) \wedge a \in dom(atm\_wdamM) \Rightarrow atm\_wdam(a) = atm\_wdamM(a)$

specifies that for all ATMs which are in the states *sentReqWD*, the withdrawal amount at the ATM is the same withdrawal amount at middleware.

In this refinement, for the transitions *sendRspWDOK*, *sendRspWDFail* and *sendRspCB*, the attribute *atm_acbalB* replaces the attribute *atm_acbal* (*grd1*). The transition will remove the ATM from the variable *atm_acbalB* (*act2*). It also will add the ATM instance into the class *atmM* (*act6*), copy the value of account balance at a bank to a variable at a middleware (*act4*) and copy the ATM card at middleware (*act7*).

Transition: sendRspWDOK, sendRspWDFail or sendRspCB
Guards: grd1: $selfATM \in dom(atm\_acbalB)$ grd2: $selfATM \in dom(atmB)$
Actions: act1: $atmB := atmB \setminus \{selfATM\}$ act2: $atm\_acbalB := \{ selfATM\} \lhd atm\_acbalB$ act3: $atm\_cardB := \{ selfATM\} \lhd atm\_cardB$ act4: $atm\_acbalM(selfATM) := atm\_acbalB(selfATM)$ act5: $atm\_wdamB := \{ selfATM\} \lhd atm\_wdamB$ act6: $atmM := atmM \cup \{ selfATM\}$ act7: $atm\_cardM(selfATM) := atm\_cardB(selfATM)$

For the transitions $recvRspWDOK$, $recvRspWDOK$ and $recvRspCB$, the attribute atm_acbalM replaces the attribute *atm_acbal* of the abstract transitions ($grd1$ and $act1$).

It will remove an ATM from the class $atmM$ (*act2*) and from the domain of all attributes of the class $atmM$ (*act3*, *act4* and *act5*)

Transition: recvRspWDOK, recvRspWDOK or recvRspCB
Guards: grd1: $selfATM \in dom(atm\_acbalM)$ grd3: $selfATM \in dom(atmM)$ grd4: $selfATM \in dom(card)$
Actions: act1: $atm\_acbalA(selfATM) := atm\_acbalM(selfATM)$ act2: $atmM := atmM \setminus \{ selfATM\}$ act3: $atm\_cardM := \{ selfATM\} \lhd atm\_cardM$ act4: $atm\_acbalM := \{ selfATM\} \lhd atm\_acbalM$ act5: $atm\_wdamM := \{ selfATM\} \lhd atm\_wdamM$

This refinement relies on the following invariants:

The invariant

$\forall a \cdot a \in (rspWDOK \cup rspWDFail \cup rspCB)\ a \in dom(atm\_acbal) \wedge a \in dom(atm\_acbalA) \Rightarrow atm\_acbalA(a) = atm\_acbal(a)$

specifies that for all ATMs which are in the states *rspWDOK*, *rspWDFail* or *rspCB*, the account balance associated with the ATM of the variable *atm_acbalA* is the same as the account balance associated with the ATM of the variable *atm_acbal*.

The invariant

$\forall a \cdot a \in (processWDOK \cup processWDFail \cup processCB) \Rightarrow a \in dom(atm\_acbal)$

specifies that each instance of ATMs which is either in the state *processWDOK*, *processWDFail* or *processCB*, has an account balance associated with it.

The invariants

$\forall a \cdot a \in processWDOK \wedge a \in dom(atm\_acbal) \wedge a \in dom(atm\_acbalB) \Rightarrow atm\_acbal(a) = atm\_acbalB(a)$

$\forall a \cdot a \in processWDFail \wedge a \in dom(atm\_acbal) \wedge a \in dom(atm\_acbalB) \Rightarrow atm\_acbal(a) = atm\_acbalB(a)$

$\forall a \cdot a \in processCB \wedge a \in dom(atm\_acbal) \wedge a \in dom(atm\_acbalB) \Rightarrow atm\_acbal(a) = atm\_acbalB(a)$

specifies that for all ATMs which are in the states *processWDOK*, *processWDFail* or *processCB*, the account balance associated with the ATM of the variable *atm_acbal* is the same as the account balance associated with the ATM of the variable *atm_acbalB*.

The transitions *recvRspWDOK*, *recvRspWDFail*, *recvRspCB*, *withdrawATMOK*, *withdrawATMFail* and *checkBalATM* refines the corresponding transitions of *ATM_R6*. The guards and actions of these transitions involving the attributes *atm_acbal* are replaced with the attributes *atm_acbalM*. The refinement relies on the following invariants:

The invariant

$\forall a \cdot a \in (sentRspWDFail \cup rspWDFail \cup sentRspWDOK \cup rspWDOK \cup sentRspCB \cup rspCB) \Rightarrow a \in dom(atm\_acbal)$

sepcifies that each instance of ATMs which is either in the states *sentRspWDOK*, *rspWDOK*,*sentRspWDFail*, *rspWDFail*, *sentRspCB* or *rspCB*, has an account balance associated with it.

The invariants

$\forall a \cdot a \in (sentRspWDFail \cup rspWDFail) \wedge a \in dom(atm\_acbal) \wedge a \in dom(atm\_acbalM) \Rightarrow atm\_acbal(a) = atm\_acbalM(a)$

$\forall a \cdot a \in (sentRspWDOK \cup rspWDOK) \wedge a \in dom(atm\_acbal) \wedge a \in dom(atm\_acbalM) \Rightarrow atm\_acbal(a) = atm\_acbalM(a)$

$\forall a \cdot a \in (sentRspCB \cup rspCB) \wedge a \in dom(atm\_acbal) \wedge a \in dom(atm\_acbalM) \Rightarrow atm\_acbal(a) = atm\_acbalM(a)$

specifies that for all ATMs which are in the states *sentRspWDOK*, *rspWDOK*, *sentRspWDFail*, *rspWDFail*, *sentRspCB* or *rspCB*, the account balance associated with the ATM of the variable *atm_acbal* is the same as the account balance associated with the ATM of the variable *atm_acbalM*.

In practice, the above invariants are specified as machine invariants in the UML-B class diagram. The invariants are discovered by using the Rodin interactive prover. Examples of using the prover to construct invariants are given in Section 8.10.

The model of this refinement level can be improved by attaching a state machine to each class *atm*, *atmM* and *atmB* rather than attached the state machines only to the class *atm*. The state machine of *atm* will be the state machine as in Figure 8.3 but without the nested state machine *waitingResponseSM* of the state *waitingResponse*. Figure 8.6 shows the partitioning of the nested state machine *waitingResponseSM* into six state machines which are highlighted by six boxes. The box labelled *atmB* will be the state machine of the class *atmB*. The other five boxes labelled *atmM* will be the state machines of the class *atmM*. This separation of state machines is done the same way when performing machine decomposition in Section 8.6.

The current model does not separate the state machines this way because previously the translator (U2B) did not support the separation of state machines. However, the

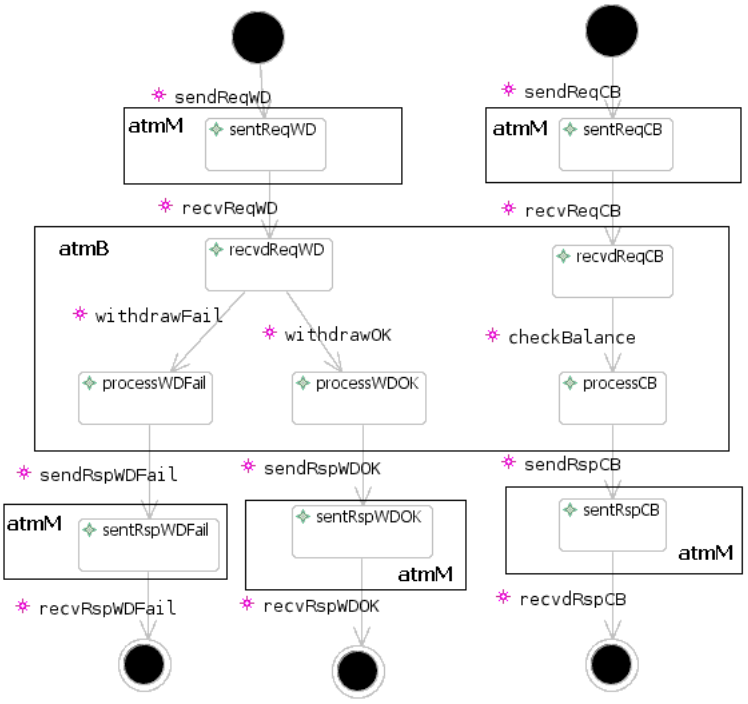


FIGURE 8.6: Partitioning of the Nested State Machine

translator has been updated to support modelling this separation of state machines into several classes. The translator will merge transitions with the same label in different state machines as one event in the generated Event-B machine. For example, the transitions *sentReqWD* of the state machine of *atm* and *atmM* classes will be translated as one event *sentReqWD* in Event-B machine. The event is a combination of both transitions. Separating the state machines into respective classes gives an advantage to the modeller by automatically generating the actions of adding instances (eg. *act2* of the *sentReqWD* and *act2* of the *recvReqWD*) and removing instances from classes and attributes (eg. *act3*, *act5*, *act6*, *act7* of the *recvReqWD*).

8.6 Decomposition of the Seventh Refinement and Composition (Eight Refinement)

The seventh refinement is decomposed into three machines: *mATM*, *mBank* and *mMW*. Machine *mATM* represents a model of ATMs which sends requests to the bank and receives responses from the bank. Machine *mBank* represents a model of a bank which receives requests from ATMs and sends responses to the ATMs. These two machines do not communicate with each other directly but they communicate indirectly via a middleware. This middleware is modelled in the machine *mMW*. This form of communication based on message passing is described in [25].

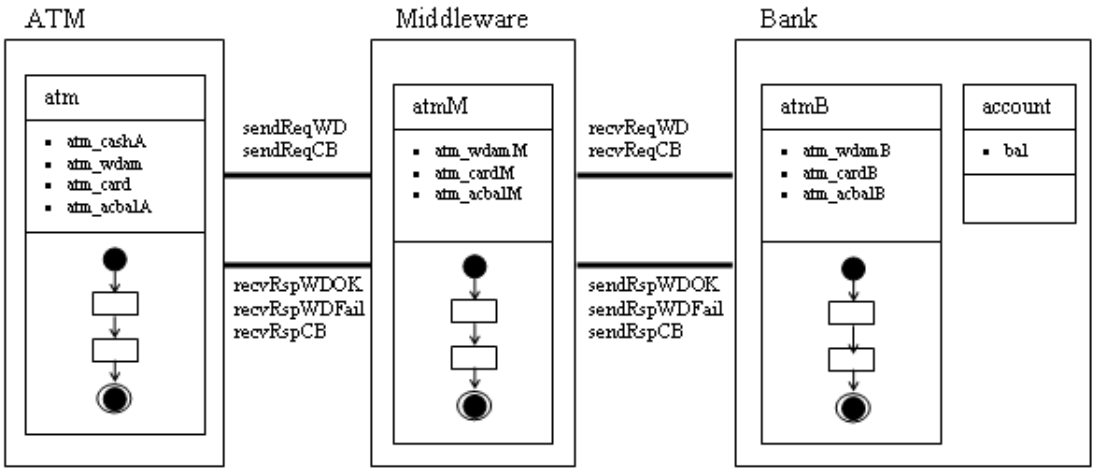


FIGURE 8.7: Architectural Illustration of Decomposition

Figure 8.7 illustrates the decomposition into the three machines. The classes of the seventh refinement are split between the machines *mATM*, *mBank* and *mMW* based on the transitions that update or use them. The class *atm* and its attributes belong to *mATM*, the classes *account*, *atmB* and their attributes belong to *mBank* and the class *atmM* and its attributes belong to *mMW*. The lines that join the machines represent the shared transitions between them. The lines are drawn to separate the send and receive transitions to make them easy to read and do not represent anything else.

The states of the state machines in the seventh refinement are partitioned into the three machines *mATM*, *mBank* and *mMW*. The structure of state machines of the seventh refinement is the same as in the sixth refinement therefore, the state machines of the sixth refinement in Figure 8.3 and 8.6 apply. The states of the state machine *ATM_SM* (Figure 8.3) occur at ATMs, therefore, they are the states of machine *mATM*. As explained in Section 8.5, in the nested state machine *waitingResponseSM* (Figure 8.6), the states within the boxes labelled *atmM* occur at the middleware, therefore, they are the states of machine *mMW*. The states within the box labelled *atmB* occur at the bank, therefore, they are the states of machine *mBank*. The incoming and outgoing transition of the partitioned states are the transitions of the respective machine.

The requesting component, *mATM* interacts with the middleware, *mMW* by sharing the send request transitions (transitions *sendReqWD* and *sendReqCB*) and receive response transitions (transitions *recvRspWDOK*, *recvRspWDFail* and *recvRspCB*). The responding component, *mBank* interacts with the middleware, *mMW* by sharing the receive request transitions (transitions *recvReqWD* and *recvReqCB*) and send response transitions (transitions *sendRspWDOK*, *sendRspWDFail* and *sendRspCB*).

The three decomposition machines are described in the following three sub-sections. The shared transitions are described in Section 8.6.4. The composition is described in

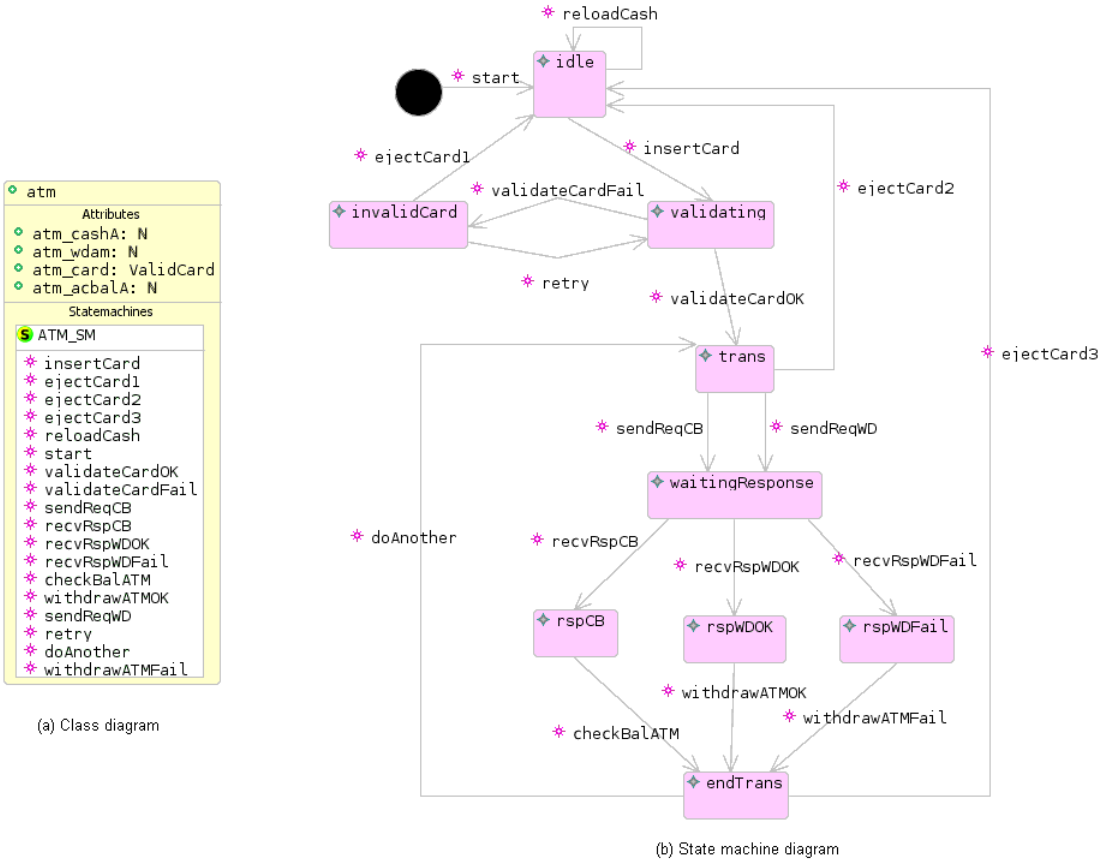


FIGURE 8.8: UML-B Machine for ATM Component

Section 8.6.5.

8.6.1 ATM Component: Machine *mATM*

Machine *mATM* is a model of the requesting component, the ATMs. Figure 8.8 shows the class diagram (a) and state machine diagram (b) of ATMs. The class diagram contains a class *atm* which represents a set of ATMs with attributes *atm_cashA*, *atm_acbalA*, *atm_wdam* and *atm_card*.

The state machine of *ATM_SM* is the state machine as in Figure 8.3 but without the nested state machine *waitingResponseSM* of the state *waitingResponse*. The state machine *ATM_SM* shows the states changes of an ATM when a transition is triggered. In particular, when an ATM is in a state *trans*, a withdrawal request (*sendReqWD*) or check balance request (*sendReqCB*) may be triggered and is send to a bank. A request will place an ATM in a waiting state (*waitingResponse)*, that is waiting for the request to be processed by bank. A receive response transition i.e., either a success withdrawal (*recvRspWDOK*), unsuccessful withdrawal (*recvRspWDFail)* or check balance (*recvRspCB*) may be triggered when an ATM is in a waiting state.

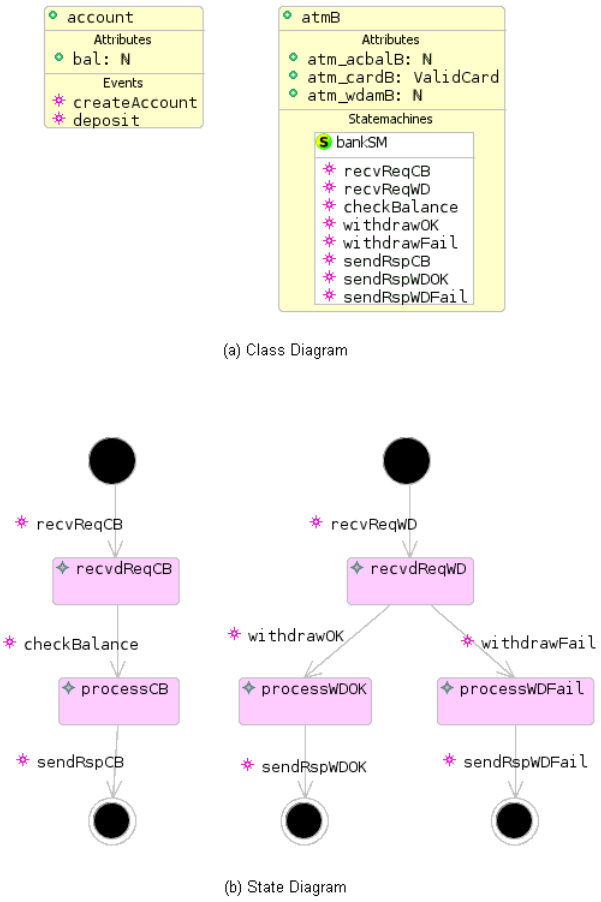


FIGURE 8.9: UML-B Machine for Bank Component

8.6.2 Bank component: Machine *mBank*

The bank component is modelled in machine *mBank* as shown in Figure 8.9. The class diagram (a) contains two classes, i.e., *account* and *atmB*. The class *account* has an attribute *bal*. The class *atmB* has attributes *atm_acbalB*, *atm_wdamB* and *atm_cardB*. Attached to the class *atmB* is a state machine *bankSM* which model the behaviour of a bank with respect to a requesting ATM.

Figure 8.9(b) shows the state machine of the bank component. It is the partitioned state machine in Figure 8.6 which is labelled as *atmB*. The state machine *bankSM* shows the states changes of an ATM when a transition is triggered. A bank may receive a withdrawal request (transition *recvReqWD*) or check balance request (transition *recvReqCB*) and will place the bank in the state *recvdReqWD* and *recvdReqCB* respectively. After the bank has process a request (transitions *withdrawOK*, *withdrawFail* or *checkBalance*), it will send the result to an ATM (transitions *sendRspWDOK*, *sendRspWDFail* and *sendRspCB*).

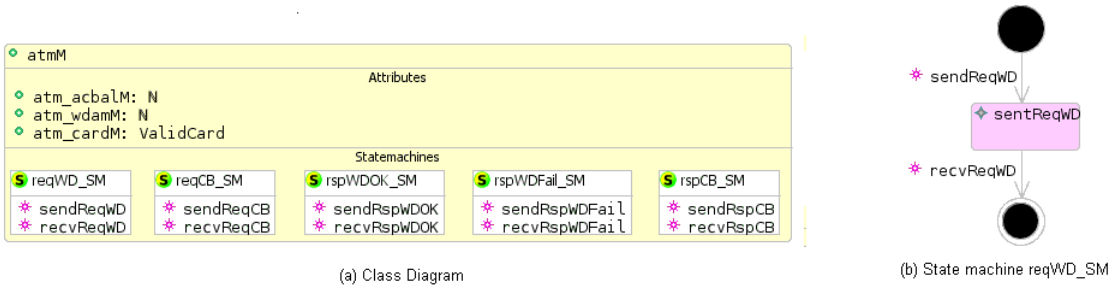


FIGURE 8.10: UML-B Machine for Middleware Component

8.6.3 Middleware component: Machine *mMW*

The middleware component is modelled in machine *mMW*. It synchronizes the send and receive transitions between ATM and bank components.

Figure 8.10(a) shows the class diagram for machine *mMW* with a class *atmM* and its attributes *atm_acbalM*, *atm_wdamM* and *atm_cardM*. Attached to the class *atmM* are the five state machines *reqWD_SM*, *reqCB_SM*, *rspWDOK_SM*, *rspWDFail_SM* and *rspCB_SM*. All these state machines have only one state (excluding initial and final states) and two transitions, i.e., send and receive transitions. These state machines are the partitioned state machines in Figure 8.6 which are labelled as *atmM*. Figure 8.10(b) shows the state machine *reqWD_SM*.

8.6.4 Shared Transitions

The send request transitions (*sendReqWD* and *sendReqCB*) and receive response transitions (*recvRspWDOK*, *recvRspWDFail* and *recvRspCB*) are shared by machines *mATM* and *mMW*. The receive request transitions (*recvReqWD* and *recvReqCB*) and send response transitions (*sendRspWDOK*, *sendRspWDFail* and *sendRspCB*) are shared by machines *mBank* and *mMW*. The guards and actions of these shared transitions are partitioned between the machines correspond to the splitting of the classes and attributes. For example, the *sendReqWD* transition of the machine *mATM* includes the guards and actions from the seventh refinement involving the class *atm* and its attributes *atm_card*, *atm_cashA* and *atm_wdam*. The rest of the guards and actions are in the *sendReqWD* transition of the machine *mMW*.

When splitting the transitions, a number of parameters may be introduced in the transitions in both machines *mATM* and *mMW* or in both machines *mMW* and *mBank*. For an example, consider the transition *sendReqWD* of the seventh refinement. We want to split the transition into the machines *mATM* and *mMW*. The parameter *am* is shared by both machines. The guards *grd1..grd3* are the guards of machine *mATM*. The action *act1* is the action of machine *mATM*. The actions *act2..act4* are the actions of machine

mMW. However, this is not possible since the action *act4* refers to an attribute *atm_card* of machine *mATM*. The action assigns the value of the attribute *atm_card* to the attribute *atm_cardM*. To overcome this, a new shared parameter c is introduced in the transition *sendReqWD* in both machines. Additionally, *grd4* is added in the machine *mATM* which specifies the parameter c as the value of the the attribute *atm_card*. Then, *act2* of *mMW* assigns c to the attribute *atm_cardM*.

Transition: sentReqWD of ATM_R7
Parameters: am (type: $\mathbb{N}$)
Guards: grd1: $selfATM \in dom(atm\_card)$ grd2: $atm\_cashA(selfATM) > MIN\_CASH$ grd3: $am \leq MIN\_CASH$
Actions: act1: $atm\_wdam(selfATM) := am$ act2: $atmM := atmM \cup \{ selfATM\}$ act3: $atm\_wdamM(selfATM) := am$ act4: $atm\_cardM(selfATM) := atm\_card(selfATM)$

Transition: sendReqWD of mATM
Parameters: par1: am (type: $\mathbb{N}$) par2: c (type: $ValidCard$)
Guards: grd1: $selfATM \in dom(atm\_card)$ grd2: $atm\_cashA(selfATM) > MIN\_CASH$ grd3: $am \leq MIN\_CASH$ grd4: $atm\_card(selfATM) = c$
Actions: act3: $atm\_wdam(selfATM) := am$

Transition: sendReqWD of mMW
Parameters: par1: am (type: $\mathbb{N}$) par2: c (type: $ValidCard$)
Actions: act1: $atm\_wdamM(selfATM) := am$ act2: $atm\_cardM(selfATM) := c$

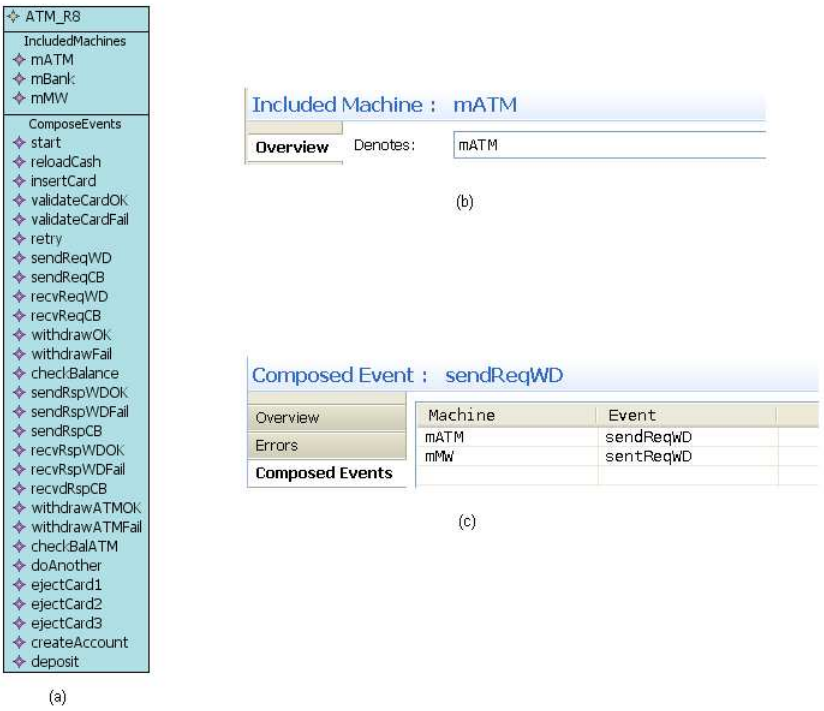


FIGURE 8.11: UML-B Composed Machine ATM_R8

8.6.5 Eighth Refinement: Composed Machine

The composed machine *ATM_R8* in Figure 8.1 composes the three component machines *mATM*, *mBank* and *mMW*. The composed machine *ATM_R8* refines the machine *ATM_R7* and it gives rise to the generated Event-B machine *ATM_R8*. The composed machine ensures that the composition of the three decomposed machines are valid refinement machine. Thus, they can be refined separately from one another which can reduce the complexity of modelling and proving a system being modelled.

Figure 8.11(a) shows the composed machine *ATM_R8*. In the *IncludedMachine* compartment is a list of include machines. Each of the included machine denotes the decomposed machine *mATM*, *mBank* and *mMW* correspondingly. Example of a denote property is in Figure 8.11(b) for the included machine *mATM*.

In the *ComposeEvent* compartment of *ATM_R8* is a list of composed events. These compose events are all the constituent events from the machines *mATM*, *mBank* and *mMW*. Each of the composed events refines its corresponding abstract event of the machine *ATM_R7*. For each composed event, it is necessary to specify which machine(s) it composed of together with the event of the machine(s). Figure 8.11(c) shows the property for the composed event *sendReqWD* that combines the *sendReq* events of machines *mATM* and *mMW*.

8.7 Refinement of The Middleware Component

The refinement of the middleware machine involves introducing message datatypes and message variables. The datatypes are modelled in the context *ATM_CXR3* which extends the context *ATM_CXR2*.

In the refinement, a class of type message and a number of different message types replace the class of the abstract machine and the class's attributes. The extension of the context and refinement machines is described in the following subsections.

8.7.1 Context *ATM_CXR3*

The sets and properties of the message types used in the middleware component are defined in a context *ATM_CXR3* which extends the context *ATM_CTXR2* (Figure 8.1). Figure 8.12 shows the context diagram of *ATM_CXR3* which contains two extended classtypes namely *ATM* and *ValidCard* and nine new classtypes which declare the types of messages that are used in the machines.

The classtype *MSG* is linked with the classtype *ATM* by the association *msg_atm*. This association identifies which ATM sends or receives a message. It also associate with the classtype *ValidCard* by the association *msg_card* which links a message with a card. These associations are also shown as the attributes of the classtype *MSG*. The classtype *MSG* has two disjoint subtypes which are *REQ_MSG* and *RSP_MSG*. These subtype relationships are translated into Event-B as subsets as outlined in Chapter 2. For example, the set *REQ_MSG* is a subset of the set *MSG*.

The classtype *REQ_MSG* represents a request message type and has two subtypes. The subtype *REQ_WD_MSG* specializes a request message as a withdrawal request message type and the subtype *REQ_CB_MSG* specializes a request message as a check balance request message type. The classtype *REQ_WD_MSG* has an attribute *msg_wdAmount* of type $\mathbb{N}$ that links a withdrawal request message with a withdrawal amount.

The classtype *RSP_MSG* represents a response message type and has an association *msg_status* with the classtype *STATUS* which has two elements *OK* and *NOT_OK*. This association links a response message with a status of a processed request. The classtype *RSP_MSG* also has an attribute *msg_bal* of type $\mathbb{N}$ which links a response message with an account balance. The classtype *RSP_MSG* has three subtypes. The subtype *RSP_WDOK_MSG* is a specialization for a successful withdrawal response message type. The subtype *RSP_WDFAIL_MSG* specializes a response message as an unsuccessful withdrawal response message type and *RSP_CB_MSG* specializes a response message as a check balance response message type.

Three partition axioms are constructed which specify the supertype is a union of its

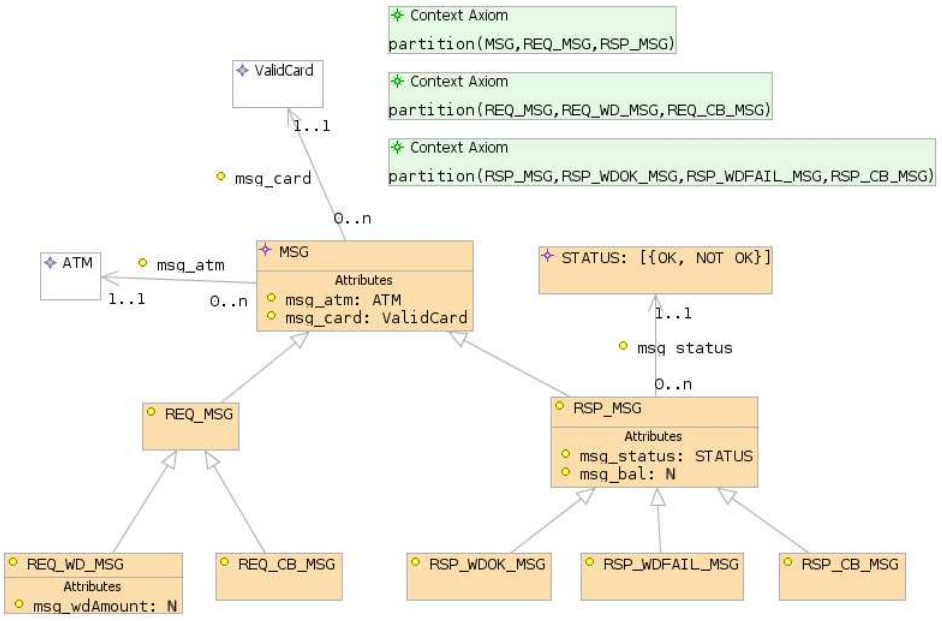


FIGURE 8.12: UML-B Context with Message Types and Properties

subtypes and the subtypes messages are disjoint. These axioms are shown in the Figure 8.12. All the three component machines *mATM*, *mBank* and *mMW* sees the context *ATM_CXR3* which means these machines can use the message types and its properties.

8.7.2 First Refinement of Middleware: Machine *mMW_R1*

The first refinement of the middleware replaces the states of the middleware with respect to ATMs with the states of type messages. These message states implicitly represent the requesting ATMs. Figure 8.13(a) shows the class *msg* for machine *mMW_R1*. The class *msg* is of type *MSG* which represents instances of messages at the middleware. These messages are partitioned into five kinds of messages which are modelled in the state machines *reqwd_SM*, *reqcb_SM*, *rspwdok_SM*, *rspwdfail_SM* and *rspcb_SM*. Similar to the abstract middleware *mMW*, all the state machines are simple with only one state and two transitions representing the send and receive events.

An example of state machine is shown in Figure 8.13(b). The example is the state machine *reqwd_SM* which consists of a state *reqwdmsg* of type *REQ_WD_MSG*. It represents a set of withdrawal request messages. The state machine *reqcb_SM* consists of a state *reqcbmsg* of type *REQ_CB_MSG* which represents a set of check balance request messages. The state machine *rspwdok_SM* consists of the state *rspwdokmsg* of type *RSP_WDOK_MSG* which represents a set of successful withdrawal response messages. The state machine *rspwdfail_SM* consists of the state *rspwdfailmsg* of type *RSP_WDFAIL_MSG* which represents a set of unsuccessful withdrawal response messages. The state machine *rspcb_SM* consists of the state *rspcbmsg* of type *RSP_CB_MSG*

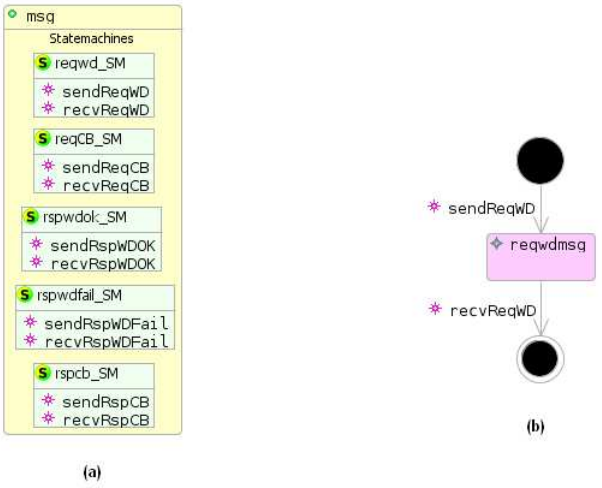


FIGURE 8.13: UML-B Machine for the First Refinement of the Middleware

which represents a set of successful check balance response messages.

The class *msg* replaces the class *atmM* of the abstract machine *mMW*. The states *reqwdmsg*, *reqcbmsg*, *rspwdokmsg*, *rspwdfailmsg* and *rspcbmsg* respectively replace the states *sentReqWD*, *sentReqCB*, *sentRspWDOK*, *sentRspWDFail* and *sentRspWDOK* of *mMW*.

The send transitions such as the transition *sendReqWD* of the state machine *reqwd_SM* is a constructor that selects an unused instance of message and adds it to the class *msg* and the state *reqwdmsg*. The transition can be triggered when a new message *selfMsg* is of type *MSG* and is not a member of the set *msg*, it is specifically of type *REQ_WD_MSG* (*grd1*), the ATM is not in the set *msg* (*grd2*), the value of the ATM field of the message is *selfATM* (*grd3*), the value of the card field of the message is *c* (*grd4*) and the value of the withdrawal amount field of the message is *am* (*grd5*).

Transition: sendReqWD
Parameters: par1: $selfATM$ (type: ATM) par2: am (type: $\mathbb{N}$) par3: c (type: $ValidCard$)
Guards: grd1: $selfMsg \in REQ\_WD\_MSG$ grd2: $selfATM \notin msg\_atm[msg]$ grd3: $msg\_atm(selfMsg) = selfATM$ grd4: $msg\_card(selfMsg) = c$ grd5: $msg\_wdAmount(selfMsg) = am$

The transition *recvReqWD* is a destructor that will remove a message from the class

msg and the state *reqwdmsg*. Similar to the *sendReqWD*, the three guards (*grd1..grd3*) specify the respective message field to its value are added.

Transition: recvReqWD
Parameters: par1: $selfATM$ (type: ATM) par2: am (type: $\mathbb{N}$) par3: c (type: $ValidCard$)
Guards: grd1: $msg\_atm(selfMsg) = selfATM$ grd2: $msg\_card(selfMsg) = c$ grd3: $msg\_wdAmount(selfMsg) = am$

The transitions of the other four state machines behave similar to the state machine *reqwd_SM*. A check balance request message consists of the fields about an ATM and an ATM card. A response message consists information about an ATM, an ATM card, a status (*msg_status*) and an account balance (*msg_bal*).

The first refinement machine relies on a number of invariants. In practice, the invariants are specified as machine invariants in the UML-B class diagram. Five of the invariants specialize the type of the states in the state machines as follows:

$$reqwdmsg \in \mathbb{P}(REQ\_WD\_MSG)$$

defined the state $reqwdmsg$ as type withdrawal request message.

$$reqcbmsg \in \mathbb{P}(REQ\_CB\_MSG)$$

defined the state $reqcbmsg$ as type check balance request message.

$$rspwdokmsg \in \mathbb{P}(RSP\_WDOK\_MSG)$$

defined the state $rspwdokmsg$ as type successful withdrawal response message.

$$rspwdfailmsg \in \mathbb{P}(RSP\_WDFAIl\_MSG)$$

defined the state $rspwdfailmsg$ as type unsuccessful withdrawal response message.

$$rspcbmsg \in \mathbb{P}(RSP\_CB\_MSG)$$

defined the state $rspcbmsg$ as type check balance response message.

The other invariants are described in Section 8.10.

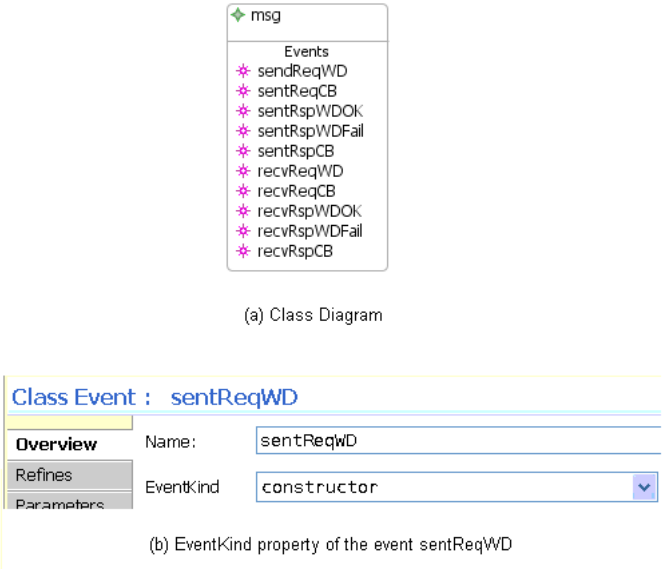


FIGURE 8.14: Class Diagram of the Second Refinement for the Middleware Component

8.7.3 Second Refinement of Middleware: Machine *mMW_R2*

The machine *mMW_R2* is the second refinement for the middleware component. It models a single message channel which merges the five channels of messages. This is done to reflect an implementation: Instead of having five separate channels, we have a single channel that can carry five different kinds of message.

Figure 8.14(a) shows a class diagram containing the refined class *msg*. In this refinement, all the five kinds of messages are removed. That means, the five refined state machines are dropped. All the transitions of machine *mMW_R1* are moved as the class events of the refined class *msg*. All the send transitions such as *sendReqWD* are specified as a constructor in the event kind property (Figure 8.14(b)). All the receive transitions such as *recvReqWD* is specified as a destructor event kind.

There is no additional guard added to the send events. For each receive event, a guard which specifies a specific type of the message corresponds to the type of message that the send event is sending is added. For example, for *recvReqWD*, the guard $selfMsg \in REQ\_WD\_MSG$ is added.

In this refinement, no gluing invariant is required. This is because, the class *msg* which represents the single channel has been introduced in the abstract machine together with the five specific channels (represented by the five states in the state machines of *msg* class).

Machines	**POs**	**aPOs**	**iPOs**
ATM_R5	72	47	25
ATM_R6	125	109	16
ATM_R7	229	222	7
ATM_R8	63	63	0
mATM	108	108	0
mBank	89	89	0
mMW	108	108	0
mMW_R1	194	188	6
mMW_R2	24	24	0
Total	1012	958	54

TABLE 8.1: Statistics from the Proof Effort

8.8 Statistics of Proof Obligations

All the proof obligations (POs) for the machines were generated and proved using the Rodin tool provers [4]. The statistics are outlined in Table 8.1 showing the total POs for each machine (POs), the number of POs which are automatically discharged (aPOs) and the number of POs which are interactively discharged (iPOs).

In *ATM_R5*, 25 POs are proved manually or interactively with the Rodin prover by rewriting the partition invariant in the hypothesis and the goal into its definition. Then the POs are discharged by calling *ML* prover. Similarly the sixteen POs in *ATM_R6* are discharged manually by rewriting the partition invariant in the hypothesis and goal. In *ATM_R7*, there are seven interactively discharged POs. Four POs are discharged manually by splitting cases, two POs are discharged by first split cases, followed by proving that two states are disjoint and one PO is discharged by proving that two states are disjoint. In *mMW_R1*, there are six interactively discharged POs. These POs are discharged manually by splitting cases.

8.9 Guidelines for Refinement and Decomposition

This section outlines some guidelines for refinement and decomposition in a UML-B development. These guidelines are based on experience in modelling the ATM case study in UML-B.

- Create classtypes in a context diagram instead of the generating sets in an implicit context from an early stage in the development because one or more of the classtypes may be extended later in the refinement.

  For the ATM case study, initially all the sets *ATM*, *Card* and *Pin* are generated from the classes in the class diagram. These sets are contained in the machine

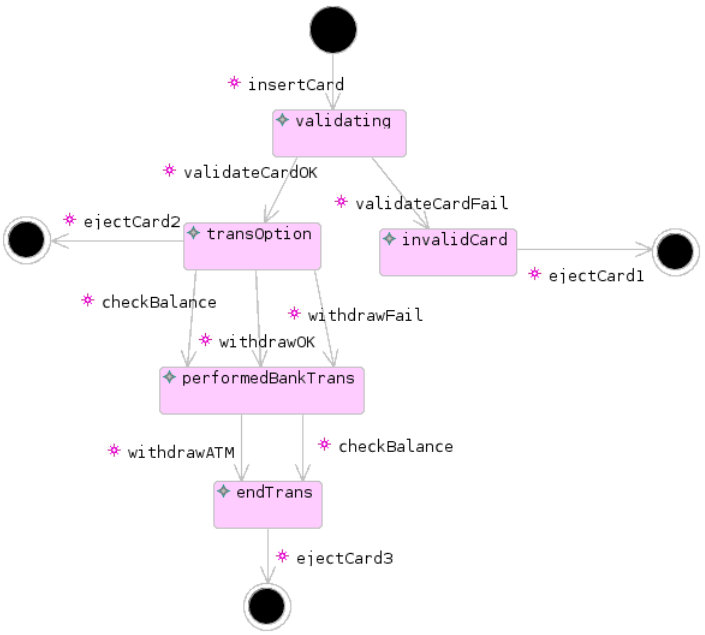


FIGURE 8.15: Example of a Bad State Machine

implicit context. However, this implicit context could not be extended graphically. In the case study, we wanted to extend the context to introduce the message types. Therefore, the ATM model is modified so that the sets *ATM*, *Card* and *Pin* are generated from the classtypes in the contexts instead of classes in the state machines.

- Get the sequences of events in the same state machine hierarchy level.

  For example the effect of withdrawal request at an ATM is that the ATM dispatches the requested cash. In the initial UML-B development of the ATM case study, the *withdrawATM* transition is introduced in *ATM_R2*. Figure 8.15 shows the state machine of the initial development. However, this is not reasonable because the transition withdrawal request is not yet introduced in *ATM_R2*. The event withdrawal request (transition *requestWD*) is introduces in the machine *ATM_R3*, therefore, it is more reasonable to introduce the transition *withdrawATM* in *ATM_R3*. By doing this, the effect of request is symmetry with the response(s). Figure 8.16 is the state machine showing a better state machine. Both the transitions *requestWD* and *withdrawATM* are introduced in the same hierarchy level i.e., in the nested state machines (b and c). Figure 8.15 is considered as a bad state machine because the transitions *withdrawATM* and *checkBalATM* exist in it. They should not exist because the request transitions (i.e., the *requestWD* and *requestCB*) which causes the transitions *withdrawATM* and *checkBalATM* to exist, do not exist in the same state machine.

- When introducing the responding part (the component which respond to requests),

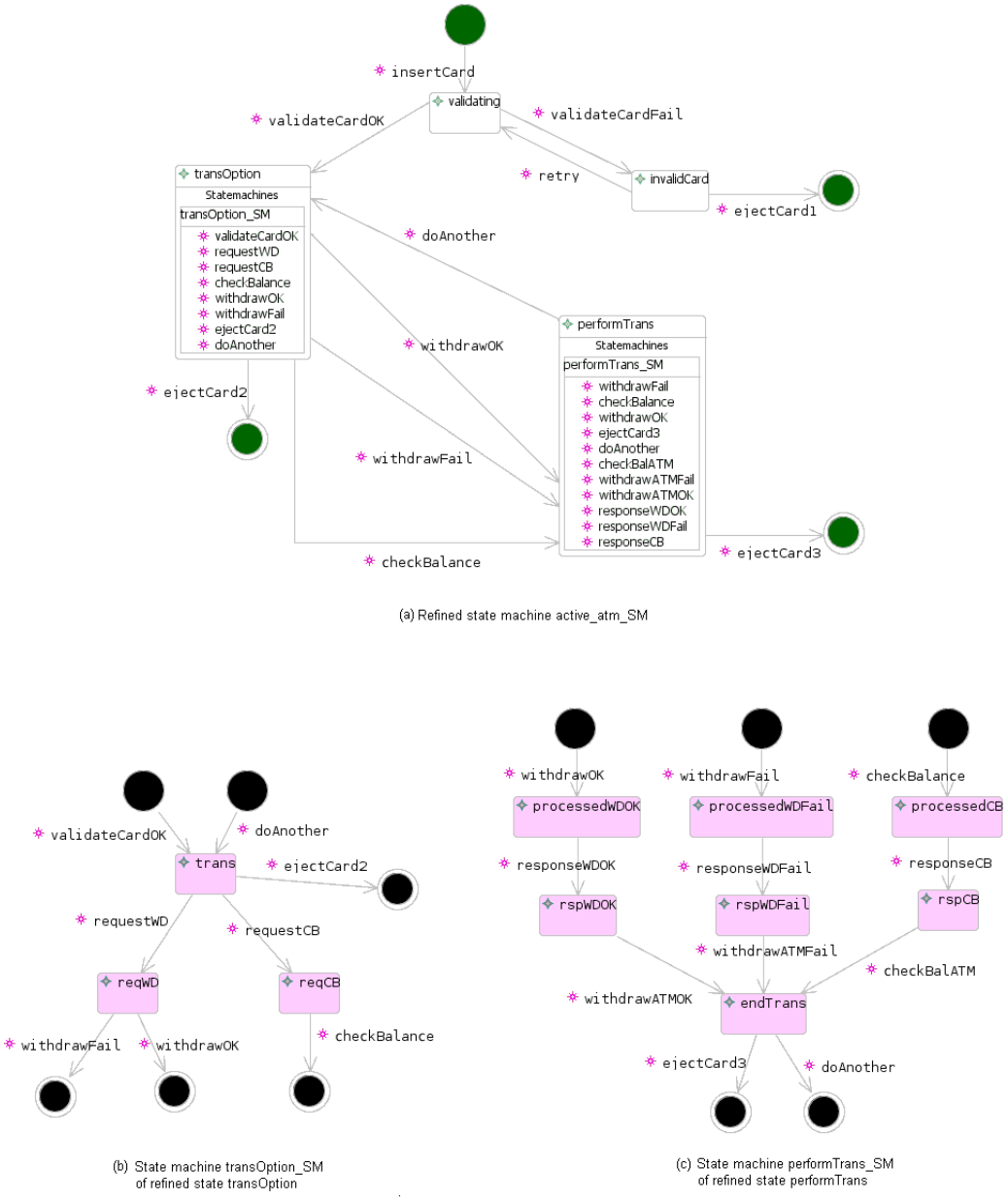


FIGURE 8.16: Example of a Good State Machine

duplicate the sending part attributes which are sent to the responding part.

The initial development of the ATM case study has introduced messages and decomposition at the same time. A copy of withdrawal request message is made when bank receiving a request ($reqwdmsgB := reqwdmsgB \cup \{m\}$). However, this approach results in a new invariant needing to be introduced in the composed machine which relate the message *reqwdmsgB* and the abstract attribute *atm_wdam*. And also further invariants need to be added to prove the invariant. Similarly for the abstract attribute *atm_card*. The additional invariants in the composed machine made it difficult to understand. To overcome this, the

attribute *atm_wdam* is duplicated by the attribute *atm_wdamB*. Similarly, the attribute *atm_card* is duplicated by the attribute *atm_cardB*. These duplications are done before decomposition.

- When introducing middleware, duplicate the sending part attributes which are sent to the responding part and duplicate the responding part attributes which are sent to the sending part.

  For the ATM case study, initially there is *atm_cardM*, *atm_wdamM* and *atm_acbalM* in the seventh refinement (*ATM_R7*) which represents the data send from a sending component to responding component (*atm_cardM*, *atm_wdamM*) and the data send from a responding component to requesting component (*atm_acbalM*). Initially, the message variables *reqwdmsg* and other messages are introduced when introducing the decomposed machines *mATM*, *mBank* and *mMW*. This resulted in many invariants being constructed when modelling the composed machine which composed the machines *mATM*, *mBank* and *mMW*. In other words, the composed machine is complicated to understand because we were modelling two things in one machine which are composition and introducing messages at the same time.

8.10 Patterns for Invariants when Introducing Messages in the Refinements of the Middleware

This section outlines some pattern of invariants when introducing message variables of the first refinement of middleware component. In future, these invariants could be generated automatically by the UML-B tool.

Pattern 1: Invariants that specify a replacement of the state variables of an abstract middleware with the new state variables representing a set of messages in the refinement.

For instance, the invariant

$$sentReqWD = msg\_atm[reqwdmsg]$$

where *sentReqWD* is a state variable in an abstract middleware machine and *reqwdmsg* is a state variable in a refinement that represents a set of withdrawal request messages. *msg_atm* is one of the fields in a message that links a message with an ATM.

The above invariant is identified by using the Rodin interactive prover when the proof obligation of the guard $selfATM \in sentReqWD$ in the event *recvReqWD* was not discharged. The hypotheses and the goal are as follows:

Hypotheses:

$m \in msg$

$m \in reqwdmsg$

The goal:

$msg\_atm(m) \in sentReqWD$

The invariant can also be written as $\forall m.m \in reqwdmsg \Rightarrow msg\_atm(m) \in sentReqWD$. We prefer $sentReqWD = msg\_atm[reqwdmsg]$ because it is proved automatically.

Similarly for check balance requests where the state variable *reqcbmsg* replaced the abstract state variable *sentReqCB*. For successful withdrawal responses, the state variable *rspwdokmsg* replaced the abstract state variable *sentRspWDOK*. For unsuccessful withdrawal responses, the state variable *rspwdfailmsg* replaced the abstract state variable *sentRspWDFail*. For check balance responses, the state variable *rspcbmsg* replaced the abstract state variable *sentRspCB*.

Pattern 2: An invariant that specifies a replacement of an abstract relation of a middleware with a set of messages in the refinement.

For examples,

The invariant

$$\forall\, m \cdot\ m \in msg \Rightarrow msg\_atm(m) \mapsto msg\_card(m) \in atm\_cardM$$

specifies for each m in msg, there is a mapping from the ATM field of the message to the card field in atm_cardM. The invariant replaces the following abstract relation

$$atm\_cardM \in atmM \rightarrow ValidCard$$

Another invariant with the same pattern is the following invariant

$$\forall\, m \cdot\ m \in reqwdmsg \Rightarrow msg\_atm(m) \mapsto msg\_wdAmount(m) \in atm\_wdamM$$

which specifies for each m in $reqwdmsg$, there is a mapping from the ATM field of the message to the withdrawal amount field in atm_wdamM. The invariant replaces the following abstract relation

$$atm\_wdamM \in atmM \nrightarrow \mathbb{N}$$

The following invariant also is the same pattern as the above two invariants.

$$\forall\, m \cdot\ m \in (rspwdokmsg \cup rspwdfailmsg \cup rspcbmsg) \Rightarrow msg\_atm(m) \mapsto msg\_bal(m) \in atm\_acbalM$$

which replaces the following abstract relation

$$atm\_acbalM \in atmM \nrightarrow \mathbb{N}$$

The above invariants are identified by using the Rodin interactive prover. For instance, the invariant $\forall\ m\cdot\ m\in msg \Rightarrow msg\_atm(m) \mapsto msg\_card(m) \in atm\_cardM$ was constructed when the proof obligation of the guard $atm\_cardM(selfATM) = c$ in the abstract event *recvReqWD* was not discharged. The hypotheses and the goal are as follows:

Hypotheses:

$m \in msg$

$m \in reqwdmsg$

The goal:

$atm\_cardM(msg\_atm(m))=msg\_card(m)$

From the hyphotheses, we chose msg because it is the superset so we do not need to have five similar invariants for all five subset of messages i.e., *reqwdmsg*, *reqcbmsg*, *rspwdokmsg*, *rspwdfailmsg* and *rspcbmsg*. The invariant can also be written as $\forall m.m\in msg \Rightarrow atm\_cardM(msg\_atm(m))=msg\_card(m)$. We prefer $msg\_atm(m) \mapsto msg\_card(m) \in atm\_cardM$ because it is easier to understand where it clearly defines that the mapping of the relations is an element of the set atm_cardM.

This pattern of invariant gives rise to the next pattern of invariant because these invariants are not proved with the receive transition.

Pattern 3: Invariants that prevent multiple messages corresponding to the same abstract element.

In the ATM case study, there cannot be multiple messages for the same ATM. The following invariant

$$\forall\ m,\ m0\cdot\ m\in msg\wedge\ m0\in msg\wedge\ msg\_atm(m)=msg\_atm(m0) \Rightarrow m=m0$$

specifies that each ATM associates with only one message in the set msg.

This invariant is constructed when the proof obligation of the invariant $\forall\ m\cdot\ m\in msg \Rightarrow msg\_atm(m) \mapsto msg\_card(m) \in atm\_cardM$ is not discharged against the event $recvReqWD$.

The undischarged invariant defines that the elements in the set msg are the elements of the abstract set atm_cardM. When a message m is removed from the set msg, it means the ATM (for instance $atm1$) and card associated with the message m is also removed

from the set atm_cardM. However, there is no constraint which restricts that there cannot be another message in msg which associated with $atm1$. Thus, the invariant is constructed.

8.11 Summary

This chapter presented the use of the flattening and grouping techniques which facilitate decomposition in the ATM case study. The case study has applied the shared-events decomposition between the three components: ATM, middleware and bank. Then, the notions of composed machine, included machine, composed event and constituent event of Chapter 7 are used in the case study to composed the decomposed machines. The development of the ATM case study has evaluated that the extensions of the metamodel described in Chapter 7 are working as expected.

Chapter 9

Conclusion

UML-B is a UML-like formal modelling language that integrates UML and B. This integration aims to make B more approachable while addressing the problem of ambiguity of UML. In this thesis, we have investigated a methodology to perform refinement and decomposition in UML-B. The motivation for this work comes from performing refinement and decomposition in Event-B. In this thesis, we have presented a brief background study on work related to this research. The later part of this report presented experience of modelling the ATM case study using the Rodin Event-B tool, followed by the notions of refined classes, refined state machines and extended classtypes. These notions are used to describe the techniques of refinement in UML-B. Then extensions of the UML-B metamodel and its implementation to support these notions are described. This chapter is followed by the ATM case study which validates the notions. This work also presented the notions which enable the composition of machines in UML-B. The techniques of flattening and grouping state machines which facilitate decomposition in UML-B are presented. The ATM case study which is used to demonstrate and validate the introduced notions and techniques is presented.

The following sections give a summary of the thesis contributions and limitations. Then we present a section on the comparison of our work to other work on translating UML diagrams to B. This section is followed by a section on the comparison to related work on the class and state machine refinements. A section on the comparison of modelling in UML-B to plain Event-B is also discussed. Finally, we present a comparison of UML-B to goal diagram and event refinement diagram.

9.1 Contributions

The main contributions on this work are extending the UML-B language and tool to support refinement and decomposition. In more detail, the contributions are as follows:

- Introducing the following techniques involving refinement of class and state machines:
  - Add new attributes and associations to a refined class.
  - Add new classes in a refinement.
  - State elaboration.
  - Transition elaboration.
  - Move events (or transitions) to a refined class or a new class in a refinement.
  - Add new attributes and associations to an extended classtype.
  - Add new classtypes in a refinement.
- Introducing the following techniques involving refinement of state machines which facilitate decomposition:
  - Flattening state machines.
  - State grouping.
- The development of the ATM case study which validates and demonstrates the extensions made to the UML-B and also the above techniques. The case study has established that the extensions to the UML-B metamodel, drawing tools and U2B translator are working as expected.
- Provides guidelines for performing refinement and decomposition in UML-B.
- Provides generic invariants in the refinement of a middleware component.

9.2 Limitations

Limitations of this work are as follows:

- The extensions to UML-B do not support parallel state machines.
- The decomposition work does not cater for the state function representation, in the techniques of flattening state machines and grouping states. It only deals with the disjoint sets representation.

9.3 Comparison to Other Work on Translating to B

There is much work on combining UML with formal notations and we now outline some of this. However, unlike our work, none of this work supports refinement or decomposition in UML to the best of our knowledge.

Lano, Clark and Androutsopoulos [54] present the translation of UML-RSDS into classical B. The work focused on translating class diagrams into B. Each class is translated into a respective B machine. Unlike UML-B, where all classes in a class diagram are translated into one Event-B machine. The constraint language used is OCL whereas we use μB.

Ledang and Souquiéres have introduced an approach for translating UML state machine diagrams into classical B in [61]. They have introduced the modelling of deferred events and the communication between UML state machines. Deferred events are events whose occurrence is responded to at a later time. These deferred events occur at inconvenient times where a state machine is in the state which cannot handle the events. Thus, these events are postponed until the state machine is in the state which can handle the events. The translations use the state function representation whereas UML-B supports both state function and state sets representations. In their work the UML notation is central which is different from UML-B where the Event-B notation is central.

Sekerinski [85] has worked on translating statecharts into classical B. He also used the state function representation. His work includes hierarchical state machines but did not combine hierarchical state machines and refinement as in our work. Another difference is that his work treated a statechart independently from a class diagram unlike our work where it is a sub-notation to a class diagram.

Mammar and Laleau [3] have presented an approach to developing database applications. The approach starts with specifying the data structures and transactions using class diagrams, state diagrams and collaboration diagrams. These diagrams are translated into B abstract models. This translation is supported by a tool. The B abstract models are then refined until the implementation phase. The B implementation models are then automatically generated into database schemas and java classes. Their work is suitable for a development of data-oriented applications in contrast to our work which is suitable for process-oriented applications. Another difference is that the refinements involve the generated abstract B models and there is no concept of refinement in UML whereas, in our work the refinements involve the class and state machine diagrams.

Compared to our work, all the above mentioned work integrates subsets of UML diagrams into classical B. Our work integrates UML-like diagrams with Event-B. Another difference is that in the other work the UML notation is central as B is only used as an analysis tool for property checking. On the contrary, in our work Event-B is central as the UML notation is used as a graphical medium which gives additional complementary structuring of Event-B models.

9.4 Comparison to Other Work on the Class and State Machine Refinements

In this section we outline some of the work related to refinement of UML diagrams.

The work on state machines refinement has been introduced by Snook and Walden in [93]. Their work is based on the old version of UML-B which was based on classical B and has been extended to include translation to an event "style" of B (which was a precursor to Event-B). They introduced state elaboration and transition elaboration techniques. The semantics of the state machine refinement are given by Event-B. However, we provide a more precise definition of refined state machine and we provide tool support based on UML-B giving a different model visualisation from the UML diagram symbol used in [93]. We also introduce class refinement techniques which are not dealt with in [93].

Plaska et al [67] have introduced a process for state machines refinement. The process involved the application of refinement patterns that are based on the techniques introduced in [93]. One of the applied refinement patterns is flattening the hierarchical states. This flattening is similar to our technique of flattening state machines. They applied the flattening technique in their work to make the state machines more readable than hierarchical state machines. We introduced the flattening technique to facilitate decomposition in UML-B. In addition, we provide tool support for the flattening technique based on UML-B.

Simons [91] has presented four informal refinement rules of state machines. The rules in the refinements are: (1) New states must be sub-states nested in the abstract states (super-states), (2) New transitions must only connect between the sub-states, (3) The incoming and outgoing transitions of the super-states must be preserved, and (4) The self transitions of the super-states must be preserved. Rules (1) and (2) must also be followed in UML-B state machine refinement. These two rules are achieved by applying the state elaboration technique. Rule (3) must also be followed in UML-B for a state machine refinement to be valid. In contrast to Rule (4), in our work, when refining self transitions, the occurrence of the transitions can either be many times or can be restricted to once. Restriction to once means removing looping behaviour and this is a valid refinement since we focus on preserving safety, not liveness, in our work. Unlike our work, Simon's work does not involve any formal notion.

The techniques of adding new attributes and associations to a class and adding new classes to a class diagram have been introduced in an informal way for refinement of UML class diagram [18] but no formal notation nor formal refinement concept is used. Templates are introduced for attributes and associations to specify the translation of model elements to low level design and implementation. Also, the technique of state elaboration has been introduced in a refinement of UML state diagram [1] again without a formal notion of refinement.

Schönborn and Kyas [51] have extended their formal language which defines the semantics of UML state machines. The language is using the mathematical notation of Davey and Priestley [32]. The language extension was done to ensure that state machine refinements are valid. They have defined a *redefinable* property when adding sub-states in a hierarchical state machine refinement. The sub-state is redefinable w.r.t. to an event e and a transition emitting it. In the abstract state machine, the event e triggers a transition emitting the super-state. Thus, in the refinement, there exists two transitions triggered by the same event e. This causes a conflict when event e occurs and the priority is that the transition with the deeper source state is selected. In contrast to UML-B, any transition emitting from a super-state is elaborated by a transition emitting from one of the sub-state. Thus, there are no conflicting transitions. They have suggested some refinement patterns for hierarchical state machines. One of the refinement patterns is that transitions emitting from a super-state may be removed or added as long as there exists a transition emitting from a sub-state triggered by the same event. This refinement is similar to a refinement in UML-B of removing an abstract transition and replacing it with a number of transitions each of which is elaborated by a transition emitting from a sub-state. This kind of refinement is explained in Section 4.4.

Knapp et al [9] have investigated the validity of UML state machine refinements by formalizing with MTLA [99]. In contrast to our work, their work does not consider state machine hierarchy in refinements. New transitions and states may be introduced in a refined state machine by replacing old states with new states and transitions. We prefer our approach because the relationship between abstract state machines and refinements is clearer. In UML-B, new transitions and states may be added in nested state machines. UML-B is more restrictive but this makes the refinement pattern simple and clear. Similar to our work, refining self transitions may be restricted to once as the work does not focus on liveness properties.

Pons [30] has investigated mapping the refinement of classes in Object-Z [92] into UML class diagrams. Pons suggests the object decomposition pattern. This pattern defines a refinement of a class by decomposing it to several classes. Pons suggests that this refinement pattern can be realized in UML class diagrams. The OCL [75] has been used to specify the pre and post conditions of operations. In UML-B, this kind of refinement has not been investigated. However, we believe that it works well in UML-B since UML-B allows an abstract class to be removed and replaced it with several classes which preserved the abstract data and behaviours. Another refinement pattern introduced is the non-atomic refinement pattern. This pattern refines a class by refining an abstract operation with several operations. This pattern is applied in UML-B however, we focus on refining a transition of a state machine by several transitions in a nested state machine. Unlike our work, this work does not aim to formalize UML diagrams but to discover possible refinement patterns which are not considered as refinement in UML based on the refinements in Object-Z. Our work has formalized and customized UML

class and state machine diagrams based on Event-B.

9.5 Comparison of UML-B to Plain Event-B

This section compares UML-B and Event-B. UML-B provides a front-end graphical tool to Event-B and supports object-orientation. In particular, UML-B supports class diagrams and state machines. As in [95], the motivation for introducing UML-B is because industrial users find the UML-like language and tool attractive. Secondly, UML-B provides additional complementary structuring of Event-B models in the form of classes and state machines.

Modelling in UML-B does not mean hiding the Event-B notation. The modeller needs to know Event-B in order to model in UML-B. Modelling in UML-B gives the advantage of generating many lines of Event-B specification from the diagram editors. In particular, the generation of the type invariants, the partition invariants, axioms, an instance parameter, guards and actions of events. We believe that this helps modellers to save time from having to specify them manually as in plain Event-B.

UML-B may attract people to Event-B because of it's resemblance to UML. UML-B is easy to learn compared to plain Event-B as suggested by Razali et al [81]. They have done an empirical assessment that compares the comprehensibility of UML-B and Event-B by conducting two controlled experiments. The results indicate that UML-B accelerates the participants of the experiments in understanding a model.

Based on experience modelling the ATM case study, the visualisation in UML-B diagrams helps in understanding what is going on in a refinement. For instance, when modelling a refinement in plain Event-B, a variable is dropped and a new attribute is introduced. An invariant is constructed to relate these attributes using the Rodin interactive prover. However, a modeller might not really understand what the invariant means. Or the modeller may understand the invariant but they may not realise that the refinement is difficult to understand by other people. But, when the same refinement is modelled in UML-B, the diagrams may become an aid for the modeller to visualize what is going on by modelling graphically the new states and relating them with the old states of the more abstract machine.

For example, for the ATM case study, in the initial development, the decomposed components ATM, bank and middleware were based on the sixth refinement machine *ATM_R6* where the states of the state machines are partitioned between the ATM and bank components except for the states *sentReqWD*, *sentReqCB*, *sendRspCB*, *sendRspWDFail* and *sentRspWDOK*. These states do not belong to either the ATM or the bank component. These states were also not in the middleware component. The middleware component contains five message classes *reqwdmsg* (represents withdrawal request messages), *re-*

qcbmsg (check balance request messages), *rspwdokmsg* (successful response messages), *rspwdfailmsg* (unsuccessful response messages) and *rspcbmsg* (check balance response messages). The intention was to model the communication between components using message passing. The three components are composed in a composed machine and the composed machine refines the machine *ATM_R6*. Using the Rodin interactive prover, one of the invariants which needed to be added is

$$\forall\, m \cdot\ m \in reqwdmsg \Rightarrow msg\_atm(m) \in sentReqWD$$

The invariant specified a replacement of the state *sentReqWD* of the abstract machine *ATM_R6* in the composed machine (i.e., refinement machine). However, it is difficult to understand how the states *sentReqWD*, *sentReqCB*, *sendRspCB*, *sendRspWDFail* and *sentRspWDOK* suddenly were removed from the state machines in the refinement(i.e., middleware component). The refinement is a valid refinement but it is difficult to understand because of the refinement gap is quite big. After reviewing the state machine diagrams of the more abstract machine and the class diagram in the refinement, we realised that we have skipped modelling the states of the middleware with respect to the ATM as in Section 8.6.3. The machine *mMW* is then refined by *mMW_R1* which introduces the states of types messages that replace the states in *mMW*. With the refinement gap filled, it is easier to describe the development of the ATM case study.

9.6 Comparison of UML-B to Goal Diagram and Event Refinement Diagram

This section describes and compares related work on visualisation for Event-B refinement which are goal diagrams and event-refinement diagrams with UML-B.

Ball has introduce the incremental development process of a multiagent systems in her thesis [15]. She introduced the goal diagram and relationships goals to be used in the incremental process. The goal diagram is based on KAOS [11] method of goal oriented requirement engineering. The goal diagram is a visual abstraction of the Event-B models as an informal representation of Event-B models to aid understanding. A goal diagram will have a root goal which may be elaborated into more goals producing a tree structure. The goals are related by four relationships which are *THEN*, *AND*, *OR* and *XOR*. In a goal diagram, there might be endpoint goals which represent early terminations. The goal diagrams and goal relationships are based on the concepts of goals and interactions of agent-based systems.

The goal diagrams show ordering of the states of a system from left to right. Each goal in a goal diagram is translated as an event and a state variable in Event-B. In UML-B,

a transition in a state machine diagram is translated as an event and a state in a state machine diagram is translated as a state variable in Event-B.

The *THEN* relationship specifies an order of the states of a system. In UML-B, a state machine diagram models a sequence or an order of the states of a system. The *XOR* relationship represents an exclusive choice between goals. Exclusive means either one of the goals need to be fulfilled and only one of them can be fulfilled in each interaction. In UML-B, this *XOR* relationship is similar to having more than one outgoing transitions from a state.

The *AND* relationship specifies that all the related goals are required to be fulfilled before the next goals can be fulfilled. The inclusive *OR* specifies that either one of the goals or both of the goals can be fulfilled before the next goals can be fulfilled. The *AND* and *OR* relationship are not represented in the state machines of UML-B.

Figure 9.1 shows an example of a goal diagram for an ATM transactions. The first level of elaboration model a sequence of interactions with an ATM and also models options. The *THEN* relationship specifies that the *start* goal must be fulfilled before *insertCard* and is followed by an option either to *withdraw*, *checkBalance* or *ejectCard*. The option is represented by labelling with *XOR* among the option goals. In the next level, the *insertCard*, *withdraw* and *checkBalance* goals are elaborated. The *insertCard* subgoal is followed by an option where either the card validation is successful(*validateCardOK*) or not(*validateCardFail*). If the card validation is successful, then the user can proceed with one of the options of withdrawal, check balance or eject card. If the card validation fails, the next goal is to terminate the interaction with the ATM (*ejectCard1*). *ejectCard1* is an example of an endpoint goal and is highlighted by a circle with inner filled circle. Both the *withdraw* and *checkBalance* goals are elaborated by the respective subgoals *withdraw* and *checkBalance*. Then, these subgoals are followed by the subgoals *ejectCard2* or *ejectCard3* respectively.

The first elaboration level of a goal diagram corresponds to an abstract Event-B machine. Each subsequent level corresponds to each subsequent refinement in Event-B. Likewise, the first level of a goal diagram corresponds to the top level hierarchy of a state machine diagram in UML-B. Each subsequent level of a goal diagram corresponds to each subsequent level of nested state machines in UML-B.

Butler has introduced the event refinement diagram in [24]. The diagram notation is based on the JSD diagram by Jackson [65]. The event refinement diagram is a tree structure diagram similar to the goal diagram. The diagram consists of a rounded rectangle and a line that links to rectangles in subsequent levels. The round rectangle represents an event and the line represents the refinement relationship between any two subsequent levels. The line can either be a dashed or solid line. A dashed line means the bottom event refines *skip*. A solid line means the bottom event refines the top event. Any event which may occur repeatedly will have an oval with the keyword *par* above it.

The event refinement diagram is read from left to right indicating a sequential control.

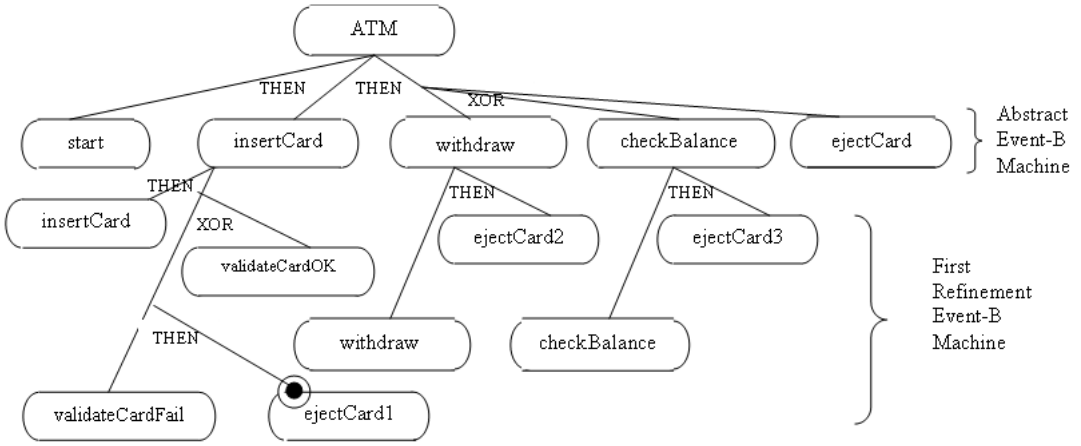


FIGURE 9.1: Example of Goal Diagram

Figure 9.2 shows an example of an event refinement diagram. The root is an abstract event *withdraw*. The next level is a refinement which models the cash withdrawal via an ATM. *start* and *insertCard* are the two new events which refine *skip*. *withdraw* event refines the abstract event. The order of events starts by the *start* event followed by *insertCard* and then *withdraw*. The next level is a refinement which introduces two new events *validateCardOK* and *ejectCard*.

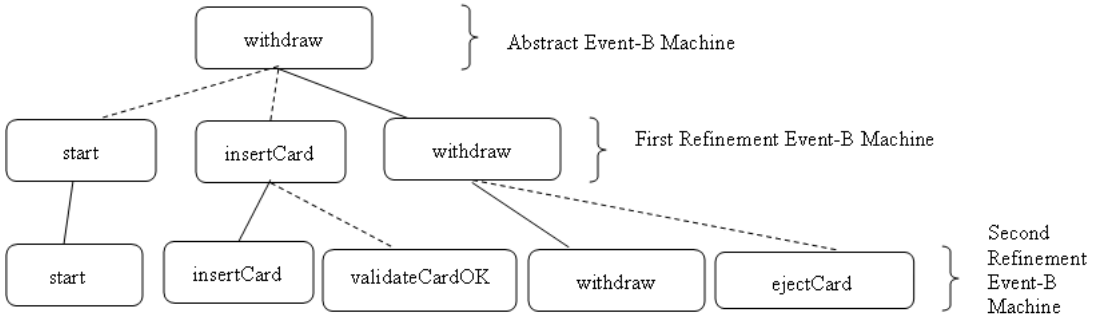


FIGURE 9.2: Example of Event Refinement Diagram

The root of an event refinement diagram corresponds to an abstract Event-B machine. This corresponds to the class event in UML-B. Each subsequent level of an event refinement diagram corresponds to each subsequent refinement in Event-B. The first refinement level of an event refinement diagram corresponds to the top level hierarchy of a state machine diagram in UML-B. Each subsequent level of an event refinement diagram correspond to each subsequent level of nested state machines in UML-B.

The difference between the goal diagrams and event refinement diagrams is that, the goal

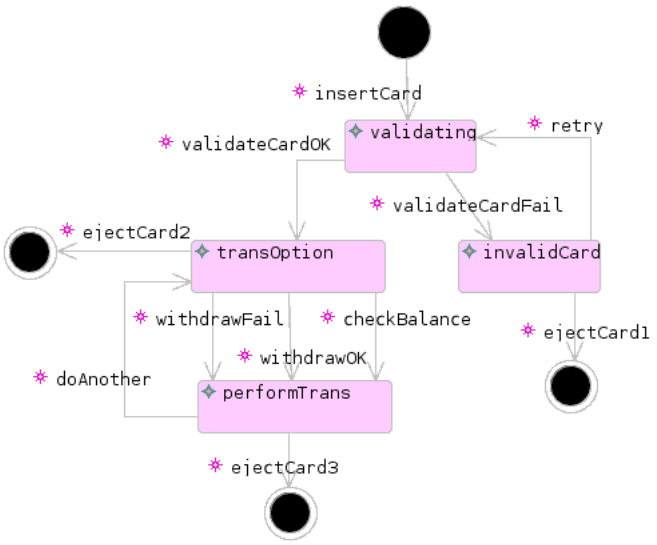


FIGURE 9.3: State Machine of ATM in UML-B

diagrams provide facilities to model choices explicitly using the OR or XOR relationships whereas for the event refinement diagrams, a choice is modelled by separate diagrams. Another difference is that, the event refinement diagrams explicitly model the refinement relation of events between subsequent levels.

The goal diagram and event refinement diagram give an additional structuring to Event-B models like the state machine diagrams in UML-B. Compared to UML-B, both of the diagrams provide an overall visualization of a model. This gives an advantage of showing the refinement relationships between events in different refinement levels which is not available in UML-B. However, the state machine diagrams of UML-B explicitly model the flow of events and the states associated with them. This allows a number of events to merge into a state. For example, in the first refinement level of the goal diagram example, after fulfilling either the goals *withdraw* or *checkBalance*, respectively the goals *ejectCard2* and *ejectCard3* can be fulfilled. In UML-B (Figure 9.3), both of the transitions *withdraw* and *checkBalance* end at the state *performTrans*. From this state the transition *ejectCard3* may be triggered. In other words, the *ejectCard3* follows either the *withdraw* or *checkBalance* transition.

9.7 Future Work

There are more tasks which could be completed in the future. An overview of future tasks are as follows:

- Extend the UML-B metamodel to support parallel state machines. The UML-B Version 1 partly has support for parallel state machines where a state can contain

a number of state machines. However, it does not have support for modelling the transitions. Thus, the UML-B metamodel can be extended in order to support the transitions for multiple parallel state machines in a single state.

- Extend the UML-B tool to support the state function representation of state machines. This work here which supports decomposition focused on the state sets representation of state machines. The other representation i.e., state function has not been catered for in UML-B.

- Add requirements to the ATM case study. The requirement of the ATM case study may evolve. For example, the guards for cash withdrawal may be strengthened to check for overdraft and direct-debit apart from just checking the current account balance. The goal is to see how good is a UML-B model to cope with requirement changes for example how many proofs need to be redone.

- More case studies may be done to validate the extensions and also to explore further extensions to UML-B.

- Add support for managing the traceability of a state machine hierarchy when elaborating a state machine to nested state machines.

- Enhance the composed machine by modelling it graphically showing the included machines and the constituent events.

Appendix A

Requirement Document of the ATM System for Event-B Development

1. Description of the ATM system

The auto teller machine is a machine that allows bank customers to do some of the banking transaction 24 hours per day. It allows bank customers to withdraw cash, check account balance, print mini statement and others. In order to perform these functions through an auto teller machine, bank customers need to use their debit cards which are provided to them by the bank. The case study focused only on the requirements for cash withdrawal.

2. Definitions

Customer - The holder of one or more accounts in the bank.
Debit card - A card provided to a bank customer which authorize access to an account using an ATM.
Account - An account in the bank.
ATM - A machine that allows customers to do cash withdrawals.
Bank - A financial institution that holds accounts for customers and issues debit card

3. Assumptions

There is many ATMs and only one bank.

4. Functional Requirements

4.1 Requirement 1

Description: Customer may insert a debit card to an ATM.

Pre-condition:
1) The ATM is available to use.

Post-condition:
1) The debit card is successfully inserted in the ATM.
2) The ATM prompt a customer to enter a PIN number.

4.2 Requirement 2

Description: The ATM has to check if the entered card is a valid debit card against the entered pin.

Pre-condition:
1) A debit card is in the ATM.
2) The entered pin number is valid.

Post-condition:
1) The ATM will response giving options of transactions.

4.3 Requirement 3

Description: A customer requests a cash withdrawal.

Pre-condition:
1) A debit card is in an ATM.

Post-condition:
1) An ATM will send the withdrawal request to the bank.

4.4 Requirement 4

Description: The withdrawal request is received by the bank.

Pre-condition:
1) The request contains a debit card information.
2) The request contains a withdrawal amount.

Post-condition:
1) The request is added to the bank request list.

4.5 Requirement 5

Description: A cash withdrawal request processed at the bank is successful.

Pre-condition:
1) A debit card is in the ATM.

2) A withdrawal amount is less than or equal to an account balance.

Post-condition:
1) The account balance is deducted by the withdrawal amount.
2) A response information is added to the response list.

4.6 Requirement 6

Description: A successful response is send/receive.

Pre-condition:
1) The response contains an account balance, debit card and successful status information.

Post-condition:
1) The response information is removed from the response list.

4.7 Requirement 7

Description: A cash withdrawal processed at an ATM is successful.

Pre-condition:
1) A debit card is in the ATM.
2) A withdrawal amount is less than or equal to the cash in an ATM.

Post-condition:
1) The cash stored in an ATM is is deducted by the withdrawal amount.

4.8 Requirement 8

Description: A cash withdrawal is not successful.

Pre-condition:
1) A withdrawal amount is more than an account balance.

Post-condition:
1) A response information is added to the response list.

Appendix B

Requirement Document of the ATM System for UML-B Development

1. Description of the ATM system

The auto teller machine is a machine that allows bank customers to do some of the banking transaction 24 hours per day. It allows bank customers to withdraw cash, check account balance, print mini statement and others. In order to perform these functions through an auto teller machine, bank customers need to use their debit cards which are provided to them by the bank. The case study focused only on the requirements for cash withdrawal and check account balance functions.

2. Definitions

Customer - The holder of one or more accounts in the bank.
Debit card - A card provided to a bank customer which authorize access to an account using an ATM.
Account - An account in the bank.
ATM - A machine that allows customers to do cash withdrawals and checking accounts balance.
Bank - A financial institution that holds accounts for customers and issues debit card

2. Abbreviation

MIN_CASH - The minimum amount of cash stored in an ATM.
MAX_CASH - The maximum amount of cash stored in an ATM.

3. Assumptions

There is a notion of a valid and invalid card. Valid card is a card which is issued by the bank and is valid to use for ATM transactions (i.e., ATM card). Invalid card is a card which is not issued by the bank and cannot be used with an ATM. There is many ATMs and only one bank.

4. Functional Requirements

4.1 Requirement 1

Description: Initialisation of cash in the ATM.

Input: ATM is loaded with MAX_CASH amount of cash.

Post-condition:
1) The amount of cash in an ATM is set to MAX_CASH.

4.2 Requirement 2

Description: Reload stock cash in the ATM.

Pre-condition:
1) The ATM is available to use.
2) The stock cash is less than the MIN_CASH.

Post-condition:
1) The amount of cash in an ATM is set to MAX_CASH.

4.3 Requirement 3

Description: Customer may insert a debit card to an ATM.

Pre-condition:
1) The ATM is available to use.

Post-condition:
1) The debit card is successfully inserted in the ATM.
2) The ATM prompt a customer to enter a PIN number.

4.4 Requirement 4

Description: The ATM has to check if the entered card is a valid debit card against the entered pin.

Pre-condition:
1) A debit card is in the ATM.
2) The entered pin number is valid.

Post-condition:
1) The ATM will response giving options of transactions.

4.5 Requirement 5

Description: The debit card inserted in an ATM is validated as failed.

Pre-condition:
1) An ATM card is in an ATM.
2) The entered pin number is invalid.

Post-condition:
1) An ATM will respond giving options to eject card or retry entering a pin.

4.5 Requirement 6

Description: A customer requests a cash withdrawal.

Pre-condition:
1) A debit card is in an ATM.
2) The amount of cash in an ATM is greater than the MIN_CASH.
3) A withdrawal amount is less than or equal to the MIN_CASH.

Post-condition:
1) An ATM will send the withdrawal request to the bank with the withdrawal amount.

4.7 Requirement 7

Description: A customer requests to check an account balance.

Pre-condition:
1) A debit card is in an ATM.

Post-condition:
1) An ATM will send the check balance request to the bank.

4.8 Requirement 8

Description: The withdrawal request is received by the bank.

Pre-condition:
1) The request contains a debit card information.
2) The request contains a withdrawal amount.

Post-condition:
1) The bank will copy the debit card information.
2) The bank will copy the withdrawal amount.
3) The request is added to the bank request list.

4.9 Requirement 9

Description: The check balance request is received by the bank.

Pre-condition:
1) The request contains a debit card information.

Post-condition:
1) The bank will copy the debit card information.
2) The request is added to the bank request list.

4.10 Requirement 10

Description: The bank sends a successful withdrawal, failure withdrawal or check balance response.

Pre-condition:
1) The response contains an account balance.

Post-condition:
1) The response information is removed from the bank's list.
2) The response information is added to the middleware's list.

4.11 Requirement 11

Description: The ATM receive a successful withdrawal, failure withdrawal or check balance response.

Pre-condition:
1) A debit card is in the ATM.
2) The response contains an account balance.

Post-condition:
1) The ATM copy the account balance.

4.12 Requirement 12

Description: A cash withdrawal request is successful.

Pre-condition:
1) A debit card is in the ATM.
2) A withdrawal amount is less than or equal to the account balance.
3) A withdrawal amount is less than or equal to the cash in an ATM.

Post-condition:
1) The account balance is deducted by the withdrawal amount.
2) The cash stored in an ATM is is deducted by the withdrawal amount.
4) The ATM will display the account balance.

5) The ATM will prompt an option either to do another transaction or ejects the ATM card.

4.13 Requirement 13

Description: A cash withdrawal is not successful.

Pre-condition:
1) A debit card is in the ATM.
2) A withdrawal amount is more than an account balance.

Post-condition:
1) The ATM will display the account balance.
2) The ATM will prompt an option either to do another transaction or ejects the ATM card.

4.14 Requirement 14

Description: A check account balance is successful.

Pre-condition:
1) A debit card is in the ATM.

Post-condition:
1) The ATM will display the account balance.
2) The ATM will prompt an option either to do another transaction or ejects the ATM card.

4.15 Requirement 15

Description: User can choose to cancel a transaction if the PIN entered is invalid or the PIN is valid but the user do not want to proceed with a transaction.

Pre-condition:
1) A debit card is in the ATM.

Post-condition:
1) The ATM ejects the ATM card.

Appendix C

ATM Case Study: Using Rodin Event-B

A.1 First Refinement

MACHINE ATM_R1

REFINES ATMM

SEES ATMC_E1

VARIABLES

account

bal

cards

card_account

pin

pinNo

atm

atm_card

atm_cash

active_atm

idle

validating

transactionOption

performWithdrawal

endWithdrawal

INVARIANTS

inv2 : $cards \in \mathbb{P}(CARD)$

inv3 : $card\_account \in cards \rightarrowtail account$

inv4 : $pin \in \mathbb{P}(PIN)$

inv5 : $pinNo \in cards \rightarrow pin$

inv6 : $atm \in \mathbb{P}(ATM)$

inv13 : $active\_atm \subseteq atm$

inv7 : $atm\_card \in active\_atm \rightarrow cards$

inv11 : $atm\_cash \in atm \rightarrow \mathbb{N}$

inv1 : $idle \subseteq atm$

inv8 : $validating \subseteq active\_atm$

inv9 : $transactionOption \subseteq active\_atm$

inv10 : $performWithdrawal \subseteq active\_atm$

inv19 : $endWithdrawal \subseteq active\_atm$

inv12 : $idle \cap validating = \varnothing$

inv14 : $idle \cap transactionOption = \varnothing$

inv15 : $idle \cap performWithdrawal = \varnothing$

inv20 : $idle \cap endWithdrawal = \varnothing$

inv16 : $validating \cap transactionOption = \varnothing$

inv17 : $validating \cap performWithdrawal = \varnothing$

inv21 : $validating \cap endWithdrawal = \varnothing$

inv18 : $transactionOption \cap performWithdrawal = \varnothing$

inv22 : $transactionOption \cap endWithdrawal = \varnothing$

inv23 : $performWithdrawal \cap endWithdrawal = \varnothing$

EVENTS

Initialisation

begin

act1 : $account := \varnothing$

act2 : $bal := \varnothing$

act7 : $cards := \varnothing$

act8 : $card\_account := \varnothing$

act9 : $pin := \varnothing$

act10 : $pinNo := \varnothing$

act11 : atm := ∅

act12 : atm_card := ∅

act16 : atm_cash := ∅

act18 : active_atm := ∅

act3 : idle := ∅

act4 : validating := ∅

act5 : transactionOption := ∅

act6 : performWithdrawal := ∅

act13 : endWithdrawal := ∅

end

Event *insertCard* $\widehat{=}$

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ idle

then

act1 : atm_card(at) := c

act3 : active_atm := active_atm ∪ {at}

act2 : validating := validating ∪ {at}

act4 : idle := idle \ {at}

end

Event *validateCardOK* $\widehat{=}$

any

c

at

p

where

grd1 : c ∈ cards

grd2 : at ∈ validating

grd7 : p ∈ pin

grd8 : pinNo(c) = p

grd3 : atm_card(at) = c

then

act2 : transactionOption := transactionOption ∪ {at}

act1 : validating := validating \ {at}

end

Event *withdrawBankOK* ≙

refines *withdraw*

any

ac

am

c

at

where

grd1 : ac ∈ account

grd2 : am ∈ ℕ

grd9 : c ∈ cards

grd10 : at ∈ transactionOption

grd4 : am ≤ bal(ac)

grd3 : atm_card(at) = c

grd6 : card_account(c) = ac

then

act1 : bal(ac) := bal(ac) − am

act3 : performWithdrawal := performWithdrawal ∪ {at}

act2 : transactionOption := transactionOption \ {at}

end

Event *withdrawFail* ≙

any

ac

am

c

at

where

grd1 : ac ∈ account

grd2 : am ∈ ℕ

grd3 : c ∈ cards

grd4 : at ∈ transactionOption

grd5 : $am > bal(ac)$

grd6 : $atm\_card(at) = c$

grd7 : $card\_account(c) = ac$

then

act1 : $performWithdrawal := performWithdrawal \cup \{at\}$

act2 : $transactionOption := transactionOption \setminus \{at\}$

end

Event *withdrawATM* $\widehat{=}$

any

ac

am

c

at

where

grd2 : $ac \in account$

grd3 : $am \in \mathbb{N}$

grd4 : $c \in cards$

grd5 : $at \in performWithdrawal$

grd8 : $atm\_cash(at) \geq am$

grd9 : $atm\_card(at) = c$

grd10 : $card\_account(c) = ac$

then

act1 : $atm\_cash(at) := atm\_cash(at) - am$

act2 : $performWithdrawal := performWithdrawal \setminus \{at\}$

act3 : $endWithdrawal := endWithdrawal \cup \{at\}$

end

Event *ejectCard* $\widehat{=}$

any

c

at

where

grd1 : $c \in cards$

grd2 : $at \in endWithdrawal$

grd5 : $atm\_card(at) = c$

then

act2 : $\texttt{atm\_card} := \texttt{atm\_card} \setminus \{\texttt{at} \mapsto \texttt{c}\}$

act3 : $\texttt{endWithdrawal} := \texttt{endWithdrawal} \setminus \{\texttt{at}\}$

act1 : $\texttt{idle} := \texttt{idle} \cup \{\texttt{at}\}$

act4 : $\texttt{active\_atm} := \texttt{active\_atm} \setminus \{\texttt{at}\}$

end

END

A.2 Second Refinement

MACHINE ATM_R2

REFINES ATM_R1

SEES ATMC_E1

VARIABLES

account

bal

cards

card_account

pin

pinNo

atm

atm_card

atm_cash

active_atm

idle

validating

processingBank

complete

trans

waiting

endWithdrawal

INVARIANTS

inv4 : $trans \subseteq active\_atm$

inv5 : $waiting \subseteq active\_atm$

inv1 : $processingBank \subseteq active\_atm$

inv2 : $complete \subseteq active\_atm$

inv6 : $transactionOption = trans \cup waiting$

inv7 : $performWithdrawal = processingBank \cup complete$

inv9 : $trans \cap waiting = \varnothing$

inv10 : $processingBank \cap complete = \varnothing$

inv3 : $waiting \cap complete = \varnothing$

inv8 : waiting ∩ processingBank = ∅

inv11 : trans ∩ processingBank = ∅

EVENTS

Initialisation

begin

act1 : account := ∅

act2 : bal := ∅

act7 : cards := ∅

act8 : card_account := ∅

act9 : pin := ∅

act10 : pinNo := ∅

act11 : atm := ∅

act12 : atm_card := ∅

act16 : atm_cash := ∅

act18 : active_atm := ∅

act3 : idle := ∅

act4 : validating := ∅

act14 : processingBank := ∅

act15 : complete := ∅

act19 : trans := ∅

act20 : waiting := ∅

act5 : endWithdrawal := ∅

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ idle

then

act1 : atm_card(at) := c

act3 : active_atm := active_atm ∪ {at}

act2 : validating := validating ∪ {at}

act4 : idle := idle \ {at}

end

Event *validateCardOK* ≙

refines *validateCardOK*

any

c

at

p

where

grd1 : c ∈ cards

grd2 : at ∈ validating

grd7 : p ∈ pin

grd8 : pinNo(c) = p

grd3 : atm_card(at) = c

then

act2 : trans := trans ∪ {at}

act1 : validating := validating \ {at}

end

Event *request* ≙

any

at

c

am

where

grd1 : at ∈ trans

grd3 : c ∈ cards

grd2 : atm_card(at) = c

grd4 : am ∈ ℕ

then

act1 : trans := trans \ {at}

act2 : waiting := waiting ∪ {at}

end

Event *withdrawBankOK* ≙

refines *withdrawBankOK*

any

ac

am

c

at

where

grd1 : $ac \in account$

grd2 : $am \in \mathbb{N}$

grd9 : $c \in cards$

grd10 : $at \in waiting$

grd4 : $am \leq bal(ac)$

grd3 : $atm\_card(at) = c$

grd6 : $card\_account(c) = ac$

then

act1 : $bal(ac) := bal(ac) - am$

act3 : $processingBank := processingBank \cup \{at\}$

act2 : $waiting := waiting \setminus \{at\}$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

ac

am

c

at

where

grd1 : $ac \in account$

grd2 : $am \in \mathbb{N}$

grd3 : $c \in cards$

grd4 : $at \in waiting$

grd5 : $am > bal(ac)$

grd6 : $atm\_card(at) = c$

grd7 : $card\_account(c) = ac$

then

act1 : processingBank := processingBank ∪ {at}

act2 : waiting := waiting \ {at}

end

Event *responseOK* ≙

any

at

where

grd1 : at ∈ processingBank

then

act1 : processingBank := processingBank \ {at}

act2 : complete := complete ∪ {at}

end

Event *withdrawATM* ≙

refines *withdrawATM*

any

ac

am

c

at

where

grd2 : ac ∈ account

grd3 : am ∈ ℕ

grd4 : c ∈ cards

grd5 : at ∈ complete

grd8 : atm_cash(at) ≥ am

grd9 : atm_card(at) = c

grd10 : card_account(c) = ac

then

act1 : atm_cash(at) := atm_cash(at) − am

act2 : complete := complete \ {at}

act3 : endWithdrawal := endWithdrawal ∪ {at}

end

Event *ejectCard* ≙

refines *ejectCard*

any

c

at

where

grd1 : $c \in cards$

grd2 : $at \in endWithdrawal$

grd5 : $atm\_card(at) = c$

then

act2 : $atm\_card := atm\_card \setminus \{at \mapsto c\}$

act3 : $endWithdrawal := endWithdrawal \setminus \{at\}$

act1 : $idle := idle \cup \{at\}$

act4 : $active\_atm := active\_atm \setminus \{at\}$

end

Event $responseNOTOK \mathrel{\widehat{=}}$

any

at

where

grd1 : $at \in processingBank$

then

act1 : $processingBank := processingBank \setminus \{at\}$

act2 : $complete := complete \cup \{at\}$

end

END

A.3 Third Refinement

MACHINE ATM_R3

REFINES ATM_R2

SEES ATMC_E1

VARIABLES

account

bal

cards

card_account

pin

pinNo

atm

atm_card

atm_cash

active_atm

idle

validating

complete

trans

endWithdrawal

conversation

bprocessing

req

rsp

INVARIANTS

inv1 : conversation $\subseteq$ active_atm

inv8 : req $\subseteq$ active_atm

inv2 : bprocessing $\subseteq$ active_atm

inv9 : rsp $\subseteq$ active_atm

inv10 : rsp $\cap$ complete $= \varnothing$

inv3 : bprocessing $\cap$ trans $= \varnothing$

inv4 : bprocessing $\cap$ complete $= \varnothing$

inv5 : bprocessing ∩ validating = ∅

inv7 : bprocessing ⊆ waiting

inv11 : req ⊆ waiting

inv12 : req ∩ trans = ∅

inv14 : req ∩ validating = ∅

inv13 : trans ∩ validating = ∅

inv15 : rsp ∩ bprocessing = ∅

inv16 : req ∩ rsp = ∅

inv17 : trans ∩ rsp = ∅

inv18 : validating ∩ rsp = ∅

inv19 : idle ∩ rsp = ∅

inv20 : bprocessing ∩ idle = ∅

inv21 : req ∩ idle = ∅

inv22 : trans ∩ idle = ∅

inv23 : rsp ⊆ processingBank

inv24 : endWithdrawal ∩ conversation = ∅

inv25 : trans ∩ endWithdrawal = ∅

inv26 : complete ∩ conversation = ∅

inv27 : trans ∩ complete = ∅

EVENTS

Initialisation

begin

act1 : account := ∅

act2 : bal := ∅

act7 : cards := ∅

act8 : card_account := ∅

act9 : pin := ∅

act10 : pinNo := ∅

act11 : atm := ∅

act12 : atm_card := ∅

act16 : atm_cash := ∅

act18 : active_atm := ∅

act3 : idle := ∅

act4 : validating := ∅

act15 : complete := ∅

act19 : trans := ∅

act5 : endWithdrawal := ∅

act6 : conversation := ∅

act13 : bprocessing := ∅

act17 : req := ∅

act20 : rsp := ∅

end

Event *insertCard* ≙

refines *insertCard*

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ idle

then

act1 : atm_card(at) := c

act3 : active_atm := active_atm ∪ {at}

act2 : validating := validating ∪ {at}

act4 : idle := idle \ {at}

end

Event *validateCardOK* ≙

refines *validateCardOK*

any

c

at

p

where

grd1 : c ∈ cards

grd2 : at ∈ validating

grd7 : p ∈ pin

grd8 : pinNo(c) = p

grd3 : atm_card(at) = c

then

act2 : trans := trans ∪ {at}

act1 : validating := validating \ {at}

end

Event *request* ≙

refines *request*

any

at

c

am

where

grd1 : at ∈ trans

grd3 : c ∈ cards

grd2 : atm_card(at) = c

grd4 : am ∈ ℕ

then

act1 : trans := trans \ {at}

act2 : conversation := conversation ∪ {at}

act3 : req := req ∪ {at}

end

Event *receiveRequest* ≙

any

at

c

where

grd1 : at ∈ req

grd3 : c ∈ cards

grd2 : atm_card(at) = c

then

act3 : bprocessing := bprocessing ∪ {at}

end

Event *withdrawBankOK* ≙

refines *withdrawBankOK*

any

ac

am

c

at

where

grd1 : $ac \in account$

grd2 : $am \in \mathbb{N}$

grd9 : $c \in cards$

grd10 : $at \in bprocessing$

grd4 : $am \leq bal(ac)$

grd3 : $atm\_card(at) = c$

grd6 : $card\_account(c) = ac$

then

act1 : $bal(ac) := bal(ac) - am$

act3 : $rsp := rsp \cup \{at\}$

act2 : $bprocessing := bprocessing \setminus \{at\}$

act4 : $req := req \setminus \{at\}$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

ac

am

c

at

where

grd1 : $ac \in account$

grd2 : $am \in \mathbb{N}$

grd3 : $c \in cards$

grd4 : $at \in bprocessing$

grd5 : $am > bal(ac)$

grd6 : $atm\_card(at) = c$

grd7 : $card\_account(c) = ac$

then

act1 : rsp := rsp ∪ {at}

act2 : bprocessing := bprocessing \ {at}

act3 : req := req \ {at}

end

Event *responseOK* ≙

refines *responseOK*

any

at

where

grd1 : at ∈ rsp

then

act1 : rsp := rsp \ {at}

act2 : complete := complete ∪ {at}

act3 : conversation := conversation \ {at}

end

Event *withdrawATM* ≙

refines *withdrawATM*

any

ac

am

c

at

where

grd2 : ac ∈ account

grd3 : am ∈ ℕ

grd4 : c ∈ cards

grd5 : at ∈ complete

grd8 : atm_cash(at) ≥ am

grd9 : atm_card(at) = c

grd10 : card_account(c) = ac

then

act1 : atm_cash(at) := atm_cash(at) − am

act2 : complete := complete \ {at}

act3 : endWithdrawal := endWithdrawal ∪ {at}

end

Event *ejectCard* $\widehat{=}$

refines *ejectCard*

any

c

at

where

grd1 : $c \in cards$

grd2 : $at \in endWithdrawal$

grd5 : $atm\_card(at) = c$

then

act2 : $atm\_card := atm\_card \setminus \{at \mapsto c\}$

act3 : $endWithdrawal := endWithdrawal \setminus \{at\}$

act1 : $idle := idle \cup \{at\}$

act4 : $active\_atm := active\_atm \setminus \{at\}$

end

Event *responseNOTOK* $\widehat{=}$

refines *responseNOTOK*

any

at

where

grd1 : $at \in rsp$

then

act1 : $rsp := rsp \setminus \{at\}$

act2 : $complete := complete \cup \{at\}$

act3 : $conversation := conversation \setminus \{at\}$

end

END

A.4 Fourth Refinement

MACHINE ATM_R4

REFINES ATM_R3

SEES ATMC_E2

VARIABLES

account

bal

cards

card_account

pin

pinNo

atm

atm_card

atm_cash

active_atm

idle

validating

complete

trans

endWithdrawal

conversation

bprocessing

reqmsg

rspmsg

INVARIANTS

inv1 : $reqmsg \subseteq REQ\_MSG$

inv2 : $rspmsg \subseteq RSP\_MSG$

inv3 : $\forall m \cdot m \in reqmsg \Rightarrow msg\_atm(m) \in req$

inv4 : $\forall m \cdot m \in rspmsg \Rightarrow msg\_atm(m) \in rsp$

inv5 : $\forall m \cdot msg\_atm(m) \in idle \Rightarrow m \notin rspmsg$

inv6 : $\forall m, m0 \cdot m \in reqmsg \land m0 \in reqmsg \land m \neq m0 \Rightarrow msg\_atm(m) \neq msg\_atm(m0)$

inv7 : $\forall m, m0 \cdot m \in rspmsg \land m0 \in rspmsg \land m \neq m0 \Rightarrow msg\_atm(m) \neq msg\_atm(m0)$

inv8 : reqmsg ∩ rspmsg = ∅

EVENTS

Initialisation

begin

act1 : account := ∅

act2 : bal := ∅

act7 : cards := ∅

act8 : card_account := ∅

act9 : pin := ∅

act10 : pinNo := ∅

act11 : atm := ∅

act12 : atm_card := ∅

act16 : atm_cash := ∅

act18 : active_atm := ∅

act3 : idle := ∅

act4 : validating := ∅

act15 : complete := ∅

act19 : trans := ∅

act5 : endWithdrawal := ∅

act6 : conversation := ∅

act13 : bprocessing := ∅

act14 : reqmsg := ∅

act21 : rspmsg := ∅

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ idle

then

act1 : atm_card(at) := c

act3 : active_atm := active_atm ∪ {at}

act2 : validating := validating ∪ {at}

act4 : idle := idle \ {at}

end

Event *validateCardOK* ≙

refines *validateCardOK*

any

c

at

p

where

grd1 : c ∈ cards

grd2 : at ∈ validating

grd7 : p ∈ pin

grd8 : pinNo(c) = p

grd3 : atm_card(at) = c

then

act2 : trans := trans ∪ {at}

act1 : validating := validating \ {at}

end

Event *request* ≙

refines *request*

any

at

m

c

am

where

grd1 : at ∈ trans

grd2 : m ∈ REQ_MSG

grd5 : c ∈ cards

grd3 : msg_atm(m) = at

grd4 : atm_card(at) = c

grd6 : am ∈ ℕ

then

act1 : trans := trans \ {at}

act2 : conversation := conversation ∪ {at}

act3 : reqmsg := reqmsg ∪ {m}

end

Event *receiveRequest* ≙

refines *receiveRequest*

any

at

m

c

where

grd2 : m ∈ reqmsg

grd1 : at ∈ conversation

grd4 : c ∈ cards

grd3 : msg_atm(m) = at

grd5 : atm_card(at) = c

then

act3 : bprocessing := bprocessing ∪ {at}

end

Event *withdrawBankOK* ≙

refines *withdrawBankOK*

any

ac

am

c

at

m

m0

where

grd1 : ac ∈ account

grd2 : am ∈ ℕ

grd9 : c ∈ cards

grd10 : at ∈ bprocessing

grd4 : $am \leq bal(ac)$

grd3 : $atm\_card(at) = c$

grd6 : $card\_account(c) = ac$

grd8 : $m \in reqmsg$

grd11 : $msg\_atm(m) = at$

grd5 : $m0 \in RSP\_MSG$

grd13 : $msg\_atm(m0) = at$

then

act1 : $bal(ac) := bal(ac) - am$

act2 : $bprocessing := bprocessing \setminus \{at\}$

act5 : $reqmsg := reqmsg \setminus \{m\}$

act3 : $rspmsg := rspmsg \cup \{m0\}$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

ac

am

c

at

m

m0

where

grd1 : $ac \in account$

grd2 : $am \in \mathbb{N}$

grd3 : $c \in cards$

grd4 : $at \in bprocessing$

grd5 : $am > bal(ac)$

grd6 : $atm\_card(at) = c$

grd7 : $card\_account(c) = ac$

grd8 : $m \in reqmsg$

grd9 : $msg\_atm(m) = at$

grd10 : $msg\_atm(m0) = at$

grd11 : $m \in RSP\_MSG$

then

act1 : $bprocessing := bprocessing \setminus \{at\}$

act2 : reqmsg := reqmsg \ {m}

act3 : rspmsg := rspmsg ∪ {m0}

end

Event *responseOK* ≙

refines *responseOK*

any

at

m0

where

grd1 : at ∈ conversation

grd2 : m0 ∈ rspmsg

grd3 : msg_atm(m0) = at

then

act1 : rspmsg := rspmsg \ {m0}

act2 : complete := complete ∪ {at}

act3 : conversation := conversation \ {at}

end

Event *withdrawATM* ≙

refines *withdrawATM*

any

ac

am

c

at

where

grd2 : ac ∈ account

grd3 : am ∈ ℕ

grd4 : c ∈ cards

grd5 : at ∈ complete

grd8 : atm_cash(at) ≥ am

grd9 : atm_card(at) = c

grd10 : card_account(c) = ac

then

act1 : atm_cash(at) := atm_cash(at) − am

act2 : complete := complete \ {at}

act3 : endWithdrawal := endWithdrawal ∪ {at}

end

Event *ejectCard* ≙

refines *ejectCard*

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ endWithdrawal

grd5 : atm_card(at) = c

then

act2 : atm_card := atm_card \ {at ↦ c}

act3 : endWithdrawal := endWithdrawal \ {at}

act1 : idle := idle ∪ {at}

act4 : active_atm := active_atm \ {at}

end

Event *responseNOTOK* ≙

refines *responseNOTOK*

any

at

m0

where

grd1 : at ∈ conversation

grd2 : m0 ∈ rspmsg

grd3 : msg_atm(m0) = at

then

act1 : rspmsg := rspmsg \ {m0}

act2 : complete := complete ∪ {at}

act3 : conversation := conversation \ {at}

end

END

A.5 Fifth Refinement

MACHINE ATM_R5

REFINES ATM_R4

SEES ATMC_E3

VARIABLES

account

bal

cards

card_account

pin

pinNo

atm

atm_card

atm_cash

active_atm

idle

validating

complete

trans

endWithdrawal

conversation

bprocessing

reqmsg

rspmsg

INVARIANTS

inv1 : $\top$

inv2 : $\forall m \cdot m \in \text{reqmsg} \Rightarrow \text{atm\_card}(\text{msg\_atm}(m)) = \text{msg\_card}(m)$

EVENTS

Initialisation

begin

act1 : account $:= \varnothing$

act2 : bal := ∅

act7 : cards := ∅

act8 : card_account := ∅

act9 : pin := ∅

act10 : pinNo := ∅

act11 : atm := ∅

act12 : atm_card := ∅

act16 : atm_cash := ∅

act18 : active_atm := ∅

act3 : idle := ∅

act4 : validating := ∅

act15 : complete := ∅

act19 : trans := ∅

act5 : endWithdrawal := ∅

act6 : conversation := ∅

act13 : bprocessing := ∅

act14 : reqmsg := ∅

act21 : rspmsg := ∅

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ idle

then

act4 : idle := idle \ {at}

act3 : active_atm := active_atm ∪ {at}

act2 : validating := validating ∪ {at}

act1 : atm_card(at) := c

end

Event *validateCardOK* $\widehat{=}$

refines *validateCardOK*

any

c

at

p

where

grd1 : $c \in cards$

grd2 : $at \in validating$

grd7 : $p \in pin$

grd8 : $pinNo(c) = p$

grd3 : $atm\_card(at) = c$

then

act2 : $trans := trans \cup \{at\}$

act1 : $validating := validating \setminus \{at\}$

end

Event *request* $\widehat{=}$

refines *request*

any

at

m

am

c

where

grd1 : $at \in trans$

grd2 : $m \in REQ\_MSG$

grd5 : $c \in cards$

grd8 : $atm\_card(at) = c$

grd3 : $msg\_atm(m) = at$

grd4 : $am \in \mathbb{N}$

grd6 : $msg\_card(m) = c$

grd7 : $reqmsg\_wdAmount(m) = am$

then

act1 : $trans := trans \setminus \{at\}$

act2 : $conversation := conversation \cup \{at\}$

act3 : $reqmsg := reqmsg \cup \{m\}$

end

Event *receiveRequest* $\widehat{=}$

refines *receiveRequest*

any

at

m

am

c

where

grd2 : $m \in reqmsg$

grd1 : $at \in conversation$

grd3 : $msg\_atm(m) = at$

grd4 : $am \in \mathbb{N}$

grd5 : $c \in cards$

grd6 : $msg\_card(m) = c$

grd7 : $reqmsg\_wdAmount(m) = am$

then

act3 : $bprocessing := bprocessing \cup \{at\}$

end

Event *withdrawBankOK* $\widehat{=}$

refines *withdrawBankOK*

any

ac

am

c

at

m

m0

where

grd1 : $ac \in account$

grd2 : $am \in \mathbb{N}$

grd9 : $c \in cards$

grd10 : $at \in bprocessing$

grd4 : $am \leq bal(ac)$

grd6 : $card\_account(c) = ac$

grd8 : $m \in reqmsg$

grd5 : m0 ∈ RSP_MSG

grd11 : msg_atm(m) = at

grd13 : msg_atm(m0) = at

grd14 : msg_card(m) = c

grd15 : reqmsg_wdAmount(m) = am

grd3 : msg_card(m0) = c

grd12 : rspmsg_status(m0) = OK

grd16 : rspmsg_bal(m0) = bal(ac)

then

act1 : bal(ac) := bal(ac) − am

act2 : bprocessing := bprocessing \ {at}

act5 : reqmsg := reqmsg \ {m}

act7 : rspmsg := rspmsg ∪ {m0}

end

Event *withdrawFail* ≙

refines *withdrawFail*

any

ac

am

c

at

m0

m

where

grd1 : ac ∈ account

grd2 : am ∈ ℕ

grd3 : c ∈ cards

grd4 : at ∈ bprocessing

grd5 : am > bal(ac)

grd6 : card_account(c) = ac

grd7 : m ∈ reqmsg

grd15 : m0 ∈ RSP_MSG

grd8 : msg_atm(m) = at

grd10 : msg_card(m) = c

grd11 : reqmsg_wdAmount(m) = am

grd9 : msg_atm(m0) = at

grd12 : msg_card(m0) = c

grd13 : rspmsg_status(m0) = NOT_OK

grd14 : rspmsg_bal(m0) = bal(ac)

then

act1 : bprocessing := bprocessing \ {at}

act2 : reqmsg := reqmsg \ {m}

act3 : rspmsg := rspmsg ∪ {m0}

end

Event *responseOK* ≙

refines *responseOK*

any

at

m0

c

ac

where

grd1 : at ∈ conversation

grd2 : m0 ∈ rspmsg

grd3 : msg_atm(m0) = at

grd5 : c ∈ cards

grd9 : ac ∈ account

grd7 : msg_card(m0) = c

grd6 : rspmsg_status(m0) = OK

grd8 : rspmsg_bal(m0) = bal(ac)

then

act1 : rspmsg := rspmsg \ {m0}

act2 : complete := complete ∪ {at}

act3 : conversation := conversation \ {at}

end

Event *responseNOTOK* ≙

refines *responseNOTOK*

any

at

m0

c

ac

where

grd1 : $\texttt{at} \in \texttt{conversation}$

grd2 : $\texttt{m0} \in \texttt{rspmsg}$

grd3 : $\texttt{c} \in \texttt{cards}$

grd4 : $\texttt{ac} \in \texttt{account}$

grd5 : $\texttt{msg\_atm}(\texttt{m0}) = \texttt{at}$

grd6 : $\texttt{msg\_card}(\texttt{m0}) = \texttt{c}$

grd7 : $\texttt{rspmsg\_status}(\texttt{m0}) = \texttt{NOT\_OK}$

grd8 : $\texttt{rspmsg\_bal}(\texttt{m0}) = \texttt{bal}(\texttt{ac})$

then

act1 : $\texttt{rspmsg} := \texttt{rspmsg} \setminus \{\texttt{m0}\}$

act2 : $\texttt{complete} := \texttt{complete} \cup \{\texttt{at}\}$

act3 : $\texttt{conversation} := \texttt{conversation} \setminus \{\texttt{at}\}$

end

Event *withdrawATM* $\widehat{=}$

refines *withdrawATM*

any

ac

am

c

at

where

grd2 : $\texttt{ac} \in \texttt{account}$

grd3 : $\texttt{am} \in \mathbb{N}$

grd4 : $\texttt{c} \in \texttt{cards}$

grd5 : $\texttt{at} \in \texttt{complete}$

grd8 : $\texttt{atm\_cash}(\texttt{at}) \geq \texttt{am}$

grd9 : $\texttt{atm\_card}(\texttt{at}) = \texttt{c}$

grd10 : $\texttt{card\_account}(\texttt{c}) = \texttt{ac}$

then

act1 : $\texttt{atm\_cash}(\texttt{at}) := \texttt{atm\_cash}(\texttt{at}) - \texttt{am}$

act2 : $\texttt{complete} := \texttt{complete} \setminus \{\texttt{at}\}$

act3 : $\texttt{endWithdrawal} := \texttt{endWithdrawal} \cup \{\texttt{at}\}$

end

Event *ejectCard* $\widehat{=}$

refines *ejectCard*

any

c

at

where

grd1 : $c \in cards$

grd2 : $at \in endWithdrawal$

grd5 : $atm\_card(at) = c$

then

act2 : $atm\_card := atm\_card \setminus \{at \mapsto c\}$

act3 : $endWithdrawal := endWithdrawal \setminus \{at\}$

act1 : $idle := idle \cup \{at\}$

act4 : $active\_atm := active\_atm \setminus \{at\}$

end

END

A. Sixth Refinement

MACHINE ATM_R6

REFINES ATM_R5

SEES ATMC_E3

VARIABLES

account

bal

cards

card_account

pin

pinNo

atm

atm_card

atm_cash

active_atm

idle

validating

complete

trans

endWithdrawal

conversation

bprocessing

msg

INVARIANTS

inv1 : $msg \subseteq MSG$

inv2 : $reqmsg = msg \cap REQ\_MSG$

inv3 : $rspmsg = msg \cap RSP\_MSG$

EVENTS

Initialisation

begin

act1 : $account := \varnothing$

act2 : bal := ∅

act7 : cards := ∅

act8 : card_account := ∅

act9 : pin := ∅

act10 : pinNo := ∅

act11 : atm := ∅

act12 : atm_card := ∅

act16 : atm_cash := ∅

act18 : active_atm := ∅

act3 : idle := ∅

act4 : validating := ∅

act15 : complete := ∅

act19 : trans := ∅

act5 : endWithdrawal := ∅

act6 : conversation := ∅

act13 : bprocessing := ∅

act23 : msg := ∅

end

Event *insertCard* ≙

refines *insertCard*

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ idle

then

act1 : atm_card(at) := c

act3 : active_atm := active_atm ∪ {at}

act2 : validating := validating ∪ {at}

act4 : idle := idle \ {at}

end

Event *validateCardOK* ≙

refines *validateCardOK*

any

c

at

p

where

grd1 : $c \in cards$

grd2 : $at \in validating$

grd7 : $p \in pin$

grd8 : $pinNo(c) = p$

grd3 : $atm\_card(at) = c$

then

act2 : $trans := trans \cup \{at\}$

act1 : $validating := validating \setminus \{at\}$

end

Event *request* $\widehat{=}$

refines *request*

any

at

m

am

c

where

grd1 : $at \in trans$

grd2 : $m \in REQ\_MSG$

grd3 : $msg\_atm(m) = at$

grd4 : $am \in \mathbb{N}$

grd5 : $c \in cards$

grd6 : $msg\_card(m) = c$

grd7 : $reqmsg\_wdAmount(m) = am$

grd8 : $atm\_card(at) = c$

then

act1 : $trans := trans \setminus \{at\}$

act2 : $conversation := conversation \cup \{at\}$

act3 : $msg := msg \cup \{m\}$

end

Event *receiveRequest* $\widehat{=}$

refines *receiveRequest*

any

at

m

am

c

where

grd2 : $m \in msg$

grd8 : $m \in REQ\_MSG$

grd1 : $at \in conversation$

grd3 : $msg\_atm(m) = at$

grd4 : $am \in \mathbb{N}$

grd5 : $c \in cards$

grd6 : $msg\_card(m) = c$

grd7 : $reqmsg\_wdAmount(m) = am$

then

act3 : $bprocessing := bprocessing \cup \{at\}$

end

Event *withdrawBankOK* $\widehat{=}$

refines *withdrawBankOK*

any

ac

am

c

at

m

m0

where

grd1 : $ac \in account$

grd2 : $am \in \mathbb{N}$

grd9 : $c \in cards$

grd10 : $at \in bprocessing$

grd4 : $am < bal(ac)$

grd6 : $card\_account(c) = ac$

grd8 : $m \in msg$

grd3 : $m \in REQ\_MSG$

grd12 : $m0 \in RSP\_MSG$

grd11 : $msg\_atm(m) = at$

grd13 : $msg\_atm(m0) = at$

grd14 : $msg\_card(m) = c$

grd15 : $reqmsg\_wdAmount(m) = am$

grd17 : $msg\_card(m0) = c$

grd18 : $rspmsg\_status(m0) = OK$

grd19 : $rspmsg\_bal(m0) = bal(ac)$

then

act1 : $bal(ac) := bal(ac) - am$

act2 : $bprocessing := bprocessing \setminus \{at\}$

act3 : $msg := (msg \setminus \{m\}) \cup \{m0\}$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

ac

am

c

at

m

m0

where

grd2 : $ac \in account$

grd3 : $am \in \mathbb{N}$

grd4 : $c \in cards$

grd5 : $at \in bprocessing$

grd6 : $m \in msg$

grd15 : $m \in REQ\_MSG$

grd16 : $m0 \in RSP\_MSG$

grd9 : $am > bal(ac)$

grd10 : $card\_account(c) = ac$

grd11 : $msg\_atm(m) = at$

grd12 : $msg\_atm(m0) = at$

grd13 : $msg\_card(m) = c$

grd14 : reqmsg_wdAmount(m) = am

grd17 : msg_card(m0) = c

grd18 : rspmsg_status(m0) = NOT_OK

grd19 : rspmsg_bal(m0) = bal(ac)

then

act2 : bprocessing := bprocessing \ {at}

act1 : msg := (msg \ {m}) ∪ {m0}

end

Event *responseOK* ≙

refines *responseOK*

any

at

m0

c

ac

where

grd1 : at ∈ conversation

grd2 : m0 ∈ msg

grd4 : m0 ∈ RSP_MSG

grd3 : msg_atm(m0) = at

grd5 : c ∈ cards

grd9 : ac ∈ account

grd7 : msg_card(m0) = c

grd6 : rspmsg_status(m0) = OK

grd8 : rspmsg_bal(m0) = bal(ac)

then

act1 : msg := msg \ {m0}

act2 : complete := complete ∪ {at}

act3 : conversation := conversation \ {at}

end

Event *responseNOTOK* ≙

refines *responseNOTOK*

any

at

m0

c

ac

where

grd1 : $at \in conversation$

grd2 : $m0 \in msg$

grd8 : $m0 \in RSP\_MSG$

grd7 : $msg\_atm(m0) = at$

grd3 : $c \in cards$

grd4 : $ac \in account$

grd5 : $msg\_card(m0) = c$

grd6 : $rspmsg\_status(m0) = NOT\_OK$

grd9 : $rspmsg\_bal(m0) = bal(ac)$

then

act1 : $msg := msg \setminus \{m0\}$

act2 : $complete := complete \cup \{at\}$

act3 : $conversation := conversation \setminus \{at\}$

end

Event *withdrawATM* $\widehat{=}$

refines *withdrawATM*

any

ac

am

c

at

where

grd2 : $ac \in account$

grd3 : $am \in \mathbb{N}$

grd4 : $c \in cards$

grd5 : $at \in complete$

grd8 : $atm\_cash(at) \geq am$

grd9 : $atm\_card(at) = c$

grd10 : $card\_account(c) = ac$

then

act1 : $atm\_cash(at) := atm\_cash(at) - am$

act2 : complete := complete \ {at}

act3 : endWithdrawal := endWithdrawal ∪ {at}

end

Event *ejectCard* $\widehat{=}$

refines *ejectCard*

any

c

at

where

grd1 : c ∈ cards

grd2 : at ∈ endWithdrawal

grd5 : atm_card(at) = c

then

act2 : atm_card := atm_card \ {at ↦ c}

act3 : endWithdrawal := endWithdrawal \ {at}

act1 : idle := idle ∪ {at}

act4 : active_atm := active_atm \ {at}

end

END

Appendix D

ATM Case Study: Using Rodin UML-B

B.0 Generated Event-B Abstract Machine

MACHINE ATM_A

SEES ATM_A_implicitContext

VARIABLES

account // class instances

bal // attribute of account

INVARIANTS

account.type : account $\in \mathbb{P}$(Account)

bal.type : bal $\in$ account $\rightarrow \mathbb{N}$

EVENTS

Initialisation

begin

account.init : account $:= \varnothing$

bal.init : bal $:= \varnothing$

end

Event *checkBalance* $\widehat{=}$

any

self // contextual instance of class account

b

where

self.type : self ∈ account

b.type : b ∈ ℕ

checkBalance.Guard1 : b = bal(self)

then

skip

end

Event *deposit* ≙

any

self // contextual instance of class account

where

self.type : self ∈ account

then

skip

end

Event *createAccount* ≙

any

self // contextual instance of class account

where

self.type : self ∈ account

then

skip

end

Event *withdraw* ≙

any

self // contextual instance of class account

am

where

self.type : self ∈ account

am.type : am ∈ ℕ

withdraw.Guard1 : bal(self) ≥ am

then

withdraw.Action1 : bal(self) := bal(self) − am

end

END

B.1 Generated Event-B First Refinement

B.1.1 Context

CONTEXT ATM_CXR1

EXTENDS ATM_CXA

SETS

- `ATM` ClassType
- `Card` ClassType

CONSTANTS

- `ValidCard` classType instances
- `InvalidCard` classType instances
- `MIN_CASH` utility constant
- `MAX_CASH` utility constant
- `card_account` attribute of ValidCard

AXIOMS

- `ValidCard.type` : $ValidCard \in \mathbb{P}(Card)$
- `InvalidCard.type` : $InvalidCard \in \mathbb{P}(Card)$
- `MIN_CASH.type` : $MIN\_CASH \in \mathbb{N}$
- `MAX_CASH.type` : $MAX\_CASH \in \mathbb{N}$
- `card_account.type` : $card\_account \in ValidCard \rightarrowtail\!\!\!\!\!\!\rightarrow Account$
- `Axiom1` : $MAX\_CASH > MIN\_CASH$

END

B.1.1 Machine

MACHINE ATM_R1

REFINES ATM_A

SEES ATM_R1_implicitContext

VARIABLES

- `account` refined class instances
- `atm` class instances
- `bal` inherited attribute of account

`atm_acbal` attribute of atm

`atm_cash` attribute of atm

`atm_card` attribute of atm

`idle` state from statemachine, ATM_SM

`active_atm` state from statemachine, ATM_SM

INVARIANTS

`atm.type` : $atm \in \mathbb{P}(ATM)$

`atm_acbal.type` : $atm\_acbal \in atm \pfun \mathbb{N}$

`atm_cash.type` : $atm\_cash \in atm \fun \mathbb{N}$

`atm_card.type` : $atm\_card \in atm \pfun ValidCard$

`idle.type` : $idle \in \mathbb{P}(atm)$

`active_atm.type` : $active\_atm \in \mathbb{P}(atm)$

`ATM_SM_partitions_atm` : $partition(atm, idle, active\_atm)$

EVENTS

Initialisation

begin

`account.init` : $account := \varnothing$

`atm.init` : $atm := \varnothing$

`bal.init` : $bal := \varnothing$

`atm_acbal.init` : $atm\_acbal := \varnothing$

`atm_cash.init` : $atm\_cash := \varnothing$

`atm_card.init` : $atm\_card := \varnothing$

`idle.init` : $idle := \varnothing$

`active_atm.init` : $active\_atm := \varnothing$

end

Event $start \mathrel{\widehat{=}}$

any

$selfATM$ constructed instance of class atm

where

`selfATM.type` : $selfATM \in ATM \setminus atm$

then

`atm_constructor` : $atm := atm \cup \{selfATM\}$

`atm.atm_cash_initialise` : $atm\_cash(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *reloadCash* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

ATM_SM_isin_idle : $selfATM \in idle$

reloadCash.Guard1 : $atm\_cash(selfATM) < MIN\_CASH$

then

reloadCash.Action1 : $atm\_cash(selfATM) := MAX\_CASH - atm\_cash(selfATM)$

end

Event *insertCard* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_idle : $selfATM \in idle$

insertCard.Guard1 : $selfATM \notin dom(atm\_card)$

then

ATM_SM_leaveState_idle : $idle := idle \setminus \{selfATM\}$

ATM_SM_enterState_active_atm : $active\_atm := active\_atm \cup \{selfATM\}$

insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *withdrawOK* $\widehat{=}$

refines *withdraw*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_active_atm : $selfATM \in active\_atm$

withdrawOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawOK.Guard2 : $atm\_card(selfATM) = c$

withdrawOK.Guard3 : $bal(ac) \geq am$

withdrawOK.Guard4 : $card\_account(c) = ac$

withdrawOK.Guard5 : $atm\_cash(selfATM) \geq am$

with

self : ac = self

then

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbal(selfATM) := bal(ac)$

withdrawOK.Action3 : $atm\_cash(selfATM) := atm\_cash(selfATM) - am$

end

Event *withdrawFail* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_active_atm : $selfATM \in active\_atm$

withdrawFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawFail.Guard2 : $atm\_card(selfATM) = c$

withdrawFail.Guard3 : $card\_account(c) = ac$

withdrawFail.Guard4 : $bal(ac) < am$

then

withdrawFail.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

selfATM contextual instance of class atm

c

ac

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ac.type : $ac \in account$

ATM_SM_isin_active_atm : $selfATM \in active\_atm$

checkBalance.Guard1 : $selfATM \in dom(atm\_card)$

checkBalance.Guard2 : $atm\_card(selfATM) = c$

checkBalance.Guard3 : $card\_account(c) = ac$

with

self : ac = self

then

checkBalance.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *ejectCard* $\widehat{=}$

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_active_atm : $selfATM \in active\_atm$

ejectCard.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *createAccount* $\widehat{=}$

refines *createAccount*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

Event *deposit* $\widehat{=}$

refines *deposit*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

END

B.2 Generated Event-B Second Refinement

B.2.1 Context

CONTEXT ATM_CXR2

EXTENDS ATM_CXR1

SETS

`Pin` ClassType

CONSTANTS

`card_pin` attribute of ValidCard

AXIOMS

`card_pin.type` : $card\_pin \in ValidCard \twoheadrightarrow Pin$

END

B.2.2 Machine

MACHINE ATM_R2

REFINES ATM_R1

SEES ATM_R2_implicitContext

VARIABLES

`account` refined class instances

`atm` refined class instances

`bal` inherited attribute of account

`atm_card` inherited attribute of atm

`atm_acbal` inherited attribute of atm

`atm_cash` inherited attribute of atm

`idle` state from refined statemachine, ATM_SM

`active_atm` state from refined statemachine, ATM_SM

`validating` state from statemachine, active_atm_SM

`transOption` state from statemachine, active_atm_SM

`invalidCard` state from statemachine, active_atm_SM

`performTrans` state from statemachine, active_atm_SM

INVARIANTS

validating.type : $validating \in \mathbb{P}(active\_atm)$

transOption.type : $transOption \in \mathbb{P}(active\_atm)$

invalidCard.type : $invalidCard \in \mathbb{P}(active\_atm)$

performTrans.type : $performTrans \in \mathbb{P}(active\_atm)$

active_atm_SM_partitions_active_atm : $partition(active\_atm, validating, transOption, invalidCard, performTrans)$

EVENTS

Initialisation

begin

account.init : $account := \varnothing$

atm.init : $atm := \varnothing$

bal.init : $bal := \varnothing$

atm_card.init : $atm\_card := \varnothing$

atm_acbal.init : $atm\_acbal := \varnothing$

atm_cash.init : $atm\_cash := \varnothing$

idle.init : $idle := \varnothing$

active_atm.init : $active\_atm := \varnothing$

validating.init : $validating := \varnothing$

transOption.init : $transOption := \varnothing$

invalidCard.init : $invalidCard := \varnothing$

performTrans.init : $performTrans := \varnothing$

end

Event $start \mathrel{\widehat{=}}$

refines *start*

any

$selfATM$ constructed instance of class atm

where

selfATM.type : $selfATM \in ATM \setminus atm$

then

atm_constructor : $atm := atm \cup \{selfATM\}$

atm.atm_cash_initialise : $atm\_cash(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *reloadCash* $\widehat{=}$

refines *reloadCash*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

ATM_SM_isin_idle : $selfATM \in idle$

reloadCash.Guard1 : $atm\_cash(selfATM) < MIN\_CASH$

then

reloadCash.Action1 : $atm\_cash(selfATM) := MAX\_CASH - atm\_cash(selfATM)$

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_idle : $selfATM \in idle$

insertCard.Guard1 : $selfATM \notin dom(atm\_card)$

then

ATM_SM_enterSuperState_active_atm : $active\_atm := active\_atm \cup \{selfATM\}$

ATM_SM_leaveState_idle : $idle := idle \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *validateCardOK* $\widehat{=}$

any

selfATM contextual instance of class atm

c

p

where

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

validateCardOK.Guard2 : $atm\_card(selfATM) = c$

validateCardOK.Guard3 : $card\_pin(c) = p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

active_atm_SM_enterState_transOption : $transOption := transOption \cup \{selfATM\}$

end

Event *validateCardFail* $\widehat{=}$

any

selfATM contextual instance of class atm

c

p

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

p.type : $p \in Pin$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

validateCardFail.Guard2 : $atm\_card(selfATM) = c$

validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

active_atm_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *retry* $\widehat{=}$

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

retry.Guard1 : $selfATM \in dom(atm\_card)$

retry.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *ejectCard1* $\widehat{=}$

refines *ejectCard*

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *ejectCard2* $\widehat{=}$

refines *ejectCard*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_transOption : $selfATM \in transOption$

ejectCard2.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard2.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \backslash \{selfATM\}$

active_atm_SM_leaveState_transOption : $transOption := transOption \backslash \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard2.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *withdrawOK* $\widehat{=}$

refines *withdrawOK*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_transOption : $selfATM \in transOption$

withdrawOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawOK.Guard2 : $atm\_card(selfATM) = c$

withdrawOK.Guard3 : $bal(ac) \geq am$

withdrawOK.Guard4 : $card\_account(c) = ac$

withdrawOK.Guard5 : $atm\_cash(selfATM) \geq am$

then

active_atm_SM_leaveState_transOption : $transOption := transOption \backslash \{selfATM\}$

active_atm_SM_enterState_performTrans : $performTrans := performTrans \cup \{selfATM\}$

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbal(selfATM) := bal(ac)$

withdrawOK.Action3 : $atm\_cash(selfATM) := atm\_cash(selfATM) - am$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

selfATM contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_transOption : $selfATM \in transOption$

withdrawFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawFail.Guard2 : $atm\_card(selfATM) = c$

withdrawFail.Guard3 : $card\_account(c) = ac$

withdrawFail.Guard4 : $bal(ac) < am$

then

active_atm_SM_leaveState_transOption : $transOption := transOption \setminus \{selfATM\}$

active_atm_SM_enterState_performTrans : $performTrans := performTrans \cup \{selfATM\}$

withdrawFail.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

selfATM contextual instance of class atm

c

ac

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ac.type : $ac \in account$

active_atm_SM_isin_transOption : $selfATM \in transOption$

checkBalance.Guard1 : $selfATM \in dom(atm\_card)$

checkBalance.Guard2 : $atm\_card(selfATM) = c$

checkBalance.Guard3 : $card\_account(c) = ac$

then

active_atm_SM_leaveState_transOption : $transOption := transOption \setminus \{selfATM\}$

active_atm_SM_enterState_performTrans : $performTrans := performTrans \cup \{selfATM\}$

checkBalance.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *doAnother* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_performTrans : $selfATM \in performTrans$

doAnother.Guard1 : $selfATM \in dom(atm\_card)$

doAnother.Guard3 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveState_performTrans : $performTrans := performTrans \setminus \{selfATM\}$

active_atm_SM_enterState_transOption : $transOption := transOption \cup \{selfATM\}$

end

Event *ejectCard3* $\widehat{=}$

refines *ejectCard*

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_performTrans : $selfATM \in performTrans$

ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

active_atm_SM_leaveState_performTrans : $performTrans := performTrans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *createAccount* $\widehat{=}$

refines *createAccount*

any

self contextual instance of refined class account

where

self.type : $self \in account$

then

skip

end

Event *deposit* $\widehat{=}$

refines *deposit*

any

self contextual instance of refined class account

where

self.type : $self \in account$

then

skip

end

END

B.3 Generated Event-B Third Refinement

MACHINE ATM_R3

REFINES ATM_R2

SEES ATM_R3_implicitContext

VARIABLES

`account` refined class instances

`atm` refined class instances

`bal` inherited attribute of account

`atm_card` inherited attribute of atm

`atm_acbal` inherited attribute of atm

`atm_cashA` attribute of atm

`atm_wdam` attribute of atm

`atm_acbalA` attribute of atm

`idle` state from refined statemachine, ATM_SM

`active_atm` state from refined statemachine, ATM_SM

`validating` state from refined statemachine, active_atm_SM

`transOption` state from refined statemachine, active_atm_SM

`invalidCard` state from refined statemachine, active_atm_SM

`performTrans` state from refined statemachine, active_atm_SM

`trans` state from statemachine, transOption_SM

`reqWD` state from statemachine, transOption_SM

`reqCB` state from statemachine, transOption_SM

`processedWDFail` state from statemachine, performTrans_SM

`processedCB` state from statemachine, performTrans_SM

`processedWDOK` state from statemachine, performTrans_SM

`endTrans` state from statemachine, performTrans_SM

`rspWDOK` state from statemachine, performTrans_SM

`rspWDFail` state from statemachine, performTrans_SM

`rspCB` state from statemachine, performTrans_SM

INVARIANTS

`atm_cashA.type` : $atm\_cashA \in atm \rightarrow \mathbb{N}$

`atm_wdam.type` : $atm\_wdam \in atm \nrightarrow \mathbb{N}$

atm_acbalA.type : $atm\_acbalA \in atm \pfun \mathbb{N}$

trans.type : $trans \in \mathbb{P}(transOption)$

reqWD.type : $reqWD \in \mathbb{P}(transOption)$

reqCB.type : $reqCB \in \mathbb{P}(transOption)$

processedWDFail.type : $processedWDFail \in \mathbb{P}(performTrans)$

processedCB.type : $processedCB \in \mathbb{P}(performTrans)$

processedWDOK.type : $processedWDOK \in \mathbb{P}(performTrans)$

endTrans.type : $endTrans \in \mathbb{P}(performTrans)$

rspWDOK.type : $rspWDOK \in \mathbb{P}(performTrans)$

rspWDFail.type : $rspWDFail \in \mathbb{P}(performTrans)$

rspCB.type : $rspCB \in \mathbb{P}(performTrans)$

transOption_SM_partitions_transOption : $partition(transOption, trans, reqWD, reqCB)$

performTrans_SM_partitions_performTrans : $partition(performTrans, processedWDFail, processedCB, processedWDOK, endTrans, rspWDOK, rspWDFail, rspCB)$

Invariant1 : $\forall a \cdot a \in atm \land a \notin rspWDOK \land a \notin processedWDOK \Rightarrow atm\_cash(a) = atm\_cashA(a)$

Invariant2 : $\forall a \cdot a \in (rspWDOK \cup processedWDOK) \land a \in dom(atm\_wdam) \Rightarrow atm\_cash(a) + atm\_wdam(a) = atm\_cashA(a)$

Invariant3 : $\forall a \cdot a \in reqWD \land a \in dom(atm\_wdam) \Rightarrow atm\_cash(a) \geq atm\_wdam(a)$

EVENTS

Initialisation

begin

account.init : $account := \varnothing$

atm.init : $atm := \varnothing$

bal.init : $bal := \varnothing$

atm_card.init : $atm\_card := \varnothing$

atm_acbal.init : $atm\_acbal := \varnothing$

atm_cashA.init : $atm\_cashA := \varnothing$

atm_wdam.init : $atm\_wdam := \varnothing$

atm_acbalA.init : $atm\_acbalA := \varnothing$

idle.init : $idle := \varnothing$

active_atm.init : $active\_atm := \varnothing$

validating.init : $validating := \varnothing$

transOption.init : $transOption := \varnothing$

invalidCard.init : $invalidCard := \varnothing$

`performTrans.init` : $performTrans := \varnothing$

`trans.init` : $trans := \varnothing$

`reqWD.init` : $reqWD := \varnothing$

`reqCB.init` : $reqCB := \varnothing$

`processedWDFail.init` : $processedWDFail := \varnothing$

`processedCB.init` : $processedCB := \varnothing$

`processedWDOK.init` : $processedWDOK := \varnothing$

`endTrans.init` : $endTrans := \varnothing$

`rspWDOK.init` : $rspWDOK := \varnothing$

`rspWDFail.init` : $rspWDFail := \varnothing$

`rspCB.init` : $rspCB := \varnothing$

end

Event *createAccount* $\widehat{=}$

refines *createAccount*

any

self contextual instance of refined class account

where

`self.type` : $self \in account$

then

`skip`

end

Event *deposit* $\widehat{=}$

refines *deposit*

any

self contextual instance of refined class account

where

`self.type` : $self \in account$

then

`skip`

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_idle : $selfATM \in idle$

insertCard.Guard1 : $selfATM \notin dom(atm\_card)$

then

ATM_SM_enterSuperState_active_atm : $active\_atm := active\_atm \cup \{selfATM\}$

ATM_SM_leaveState_idle : $idle := idle \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *reloadCash* $\widehat{=}$

refines *reloadCash*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

ATM_SM_isin_idle : $selfATM \in idle$

reloadCash.Guard1 : $atm\_cashA(selfATM) < MIN\_CASH$

then

reloadCash.Action1 : $atm\_cashA(selfATM) := MAX\_CASH - atm\_cashA(selfATM)$

end

Event *ejectCard1* $\widehat{=}$

refines *ejectCard1*

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *withdrawOK* $\widehat{=}$

refines *withdrawOK*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

transOption_SM_isin_reqWD : $selfATM \in reqWD$

withdrawOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawOK.Guard2 : $atm\_card(selfATM) = c$

withdrawOK.Guard3 : $bal(ac) \geq am$

withdrawOK.Guard4 : $card\_account(c) = ac$

withdrawOK.Guard5 : $selfATM \in dom(atm\_wdam)$

withdrawOK.Guard6 : $atm\_wdam(selfATM) = am$

then

active_atm_SM_enterSuperState_performTrans : $performTrans := performTrans \cup \{selfATM\}$

active_atm_SM_leaveSuperState_transOption : $transOption := transOption \setminus \{selfATM\}$

transOption_SM_leaveState_reqWD : $reqWD := reqWD \setminus \{selfATM\}$

performTrans_SM_enterState_processedWDOK : $processedWDOK := processedWDOK \cup \{selfATM\}$

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbal(selfATM) := bal(ac) - am$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

selfATM contextual instance of class atm

c

am

ac

where

`am.type :` $am \in \mathbb{N}$

`ac.type :` $ac \in account$

`selfATM.type :` $selfATM \in atm$

`c.type :` $c \in ValidCard$

`transOption_SM_isin_reqWD :` $selfATM \in reqWD$

`withdrawFail.Guard1 :` $selfATM \in dom(atm\_card)$

`withdrawFail.Guard2 :` $atm\_card(selfATM) = c$

`withdrawFail.Guard3 :` $card\_account(c) = ac$

`withdrawFail.Guard4 :` $bal(ac) < am$

`withdrawFail.Guard5 :` $selfATM \in dom(atm\_wdam)$

`withdrawFail.Guard6 :` $atm\_wdam(selfATM) = am$

then

`active_atm_SM_enterSuperState_performTrans :` $performTrans := performTrans \cup \{selfATM\}$

`active_atm_SM_leaveSuperState_transOption :` $transOption := transOption \setminus \{selfATM\}$

`transOption_SM_leaveState_reqWD :` $reqWD := reqWD \setminus \{selfATM\}$

`performTrans_SM_enterState_processedWDFail :` $processedWDFail := processedWDFail \cup \{selfATM\}$

`withdrawFail.Action1 :` $atm\_acbal(selfATM) := bal(ac)$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

selfATM contextual instance of class atm

c

ac

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ac.type : $ac \in account$

transOption_SM_isin_reqCB : $selfATM \in reqCB$

checkBalance.Guard1 : $selfATM \in dom(atm\_card)$

checkBalance.Guard2 : $atm\_card(selfATM) = c$

checkBalance.Guard3 : $card\_account(c) = ac$

then

active_atm_SM_enterSuperState_performTrans : $performTrans := performTrans \cup \{selfATM\}$

active_atm_SM_leaveSuperState_transOption : $transOption := transOption \setminus \{selfATM\}$

transOption_SM_leaveState_reqCB : $reqCB := reqCB \setminus \{selfATM\}$

performTrans_SM_enterState_processedCB : $processedCB := processedCB \cup \{selfATM\}$

checkBalance.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *ejectCard2* $\widehat{=}$

refines *ejectCard2*

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

transOption_SM_isin_trans : $selfATM \in trans$

ejectCard2.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard2.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveSuperState_transOption : $transOption := transOption \setminus \{selfATM\}$

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard2.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *ejectCard3* $\widehat{=}$

refines *ejectCard3*

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveSuperState_performTrans : $performTrans := performTrans \setminus \{selfATM\}$

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *start* $\widehat{=}$

refines *start*

any

$selfATM$ constructed instance of class atm

where

selfATM.type : $selfATM \in ATM \setminus atm$

then

atm_constructor : $atm := atm \cup \{selfATM\}$

atm.atm_cashA_initialise : $atm\_cashA(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *validateCardFail* $\widehat{=}$

refines *validateCardFail*

any

selfATM contextual instance of class atm

c

p

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

p.type : $p \in Pin$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

validateCardFail.Guard2 : $atm\_card(selfATM) = c$

validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

active_atm_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *validateCardOK* $\widehat{=}$

refines *validateCardOK*

any

selfATM contextual instance of class atm

c

p

where

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

validateCardOK.Guard2 : $atm\_card(selfATM) = c$

validateCardOK.Guard3 : $card\_pin(c) = p$

then

active_atm_SM_enterSuperState_transOption : $transOption := transOption \cup \{selfATM\}$

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *retry* $\widehat{=}$

refines *retry*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

retry.Guard1 : $selfATM \in dom(atm\_card)$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *doAnother* $\widehat{=}$

refines *doAnother*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

doAnother.Guard1 : $selfATM \in dom(atm\_card)$

then

active_atm_SM_enterSuperState_transOption : $transOption := transOption \cup \{selfATM\}$

active_atm_SM_leaveSuperState_performTrans : $performTrans := performTrans \setminus \{selfATM\}$

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *requestWD* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

am

where

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

transOption_SM_isin_trans : $selfATM \in trans$

requestWD.Guard1 : $selfATM \in dom(atm\_card)$

requestWD.Guard3 : $atm\_cashA(selfATM) > MIN\_CASH$

requestWD.Guard5 : $am \leq MIN\_CASH$

then

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

transOption_SM_enterState_reqWD : $reqWD := reqWD \cup \{selfATM\}$

requestWD.Action1 : $atm\_wdam(selfATM) := am$

end

Event *requestCB* $\widehat{=}$

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

transOption_SM_isin_trans : $selfATM \in trans$

requestCB.Guard1 : $selfATM \in dom(atm\_card)$

then

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

transOption_SM_enterState_reqCB : $reqCB := reqCB \cup \{selfATM\}$

end

Event *responseWDFail* $\widehat{=}$

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_processedWDFail : $selfATM \in processedWDFail$

responseWDFail.Guard1 : $selfATM \in dom(atm\_card)$

responseWDFail.Guard2 : $selfATM \in dom(atm\_acbal)$

then

performTrans_SM_leaveState_processedWDFail : $processedWDFail := processedWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDFail : $rspWDFail := rspWDFail \cup \{selfATM\}$

responseWDFail.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *responseCB* $\widehat{=}$

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_processedCB : $selfATM \in processedCB$

responseCB.Guard1 : $selfATM \in dom(atm\_card)$

responseCB.Guard2 : $selfATM \in dom(atm\_acbal)$

then

performTrans_SM_leaveState_processedCB : $processedCB := processedCB \setminus \{selfATM\}$

performTrans_SM_enterState_rspCB : $rspCB := rspCB \cup \{selfATM\}$

responseCB.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *responseWDOK* $\widehat{=}$

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_processedWDOK : $selfATM \in processedWDOK$

responseWDOK.Guard1 : $selfATM \in dom(atm\_card)$

responseWDOK.Guard2 : $selfATM \in dom(atm\_acbal)$

then

performTrans_SM_leaveState_processedWDOK : $processedWDOK := processedWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDOK : $rspWDOK := rspWDOK \cup \{selfATM\}$

responseWDOK.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *withdrawATMOK* $\widehat{=}$

any

selfATM contextual instance of class atm

am

where

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDOK : $selfATM \in rspWDOK$

withdrawATMOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMOK.Guard2 : $selfATM \in dom(atm\_wdam)$

withdrawATMOK.Guard3 : $selfATM \in dom(atm\_acbalA)$

withdrawATMOK.Guard5 : $atm\_wdam(selfATM) = am$

withdrawATMOK.Guard8 : $atm\_cashA(selfATM) \geq am$

withdrawATMOK.Guard4 : $selfATM \in dom(atm\_acbal)$

withdrawATMOK.Guard6 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDOK : $rspWDOK := rspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

withdrawATMOK.Action1 : $atm\_cashA(selfATM) := atm\_cashA(selfATM) - am$

end

Event *withdrawATMFail* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDFail : $selfATM \in rspWDFail$

withdrawATMFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMFail.Guard2 : $selfATM \in dom(atm\_acbalA)$

withdrawATMFail.Guard3 : $selfATM \in dom(atm\_acbal)$

withdrawATMFail.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDFail : $rspWDFail := rspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *checkBalATM* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspCB : $selfATM \in rspCB$

checkBalATM.Guard1 : $selfATM \in dom(atm\_card)$

checkBalATM.Guard2 : $selfATM \in dom(atm\_acbalA)$

checkBalATM.Guard3 : $selfATM \in dom(atm\_acbal)$

checkBalATM.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspCB : $rspCB := rspCB \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

END

B.4 Generated Event-B Fourth Refinement

MACHINE ATM_R4

REFINES ATM_R3

SEES ATM_R4_implicitContext

VARIABLES

`account` refined class instances

`atm` refined class instances

`atmB` class instances

`bal` inherited attribute of account

`atm_card` inherited attribute of atm

`atm_cashA` inherited attribute of atm

`atm_wdam` inherited attribute of atm

`atm_acbalA` inherited attribute of atm

`atm_acbal` inherited attribute of atm

`idle` state from refined statemachine, ATM_SM

`active_atm` state from refined statemachine, ATM_SM

`validating` state from refined statemachine, active_atm_SM

`transOption` state from refined statemachine, active_atm_SM

`invalidCard` state from refined statemachine, active_atm_SM

`performTrans` state from refined statemachine, active_atm_SM

`trans` state from refined statemachine, transOption_SM

`reqWD` state from refined statemachine, transOption_SM

`reqCB` state from refined statemachine, transOption_SM

`sentReqWD` state from statemachine, reqWD_SM

`recvdReqWD` state from statemachine, reqWD_SM

`sentReqCB` state from statemachine, reqCB_SM

`recvdReqCB` state from statemachine, reqCB_SM

`processedWDFail` state from refined statemachine, performTrans_SM

`processedWDOK` state from refined statemachine, performTrans_SM

`processedCB` state from refined statemachine, performTrans_SM

`endTrans` state from refined statemachine, performTrans_SM

`rspWDOK` state from refined statemachine, performTrans_SM

`rspWDFail` state from refined statemachine, performTrans_SM

`rspCB` state from refined statemachine, performTrans_SM

`processWDFail` state from statemachine, processedWDFail_SM

`sentRspWDFail` state from statemachine, processedWDFail_SM

`processWDOK` state from statemachine, processedWDOK_SM

`sentRspWDOK` state from statemachine, processedWDOK_SM

`processCB` state from statemachine, processedCB_SM

`sentRspCB` state from statemachine, processedCB_SM

`atm_cardB` attribute of atmB

`atm_wdamB` attribute of atmB

INVARIANTS

`atmB.type` : $atmB \in \mathbb{P}(atm)$

`sentReqWD.type` : $sentReqWD \in \mathbb{P}(reqWD)$

`recvdReqWD.type` : $recvdReqWD \in \mathbb{P}(reqWD)$

`sentReqCB.type` : $sentReqCB \in \mathbb{P}(reqCB)$

`recvdReqCB.type` : $recvdReqCB \in \mathbb{P}(reqCB)$

`processWDFail.type` : $processWDFail \in \mathbb{P}(processedWDFail)$

`sentRspWDFail.type` : $sentRspWDFail \in \mathbb{P}(processedWDFail)$

`processWDOK.type` : $processWDOK \in \mathbb{P}(processedWDOK)$

`sentRspWDOK.type` : $sentRspWDOK \in \mathbb{P}(processedWDOK)$

`processCB.type` : $processCB \in \mathbb{P}(processedCB)$

`sentRspCB.type` : $sentRspCB \in \mathbb{P}(processedCB)$

`atm_cardB.type` : $atm\_cardB \in atmB \rightarrow ValidCard$

`atm_wdamB.type` : $atm\_wdamB \in atmB \nrightarrow \mathbb{N}$

`reqWD_SM_partitions_reqWD` : $partition(reqWD, sentReqWD, recvdReqWD)$

`reqCB_SM_partitions_reqCB` : $partition(reqCB, sentReqCB, recvdReqCB)$

`processedWDFail_SM_partitions_processedWDFail` : $partition(processedWDFail, processWDFail, sentRspWDFail)$

`processedWDOK_SM_partitions_processedWDOK` : $partition(processedWDOK, processWDOK, sentRspWDOK)$

`processedCB_SM_partitions_processedCB` : $partition(processedCB, processCB, sentRspCB)$

`Invariant1` : $\forall a \cdot a \in (recvdReqWD \cup recvdReqCB) \Rightarrow a \in dom(atm\_card)$

`Invariant2` : $\forall a \cdot a \in (recvdReqWD \cup recvdReqCB) \land a \in dom(atm\_card) \land a \in dom(atm\_cardB) \Rightarrow atm\_card(a) = atm\_cardB(a)$

Invariant3 : $\forall a \cdot a \in recvdReqWD \Rightarrow a \in dom(atm\_wdam)$

Invariant4 : $\forall a \cdot a \in recvdReqWD \land a \in dom(atm\_wdam) \land a \in dom(atm\_wdamB) \Rightarrow atm\_wdam(a) = atm\_wdamB(a)$

EVENTS

Initialisation

begin

account.init : $account := \varnothing$

atm.init : $atm := \varnothing$

atmB.init : $atmB := \varnothing$

bal.init : $bal := \varnothing$

atm_card.init : $atm\_card := \varnothing$

atm_cashA.init : $atm\_cashA := \varnothing$

atm_wdam.init : $atm\_wdam := \varnothing$

atm_acbalA.init : $atm\_acbalA := \varnothing$

atm_acbal.init : $atm\_acbal := \varnothing$

idle.init : $idle := \varnothing$

active_atm.init : $active\_atm := \varnothing$

validating.init : $validating := \varnothing$

transOption.init : $transOption := \varnothing$

invalidCard.init : $invalidCard := \varnothing$

performTrans.init : $performTrans := \varnothing$

trans.init : $trans := \varnothing$

reqWD.init : $reqWD := \varnothing$

reqCB.init : $reqCB := \varnothing$

sentReqWD.init : $sentReqWD := \varnothing$

recvdReqWD.init : $recvdReqWD := \varnothing$

sentReqCB.init : $sentReqCB := \varnothing$

recvdReqCB.init : $recvdReqCB := \varnothing$

processedWDFail.init : $processedWDFail := \varnothing$

processedWDOK.init : $processedWDOK := \varnothing$

processedCB.init : $processedCB := \varnothing$

endTrans.init : $endTrans := \varnothing$

rspWDOK.init : $rspWDOK := \varnothing$

rspWDFail.init : $rspWDFail := \varnothing$

rspCB.init : $rspCB := \varnothing$

processWDFail.init : $processWDFail := \varnothing$

`sentRspWDFail.init :` $sentRspWDFail := \varnothing$

`processWDOK.init :` $processWDOK := \varnothing$

`sentRspWDOK.init :` $sentRspWDOK := \varnothing$

`processCB.init :` $processCB := \varnothing$

`sentRspCB.init :` $sentRspCB := \varnothing$

`atm_cardB.init :` $atm\_cardB := \varnothing$

`atm_wdamB.init :` $atm\_wdamB := \varnothing$

end

Event *createAccount* $\widehat{=}$

refines *createAccount*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

Event *deposit* $\widehat{=}$

refines *deposit*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_idle : $selfATM \in idle$

insertCard.Guard1 : $selfATM \notin dom(atm\_card)$

then

ATM_SM_enterSuperState_active_atm : $active\_atm := active\_atm \cup \{selfATM\}$

ATM_SM_leaveState_idle : $idle := idle \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *reloadCash* $\widehat{=}$

refines *reloadCash*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

ATM_SM_isin_idle : $selfATM \in idle$

reloadCash.Guard1 : $atm\_cashA(selfATM) < MIN\_CASH$

then

reloadCash.Action1 : $atm\_cashA(selfATM) := MAX\_CASH$

end

Event *ejectCard1* $\widehat{=}$

refines *ejectCard1*

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *withdrawOK* $\widehat{=}$

refines *withdrawOK*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawOK.Guard7 : $selfATM \in atmB$

withdrawOK.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawOK.Guard2 : $atm\_cardB(selfATM) = c$

withdrawOK.Guard3 : $bal(ac) \geq am$

withdrawOK.Guard4 : $card\_account(c) = ac$

withdrawOK.Guard5 : $am = atm\_wdamB(selfATM)$

then

performTrans_SM_enterSuperState_processedWDOK : $processedWDOK := processedWDO$
$\{selfATM\}$

active_atm_SM_enterSuperState_performTrans : $performTrans := performTrans \cup$
$\{selfATM\}$

transOption_SM_leaveSuperState_reqWD : $reqWD := reqWD \setminus \{selfATM\}$

active_atm_SM_leaveSuperState_transOption : $transOption := transOption \setminus$
$\{selfATM\}$

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDOK_SM_enterState_processWDOK : $processWDOK := processWDOK \cup$
$\{selfATM\}$

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbal(selfATM) := bal(ac) - am$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawFail.Guard7 : $selfATM \in atmB$

withdrawFail.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawFail.Guard2 : $atm\_cardB(selfATM) = c$

withdrawFail.Guard3 : $card\_account(c) = ac$

withdrawFail.Guard4 : $bal(ac) < am$

withdrawFail.Guard5 : $am = atm\_wdamB(selfATM)$

then

performTrans_SM_enterSuperState_processedWDFail : $processedWDFail := processedWDFail \cup \{selfATM\}$

active_atm_SM_enterSuperState_performTrans : $performTrans := performTrans \cup \{selfATM\}$

transOption_SM_leaveSuperState_reqWD : $reqWD := reqWD \setminus \{selfATM\}$

active_atm_SM_leaveSuperState_transOption : $transOption := transOption \setminus \{selfATM\}$

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDFail_SM_enterState_processWDFail : $processWDFail := processWDFail \cup \{selfATM\}$

withdrawFail.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

selfATM contextual instance of class atm

c

ac

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ac.type : $ac \in account$

reqCB_SM_isin_recvdReqCB : $selfATM \in recvdReqCB$

checkBalance.Guard4 : $selfATM \in atmB$

checkBalance.Guard2 : $atm\_cardB(selfATM) = c$

checkBalance.Guard3 : $card\_account(c) = ac$

then

performTrans_SM_enterSuperState_processedCB : $processedCB := processedCB \cup \{selfATM\}$

active_atm_SM_enterSuperState_performTrans : $performTrans := performTrans \cup \{selfATM\}$

transOption_SM_leaveSuperState_reqCB : $reqCB := reqCB \setminus \{selfATM\}$

active_atm_SM_leaveSuperState_transOption : $transOption := transOption \setminus \{selfATM\}$

reqCB_SM_leaveState_recvdReqCB : $recvdReqCB := recvdReqCB \setminus \{selfATM\}$

processedCB_SM_enterState_processCB : $processCB := processCB \cup \{selfATM\}$

checkBalance.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *ejectCard2* $\widehat{=}$

refines *ejectCard2*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

transOption_SM_isin_trans : $selfATM \in trans$

ejectCard2.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard2.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveSuperState_transOption : $transOption := transOption \setminus \{selfATM\}$

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard2.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *ejectCard3* $\widehat{=}$

refines *ejectCard3*

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveSuperState_performTrans : $performTrans := performTrans \setminus \{selfATM\}$

ATM_SM_leaveSuperState_active_atm : $active\_atm := active\_atm \setminus \{selfATM\}$

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *start* $\widehat{=}$

refines *start*

any

$selfATM$ constructed instance of class atm

where

selfATM.type : $selfATM \in ATM \setminus atm$

then

atm_constructor : $atm := atm \cup \{selfATM\}$

atm.atm_cashA_initialise : $atm\_cashA(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *validateCardFail* $\widehat{=}$

refines *validateCardFail*

any

$selfATM$ contextual instance of class atm

c

p

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

p.type : $p \in Pin$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

validateCardFail.Guard2 : $atm\_card(selfATM) = c$

validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

active_atm_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *validateCardOK* $\widehat{=}$

refines *validateCardOK*

any

$selfATM$ contextual instance of class atm

c

p

where

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

validateCardOK.Guard2 : $atm\_card(selfATM) = c$

validateCardOK.Guard3 : $card\_pin(c) = p$

then

active_atm_SM_enterSuperState_transOption : $transOption := transOption \cup \{selfATM\}$

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *retry* $\widehat{=}$

refines *retry*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

retry.Guard1 : $selfATM \in dom(atm\_card)$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *doAnother* $\widehat{=}$

refines *doAnother*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

doAnother.Guard1 : $selfATM \in dom(atm\_card)$

then

active_atm_SM_enterSuperState_transOption : $transOption := transOption \cup \{selfATM\}$

active_atm_SM_leaveSuperState_performTrans : $performTrans := performTrans \setminus \{selfATM\}$

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *sendReqWD* $\widehat{=}$

refines *requestWD*

any

selfATM contextual instance of class atm

am

where

selfATM.type : $selfATM \in atm$

am.type : $am \in \mathbb{N}$

transOption_SM_isin_trans : $selfATM \in trans$

sendReqWD.Guard3 : $selfATM \in dom(atm\_card)$

sendReqWD.Guard1 : $atm\_cashA(selfATM) > MIN\_CASH$

sendReqWD.Guard5 : $am \leq MIN\_CASH$

then

transOption_SM_enterSuperState_reqWD : $reqWD := reqWD \cup \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

reqWD_SM_enterState_sentReqWD : $sentReqWD := sentReqWD \cup \{selfATM\}$

sendReqWD.Action1 : $atm\_wdam(selfATM) := am$

end

Event *sendReqCB* $\widehat{=}$

refines *requestCB*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

transOption_SM_isin_trans : $selfATM \in trans$

sendReqCB.Guard1 : $selfATM \in dom(atm\_card)$

then

transOption_SM_enterSuperState_reqCB : $reqCB := reqCB \cup \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

reqCB_SM_enterState_sentReqCB : $sentReqCB := sentReqCB \cup \{selfATM\}$

end

Event *recvReqWD* $\widehat{=}$

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqWD_SM_isin_sentReqWD : $selfATM \in sentReqWD$

recvReqWD.Guard1 : $selfATM \in dom(atm\_wdam)$

recvReqWD.Guard2 : $selfATM \in dom(atm\_card)$

then

reqWD_SM_leaveState_sentReqWD : $sentReqWD := sentReqWD \setminus \{selfATM\}$

reqWD_SM_enterState_recvdReqWD : $recvdReqWD := recvdReqWD \cup \{selfATM\}$

recvReqWD.Action1 : $atmB := atmB \cup \{selfATM\}$

recvReqWD.Action2 : $atm\_cardB(selfATM) := atm\_card(selfATM)$

recvReqWD.Action3 : $atm\_wdamB(selfATM) := atm\_wdam(selfATM)$

end

Event *recvReqCB* $\widehat{=}$

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqCB_SM_isin_sentReqCB : $selfATM \in sentReqCB$

recvReqCB.Guard1 : $selfATM \in dom(atm\_card)$

then

reqCB_SM_leaveState_sentReqCB : $sentReqCB := sentReqCB \setminus \{selfATM\}$

reqCB_SM_enterState_recvdReqCB : $recvdReqCB := recvdReqCB \cup \{selfATM\}$

recvReqCB.Action2 : $atm\_cardB(selfATM) := atm\_card(selfATM)$

recvReqCB.Action1 : $atmB := atmB \cup \{selfATM\}$

end

Event *recvRspWDFail* $\widehat{=}$

refines *responseWDFail*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_sentRspWDFail : $selfATM \in sentRspWDFail$

recvRspWDFail.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDFail.Guard2 : $selfATM \in dom(atm\_acbal)$

then

performTrans_SM_leaveSuperState_processedWDFail : $processedWDFail := processedWDFail \setminus \{selfATM\}$

processedWDFail_SM_leaveState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDFail : $rspWDFail := rspWDFail \cup \{selfATM\}$

recvRspWDFail.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event $recvRspWDOK \mathrel{\widehat{=}}$

refines $responseWDOK$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_sentRspWDOK : $selfATM \in sentRspWDOK$

recvRspWDOK.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDOK.Guard2 : $selfATM \in dom(atm\_acbal)$

then

performTrans_SM_leaveSuperState_processedWDOK : $processedWDOK := processedWDO \{selfATM\}$

processedWDOK_SM_leaveState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDOK : $rspWDOK := rspWDOK \cup \{selfATM\}$

recvRspWDOK.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event $recvRspCB \mathrel{\widehat{=}}$

refines $responseCB$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_sentRspCB : $selfATM \in sentRspCB$

recvRspCB.Guard1 : $selfATM \in dom(atm\_card)$

recvRspCB.Guard2 : $selfATM \in dom(atm\_acbal)$

then

performTrans_SM_leaveSuperState_processedCB : $processedCB := processedCB \setminus \{selfATM\}$

processedCB_SM_leaveState_sentRspCB : $sentRspCB := sentRspCB \setminus \{selfATM\}$

performTrans_SM_enterState_rspCB : $rspCB := rspCB \cup \{selfATM\}$

recvRspCB.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *withdrawATMOK* $\widehat{=}$

refines *withdrawATMOK*

any

$selfATM$ contextual instance of class atm

am

where

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDOK : $selfATM \in rspWDOK$

withdrawATMOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMOK.Guard2 : $selfATM \in dom(atm\_wdam)$

withdrawATMOK.Guard3 : $selfATM \in dom(atm\_acbalA)$

withdrawATMOK.Guard5 : $atm\_wdam(selfATM) = am$

withdrawATMOK.Guard7 : $atm\_cashA(selfATM) \geq am$

withdrawATMOK.Guard4 : $selfATM \in dom(atm\_acbal)$

withdrawATMOK.Guard6 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDOK : $rspWDOK := rspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

withdrawATMOK.Action2 : $atm\_cashA(selfATM) := atm\_cashA(selfATM) - am$

end

Event *withdrawATMFail* $\widehat{=}$

refines *withdrawATMFail*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDFail : $selfATM \in rspWDFail$

withdrawATMFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMFail.Guard2 : $selfATM \in dom(atm\_acbalA)$

withdrawATMFail.Guard3 : $selfATM \in dom(atm\_acbal)$

withdrawATMFail.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDFail : $rspWDFail := rspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *checkBalATM* $\widehat{=}$

refines *checkBalATM*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspCB : $selfATM \in rspCB$

checkBalATM.Guard1 : $selfATM \in dom(atm\_card)$

checkBalATM.Guard2 : $selfATM \in dom(atm\_acbalA)$

checkBalATM.Guard3 : $selfATM \in dom(atm\_acbal)$

checkBalATM.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspCB : $rspCB := rspCB \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *sendRspWDFail* $\widehat{=}$

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_processWDFail : $selfATM \in processWDFail$

sendRspWDFail.Guard1 : $selfATM \in dom(atm\_cardB)$

sendRspWDFail.Guard3 : $selfATM \in atmB$

sendRspWDFail.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDFail_SM_leaveState_processWDFail : $processWDFail := processWDFail \setminus \{selfATM\}$

processedWDFail_SM_enterState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \cup \{selfATM\}$

sendRspWDFail.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDFail.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDFail.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event $sendRspWDOK \mathrel{\widehat{=}}$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_processWDOK : $selfATM \in processWDOK$

sendRspWDOK.Guard3 : $selfATM \in atmB$

sendRspWDOK.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDOK_SM_leaveState_processWDOK : $processWDOK := processWDOK \setminus \{selfATM\}$

processedWDOK_SM_enterState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \cup \{selfATM\}$

sendRspWDOK.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDOK.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDOK.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event $sendRspCB \mathrel{\widehat{=}}$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_processCB : $selfATM \in processCB$

sendRspCB.Guard3 : $selfATM \in atmB$

sendRspCB.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedCB_SM_leaveState_processCB : $processCB := processCB \setminus \{selfATM\}$

processedCB_SM_enterState_sentRspCB : $sentRspCB := sentRspCB \cup \{selfATM\}$

sendRspCB.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspCB.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspCB.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

END

B.5 Generated Event-B Fifth Refinement

MACHINE ATM_R5

REFINES ATM_R4

SEES ATM_R5_implicitContext

VARIABLES

`account` refined class instances

`atm` refined class instances

`atmB` refined class instances

`bal` inherited attribute of account

`atm_card` inherited attribute of atm

`atm_cashA` inherited attribute of atm

`atm_wdam` inherited attribute of atm

`atm_acbalA` inherited attribute of atm

`atm_acbal` inherited attribute of atm

`idle` state from refined statemachine, ATM_SM

`validating` state from refined statemachine, ATM_SM

`trans` state from refined statemachine, ATM_SM

`sentReqWD` state from refined statemachine, ATM_SM

`recvdReqWD` state from refined statemachine, ATM_SM

`sentReqCB` state from refined statemachine, ATM_SM

`recvdReqCB` state from refined statemachine, ATM_SM

`invalidCard` state from refined statemachine, ATM_SM

`processWDFail` state from refined statemachine, ATM_SM

`sentRspWDFail` state from refined statemachine, ATM_SM

`rspWDFail` state from refined statemachine, ATM_SM

`processWDOK` state from refined statemachine, ATM_SM

`sentRspWDOK` state from refined statemachine, ATM_SM

`rspWDOK` state from refined statemachine, ATM_SM

`processCB` state from refined statemachine, ATM_SM

`sentRspCB` state from refined statemachine, ATM_SM

`rspCB` state from refined statemachine, ATM_SM

`endTrans` state from refined statemachine, ATM_SM

`atm_cardB` inherited attribute of atmB

`atm_wdamB` inherited attribute of atmB

INVARIANTS

`ATM_SM_partitions_atm` : $partition(atm, idle, validating, trans, sentReqWD, recvdReqWD, sentReqCB, recvdReqCB, invalidCard, processWDFail, sentRspWDFail, rspWDFail, processWDOK, sentRspWDOK, rspWDOK, processCB, sentRspCB, rspCB, endTrans)$

EVENTS

Initialisation

begin

`account.init` : $account := \varnothing$

`atm.init` : $atm := \varnothing$

`atmB.init` : $atmB := \varnothing$

`bal.init` : $bal := \varnothing$

`atm_card.init` : $atm\_card := \varnothing$

`atm_cashA.init` : $atm\_cashA := \varnothing$

`atm_wdam.init` : $atm\_wdam := \varnothing$

`atm_acbalA.init` : $atm\_acbalA := \varnothing$

`atm_acbal.init` : $atm\_acbal := \varnothing$

`idle.init` : $idle := \varnothing$

`validating.init` : $validating := \varnothing$

`trans.init` : $trans := \varnothing$

`sentReqWD.init` : $sentReqWD := \varnothing$

`recvdReqWD.init` : $recvdReqWD := \varnothing$

`sentReqCB.init` : $sentReqCB := \varnothing$

`recvdReqCB.init` : $recvdReqCB := \varnothing$

`invalidCard.init` : $invalidCard := \varnothing$

`processWDFail.init` : $processWDFail := \varnothing$

`sentRspWDFail.init` : $sentRspWDFail := \varnothing$

`rspWDFail.init` : $rspWDFail := \varnothing$

`processWDOK.init` : $processWDOK := \varnothing$

`sentRspWDOK.init` : $sentRspWDOK := \varnothing$

`rspWDOK.init` : $rspWDOK := \varnothing$

`processCB.init` : $processCB := \varnothing$

`sentRspCB.init` : $sentRspCB := \varnothing$

`rspCB.init` : $rspCB := \varnothing$

`endTrans.init` : $endTrans := \varnothing$

atm_cardB.init : $atm\_cardB := \varnothing$

atm_wdamB.init : $atm\_wdamB := \varnothing$

end

Event *createAccount* $\widehat{=}$

refines *createAccount*

any

self contextual instance of refined class account

where

self.type : $self \in account$

then

skip

end

Event *deposit* $\widehat{=}$

refines *deposit*

any

self contextual instance of refined class account

where

self.type : $self \in account$

then

skip

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_idle : $selfATM \in idle$

insertCard.Guard1 : $selfATM \notin dom(atm\_card)$

then

ATM_SM_leaveState_idle : $idle := idle \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *reloadCash* $\widehat{=}$

refines *reloadCash*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

ATM_SM_isin_idle : $selfATM \in idle$

reloadCash.Guard1 : $atm\_cashA(selfATM) < MIN\_CASH$

then

reloadCash.Action1 : $atm\_cashA(selfATM) := MAX\_CASH$

end

Event *validateCardFail* $\widehat{=}$

refines *validateCardFail*

any

selfATM contextual instance of class atm

c

p

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

p.type : $p \in Pin$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

validateCardFail.Guard2 : $atm\_card(selfATM) = c$

validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

active_atm_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *validateCardOK* $\widehat{=}$

refines *validateCardOK*

any

selfATM contextual instance of class atm

c

p

where

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

validateCardOK.Guard2 : $atm\_card(selfATM) = c$

validateCardOK.Guard3 : $card\_pin(c) = p$

then

active_atm_SM_leaveState_validating : $validating := validating \backslash \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *sendReqWD* $\widehat{=}$

refines *sendReqWD*

any

selfATM contextual instance of class atm

am

where

selfATM.type : $selfATM \in atm$

am.type : $am \in \mathbb{N}$

transOption_SM_isin_trans : $selfATM \in trans$

sendReqWD.Guard3 : $selfATM \in dom(atm\_card)$

sendReqWD.Guard1 : $atm\_cashA(selfATM) > MIN\_CASH$

sendReqWD.Guard5 : $am \leq MIN\_CASH$

then

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

reqWD_SM_enterState_sentReqWD : $sentReqWD := sentReqWD \cup \{selfATM\}$

sendReqWD.Action1 : $atm\_wdam(selfATM) := am$

end

Event *sendReqCB* $\widehat{=}$

refines *sendReqCB*

any

selfATM contextual instance of class atm

where

`selfATM.type :` $selfATM \in atm$

`transOption_SM_isin_trans :` $selfATM \in trans$

`sendReqCB.Guard1 :` $selfATM \in dom(atm\_card)$

then

`transOption_SM_leaveState_trans :` $trans := trans \setminus \{selfATM\}$

`reqCB_SM_enterState_sentReqCB :` $sentReqCB := sentReqCB \cup \{selfATM\}$

end

Event *ejectCard2* $\widehat{=}$

refines *ejectCard2*

any

selfATM contextual instance of class atm

c

where

`selfATM.type :` $selfATM \in atm$

`c.type :` $c \in ValidCard$

`transOption_SM_isin_trans :` $selfATM \in trans$

`ejectCard2.Guard1 :` $selfATM \in dom(atm\_card)$

`ejectCard2.Guard2 :` $atm\_card(selfATM) = c$

then

`transOption_SM_leaveState_trans :` $trans := trans \setminus \{selfATM\}$

`ATM_SM_enterState_idle :` $idle := idle \cup \{selfATM\}$

`ejectCard2.Action1 :` $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *recvReqWD* $\widehat{=}$

refines *recvReqWD*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqWD_SM_isin_sentReqWD : $selfATM \in sentReqWD$

recvReqWD.Guard1 : $selfATM \in dom(atm\_wdam)$

recvReqWD.Guard2 : $selfATM \in dom(atm\_card)$

then

reqWD_SM_leaveState_sentReqWD : $sentReqWD := sentReqWD \setminus \{selfATM\}$

reqWD_SM_enterState_recvdReqWD : $recvdReqWD := recvdReqWD \cup \{selfATM\}$

recvReqWD.Action1 : $atmB := atmB \cup \{selfATM\}$

recvReqWD.Action2 : $atm\_cardB(selfATM) := atm\_card(selfATM)$

recvReqWD.Action3 : $atm\_wdamB(selfATM) := atm\_wdam(selfATM)$

end

Event *withdrawOK* $\widehat{=}$

refines *withdrawOK*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawOK.Guard1 : $selfATM \in dom(atm\_cardB)$

withdrawOK.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawOK.Guard2 : $atm\_cardB(selfATM) = c$

withdrawOK.Guard3 : $bal(ac) \geq am$

withdrawOK.Guard4 : $card\_account(c) = ac$

withdrawOK.Guard5 : $am = atm\_wdamB(selfATM)$

then

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDOK_SM_enterState_processWDOK : $processWDOK := processWDOK \cup \{selfATM\}$

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbal(selfATM) := bal(ac) - am$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawFail.Guard1 : $selfATM \in dom(atm\_cardB)$

withdrawFail.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawFail.Guard2 : $atm\_cardB(selfATM) = c$

withdrawFail.Guard3 : $card\_account(c) = ac$

withdrawFail.Guard4 : $bal(ac) < am$

withdrawFail.Guard5 : $am = atm\_wdamB(selfATM)$

then

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDFail_SM_enterState_processWDFail : $processWDFail := processWDFail \cup \{selfATM\}$

withdrawFail.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *recvReqCB* $\widehat{=}$

refines *recvReqCB*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqCB_SM_isin_sentReqCB : $selfATM \in sentReqCB$

recvReqCB.Guard1 : $selfATM \in dom(atm\_card)$

then

reqCB_SM_leaveState_sentReqCB : $sentReqCB := sentReqCB \setminus \{selfATM\}$

reqCB_SM_enterState_recvdReqCB : $recvdReqCB := recvdReqCB \cup \{selfATM\}$

recvReqCB.Action1 : $atmB := atmB \cup \{selfATM\}$

recvReqCB.Action2 : $atm\_cardB(selfATM) := atm\_card(selfATM)$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

$selfATM$ contextual instance of class atm

c

ac

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ac.type : $ac \in account$

reqCB_SM_isin_recvdReqCB : $selfATM \in recvdReqCB$

checkBalance.Guard1 : $selfATM \in dom(atm\_cardB)$

checkBalance.Guard2 : $atm\_cardB(selfATM) = c$

checkBalance.Guard3 : $card\_account(c) = ac$

then

reqCB_SM_leaveState_recvdReqCB : $recvdReqCB := recvdReqCB \setminus \{selfATM\}$

processedCB_SM_enterState_processCB : $processCB := processCB \cup \{selfATM\}$

checkBalance.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *ejectCard1* $\widehat{=}$

refines *ejectCard1*

any

$selfATM$ contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *retry* $\widehat{=}$

refines *retry*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

retry.Guard1 : $selfATM \in dom(atm\_card)$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *sendRspWDFail* $\widehat{=}$

refines *sendRspWDFail*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_processWDFail : $selfATM \in processWDFail$

sendRspWDFail.Guard1 : $selfATM \in atmB$

sendRspWDFail.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDFail_SM_leaveState_processWDFail : $processWDFail := processWDFail \setminus \{selfATM\}$

processedWDFail_SM_enterState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \cup \{selfATM\}$

sendRspWDFail.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDFail.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDFail.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event *recvRspWDFail* $\widehat{=}$

refines *recvRspWDFail*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_sentRspWDFail : $selfATM \in sentRspWDFail$

recvRspWDFail.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDFail.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDFail_SM_leaveState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDFail : $rspWDFail := rspWDFail \cup \{selfATM\}$

recvRspWDFail.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *withdrawATMFail* $\widehat{=}$

refines *withdrawATMFail*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDFail : $selfATM \in rspWDFail$

withdrawATMFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMFail.Guard2 : $selfATM \in dom(atm\_acbalA)$

withdrawATMFail.Guard3 : $selfATM \in dom(atm\_acbal)$

withdrawATMFail.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDFail : $rspWDFail := rspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *sendRspWDOK* $\widehat{=}$

refines *sendRspWDOK*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_processWDOK : $selfATM \in processWDOK$

sendRspWDOK.Guard1 : $selfATM \in atmB$

sendRspWDOK.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDOK_SM_leaveState_processWDOK : $processWDOK := processWDOK \setminus \{selfATM\}$

processedWDOK_SM_enterState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \cup \{selfATM\}$

sendRspWDOK.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDOK.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDOK.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event *recvRspWDOK* $\widehat{=}$

refines *recvRspWDOK*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_sentRspWDOK : $selfATM \in sentRspWDOK$

recvRspWDOK.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDOK.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDOK_SM_leaveState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDOK : $rspWDOK := rspWDOK \cup \{selfATM\}$

recvRspWDOK.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *withdrawATMOK* $\widehat{=}$

refines *withdrawATMOK*

any

$selfATM$ contextual instance of class atm

am

where

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDOK : $selfATM \in rspWDOK$

withdrawATMOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMOK.Guard2 : $selfATM \in dom(atm\_wdam)$

withdrawATMOK.Guard3 : $selfATM \in dom(atm\_acbalA)$

withdrawATMOK.Guard5 : $atm\_wdam(selfATM) = am$

withdrawATMOK.Guard7 : $atm\_cashA(selfATM) \geq am$

withdrawATMOK.Guard4 : $selfATM \in dom(atm\_acbal)$

withdrawATMOK.Guard6 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDOK : $rspWDOK := rspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

withdrawATMOK.Action2 : $atm\_cashA(selfATM) := atm\_cashA(selfATM) - am$

end

Event $sendRspCB \mathrel{\widehat{=}}$

refines $sendRspCB$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_processCB : $selfATM \in processCB$

sendRspCB.Guard1 : $selfATM \in atmB$

sendRspCB.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedCB_SM_leaveState_processCB : $processCB := processCB \setminus \{selfATM\}$

processedCB_SM_enterState_sentRspCB : $sentRspCB := sentRspCB \cup \{selfATM\}$

sendRspCB.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspCB.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspCB.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event *recvRspCB* $\widehat{=}$

refines *recvRspCB*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_sentRspCB : $selfATM \in sentRspCB$

recvRspCB.Guard1 : $selfATM \in dom(atm\_card)$

recvRspCB.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedCB_SM_leaveState_sentRspCB : $sentRspCB := sentRspCB \setminus \{selfATM\}$

performTrans_SM_enterState_rspCB : $rspCB := rspCB \cup \{selfATM\}$

recvRspCB.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *checkBalATM* $\widehat{=}$

refines *checkBalATM*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspCB : $selfATM \in rspCB$

checkBalATM.Guard1 : $selfATM \in dom(atm\_card)$

checkBalATM.Guard2 : $selfATM \in dom(atm\_acbalA)$

checkBalATM.Guard3 : $selfATM \in dom(atm\_acbal)$

checkBalATM.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspCB : $rspCB := rspCB \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *ejectCard3* $\widehat{=}$

refines *ejectCard3*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *doAnother* $\widehat{=}$

refines *doAnother*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

doAnother.Guard1 : $selfATM \in dom(atm\_card)$

then

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *start* $\widehat{=}$

refines *start*

any

selfATM constructed instance of class atm

where

selfATM.type : $selfATM \in ATM \setminus atm$

then

atm_constructor : $atm := atm \cup \{selfATM\}$

atm.atm_cashA_initialise : $atm\_cashA(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

END

B.6 Generated Event-B Sixth Refinement

MACHINE ATM_R6

REFINES ATM_R5

SEES ATM_R6_implicitContext

VARIABLES

`account` refined class instances

`atm` refined class instances

`atmB` refined class instances

`bal` inherited attribute of account

`atm_card` inherited attribute of atm

`atm_cashA` inherited attribute of atm

`atm_wdam` inherited attribute of atm

`atm_acbalA` inherited attribute of atm

`atm_acbal` inherited attribute of atm

`idle` state from refined statemachine, ATM_SM

`validating` state from refined statemachine, ATM_SM

`trans` state from refined statemachine, ATM_SM

`invalidCard` state from refined statemachine, ATM_SM

`rspWDFail` state from refined statemachine, ATM_SM

`rspWDOK` state from refined statemachine, ATM_SM

`rspCB` state from refined statemachine, ATM_SM

`endTrans` state from refined statemachine, ATM_SM

`waitingResponse` state from statemachine, ATM_SM

`sentReqWD` state from refined statemachine, waitingResponseSM

`recvdReqWD` state from refined statemachine, waitingResponseSM

`sentReqCB` state from refined statemachine, waitingResponseSM

`recvdReqCB` state from refined statemachine, waitingResponseSM

`processWDFail` state from refined statemachine, waitingResponseSM

`sentRspWDFail` state from refined statemachine, waitingResponseSM

`processWDOK` state from refined statemachine, waitingResponseSM

`sentRspWDOK` state from refined statemachine, waitingResponseSM

`processCB` state from refined statemachine, waitingResponseSM

`sentRspCB` state from refined statemachine, waitingResponseSM

`atm_cardB` inherited attribute of atmB

`atm_wdamB` inherited attribute of atmB

INVARIANTS

`waitingResponse.type` : $waitingResponse \in \mathbb{P}(atm)$

`sentReqWD.type` : $sentReqWD \in \mathbb{P}(waitingResponse)$

`recvdReqWD.type` : $recvdReqWD \in \mathbb{P}(waitingResponse)$

`sentReqCB.type` : $sentReqCB \in \mathbb{P}(waitingResponse)$

`recvdReqCB.type` : $recvdReqCB \in \mathbb{P}(waitingResponse)$

`processWDFail.type` : $processWDFail \in \mathbb{P}(waitingResponse)$

`sentRspWDFail.type` : $sentRspWDFail \in \mathbb{P}(waitingResponse)$

`processWDOK.type` : $processWDOK \in \mathbb{P}(waitingResponse)$

`sentRspWDOK.type` : $sentRspWDOK \in \mathbb{P}(waitingResponse)$

`processCB.type` : $processCB \in \mathbb{P}(waitingResponse)$

`sentRspCB.type` : $sentRspCB \in \mathbb{P}(waitingResponse)$

`ATM_SM_partitions_atm` : $partition(atm, idle, validating, trans, invalidCard, rspWDFail, rspWDOK, rspCB, endTrans, waitingResponse)$

`waitingResponseSM_partitions_waitingResponse` : $partition(waitingResponse, sentReqWD, recvdReqWD, sentReqCB, recvdReqCB, processWDFail, sentRspWDFail, processWDOK, sentRspWDOK, processCB, sentRspCB)$

EVENTS

Initialisation

begin

`account.init` : $account := \varnothing$

`atm.init` : $atm := \varnothing$

`atmB.init` : $atmB := \varnothing$

`bal.init` : $bal := \varnothing$

`atm_card.init` : $atm\_card := \varnothing$

`atm_cashA.init` : $atm\_cashA := \varnothing$

`atm_wdam.init` : $atm\_wdam := \varnothing$

`atm_acbalA.init` : $atm\_acbalA := \varnothing$

`atm_acbal.init` : $atm\_acbal := \varnothing$

`idle.init` : $idle := \varnothing$

`validating.init` : $validating := \varnothing$

`trans.init :` $trans := \varnothing$

`invalidCard.init :` $invalidCard := \varnothing$

`rspWDFail.init :` $rspWDFail := \varnothing$

`rspWDOK.init :` $rspWDOK := \varnothing$

`rspCB.init :` $rspCB := \varnothing$

`endTrans.init :` $endTrans := \varnothing$

`waitingResponse.init :` $waitingResponse := \varnothing$

`sentReqWD.init :` $sentReqWD := \varnothing$

`recvdReqWD.init :` $recvdReqWD := \varnothing$

`sentReqCB.init :` $sentReqCB := \varnothing$

`recvdReqCB.init :` $recvdReqCB := \varnothing$

`processWDFail.init :` $processWDFail := \varnothing$

`sentRspWDFail.init :` $sentRspWDFail := \varnothing$

`processWDOK.init :` $processWDOK := \varnothing$

`sentRspWDOK.init :` $sentRspWDOK := \varnothing$

`processCB.init :` $processCB := \varnothing$

`sentRspCB.init :` $sentRspCB := \varnothing$

`atm_cardB.init :` $atm\_cardB := \varnothing$

`atm_wdamB.init :` $atm\_wdamB := \varnothing$

end

Event *createAccount* $\widehat{=}$

refines *createAccount*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

Event *deposit* $\widehat{=}$

refines *deposit*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

$selfATM$ contextual instance of class atm

c

where

`selfATM.type :` $selfATM \in atm$

`c.type :` $c \in ValidCard$

`ATM_SM_isin_idle :` $selfATM \in idle$

`insertCard.Guard1 :` $selfATM \notin dom(atm\_card)$

then

`ATM_SM_leaveState_idle :` $idle := idle \setminus \{selfATM\}$

`active_atm_SM_enterState_validating :` $validating := validating \cup \{selfATM\}$

`insertCard.Action1 :` $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *reloadCash* $\widehat{=}$

refines *reloadCash*

any

$selfATM$ contextual instance of class atm

where

`selfATM.type :` $selfATM \in atm$

`ATM_SM_isin_idle :` $selfATM \in idle$

`reloadCash.Guard1 :` $atm\_cashA(selfATM) < MIN\_CASH$

then

`reloadCash.Action1 :` $atm\_cashA(selfATM) := MAX\_CASH$

end

Event *validateCardFail* $\widehat{=}$

refines *validateCardFail*

any

selfATM contextual instance of class atm

c

p

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

p.type : $p \in Pin$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

validateCardFail.Guard2 : $atm\_card(selfATM) = c$

validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

active_atm_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *validateCardOK* $\widehat{=}$

refines *validateCardOK*

any

selfATM contextual instance of class atm

c

p

where

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

validateCardOK.Guard2 : $atm\_card(selfATM) = c$

validateCardOK.Guard3 : $card\_pin(c) = p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *ejectCard2* $\widehat{=}$

refines *ejectCard2*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

transOption_SM_isin_trans : $selfATM \in trans$

ejectCard2.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard2.Guard2 : $atm\_card(selfATM) = c$

then

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard2.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *sendReqWD* $\widehat{=}$

refines *sendReqWD*

any

selfATM contextual instance of class atm

am

where

selfATM.type : $selfATM \in atm$

am.type : $am \in \mathbb{N}$

transOption_SM_isin_trans : $selfATM \in trans$

sendReqWD.Guard3 : $selfATM \in dom(atm\_card)$

sendReqWD.Guard1 : $atm\_cashA(selfATM) > MIN\_CASH$

sendReqWD.Guard5 : $am \leq MIN\_CASH$

then

ATM_SM_enterSuperState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

reqWD_SM_enterState_sentReqWD : $sentReqWD := sentReqWD \cup \{selfATM\}$

sendReqWD.Action1 : $atm\_wdam(selfATM) := am$

end

Event *sendReqCB* $\widehat{=}$

refines *sendReqCB*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

transOption_SM_isin_trans : $selfATM \in trans$

sendReqCB.Guard1 : $selfATM \in dom(atm\_card)$

then

ATM_SM_enterSuperState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

reqCB_SM_enterState_sentReqCB : $sentReqCB := sentReqCB \cup \{selfATM\}$

end

Event *ejectCard1* $\widehat{=}$

refines *ejectCard1*

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *retry* $\widehat{=}$

refines *retry*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

retry.Guard1 : $selfATM \in dom(atm\_card)$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *withdrawATMFail* $\widehat{=}$

refines *withdrawATMFail*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDFail : $selfATM \in rspWDFail$

withdrawATMFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMFail.Guard2 : $selfATM \in dom(atm\_acbalA)$

withdrawATMFail.Guard3 : $selfATM \in dom(atm\_acbal)$

withdrawATMFail.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDFail : $rspWDFail := rspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *withdrawATMOK* $\widehat{=}$

refines *withdrawATMOK*

any

$selfATM$ contextual instance of class atm

am

where

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDOK : $selfATM \in rspWDOK$

withdrawATMOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMOK.Guard2 : $selfATM \in dom(atm\_wdam)$

withdrawATMOK.Guard3 : $selfATM \in dom(atm\_acbalA)$

withdrawATMOK.Guard5 : $atm\_wdam(selfATM) = am$

withdrawATMOK.Guard7 : $atm\_cashA(selfATM) \geq am$

withdrawATMOK.Guard4 : $selfATM \in dom(atm\_acbal)$

withdrawATMOK.Guard6 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspWDOK : $rspWDOK := rspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

withdrawATMOK.Action2 : $atm\_cashA(selfATM) := atm\_cashA(selfATM) - am$

end

Event *checkBalATM* $\widehat{=}$

refines *checkBalATM*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspCB : $selfATM \in rspCB$

checkBalATM.Guard1 : $selfATM \in dom(atm\_card)$

checkBalATM.Guard2 : $selfATM \in dom(atm\_acbalA)$

checkBalATM.Guard3 : $selfATM \in dom(atm\_acbal)$

checkBalATM.Guard4 : $atm\_acbalA(selfATM) = atm\_acbal(selfATM)$

then

performTrans_SM_leaveState_rspCB : $rspCB := rspCB \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *ejectCard3* $\widehat{=}$

refines *ejectCard3*

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *doAnother* $\widehat{=}$

refines *doAnother*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

doAnother.Guard1 : $selfATM \in dom(atm\_card)$

then

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *start* $\widehat{=}$

refines *start*

any

selfATM constructed instance of class atm

where

selfATM.type : $selfATM \in ATM \setminus atm$

then

atm_constructor : $atm := atm \cup \{selfATM\}$

atm.atm_cashA_initialise : $atm\_cashA(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *recvRspWDFail* $\widehat{=}$

refines *recvRspWDFail*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_sentRspWDFail : $selfATM \in sentRspWDFail$

recvRspWDFail.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDFail.Guard2 : $selfATM \in dom(atm\_acbal)$

then

ATM_SM_leaveSuperState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

processedWDFail_SM_leaveState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDFail : $rspWDFail := rspWDFail \cup \{selfATM\}$

recvRspWDFail.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *recvRspWDOK* $\widehat{=}$

refines *recvRspWDOK*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_sentRspWDOK : $selfATM \in sentRspWDOK$

recvRspWDOK.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDOK.Guard2 : $selfATM \in dom(atm\_acbal)$

then

ATM_SM_leaveSuperState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

processedWDOK_SM_leaveState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDOK : $rspWDOK := rspWDOK \cup \{selfATM\}$

recvRspWDOK.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *recvdRspCB* $\widehat{=}$

refines *recvRspCB*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_sentRspCB : $selfATM \in sentRspCB$

recvdRspCB.Guard1 : $selfATM \in dom(atm\_card)$

recvdRspCB.Guard2 : $selfATM \in dom(atm\_acbal)$

then

ATM_SM_leaveSuperState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

processedCB_SM_leaveState_sentRspCB : $sentRspCB := sentRspCB \setminus \{selfATM\}$

performTrans_SM_enterState_rspCB : $rspCB := rspCB \cup \{selfATM\}$

recvdRspCB.Action1 : $atm\_acbalA(selfATM) := atm\_acbal(selfATM)$

end

Event *recvReqWD* $\widehat{=}$

refines *recvReqWD*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqWD_SM_isin_sentReqWD : $selfATM \in sentReqWD$

recvReqWD.Guard1 : $selfATM \in dom(atm\_wdam)$

recvReqWD.Guard2 : $selfATM \in dom(atm\_card)$

then

reqWD_SM_leaveState_sentReqWD : $sentReqWD := sentReqWD \setminus \{selfATM\}$

reqWD_SM_enterState_recvdReqWD : $recvdReqWD := recvdReqWD \cup \{selfATM\}$

recvReqWD.Action1 : $atmB := atmB \cup \{selfATM\}$

recvReqWD.Action2 : $atm\_cardB(selfATM) := atm\_card(selfATM)$

recvReqWD.Action3 : $atm\_wdamB(selfATM) := atm\_wdam(selfATM)$

end

Event *withdrawOK* $\widehat{=}$

refines *withdrawOK*

any

selfATM contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawOK.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawOK.Guard2 : $atm\_cardB(selfATM) = c$

withdrawOK.Guard3 : $bal(ac) \geq am$

withdrawOK.Guard4 : $card\_account(c) = ac$

withdrawOK.Guard5 : $am = atm\_wdamB(selfATM)$

then

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDOK_SM_enterState_processWDOK : $processWDOK := processWDOK \cup \{selfATM\}$

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbal(selfATM) := bal(ac) - am$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

selfATM contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawFail.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawFail.Guard2 : $atm\_cardB(selfATM) = c$

withdrawFail.Guard3 : $card\_account(c) = ac$

withdrawFail.Guard4 : $bal(ac) < am$

withdrawFail.Guard5 : $am = atm\_wdamB(selfATM)$

then

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDFail_SM_enterState_processWDFail : $processWDFail := processWDFail \cup \{selfATM\}$

withdrawFail.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event *recvReqCB* $\widehat{=}$

refines *recvReqCB*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqCB_SM_isin_sentReqCB : $selfATM \in sentReqCB$

recvReqCB.Guard1 : $selfATM \in dom(atm\_card)$

then

reqCB_SM_leaveState_sentReqCB : $sentReqCB := sentReqCB \setminus \{selfATM\}$

reqCB_SM_enterState_recvdReqCB : $recvdReqCB := recvdReqCB \cup \{selfATM\}$

recvReqCB.Action1 : $atmB := atmB \cup \{selfATM\}$

recvReqCB.Action2 : $atm\_cardB(selfATM) := atm\_card(selfATM)$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

$selfATM$ contextual instance of class atm

c

ac

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ac.type : $ac \in account$

reqCB_SM_isin_recvdReqCB : $selfATM \in recvdReqCB$

checkBalance.Guard6 : $selfATM \in dom(atm\_cardB)$

checkBalance.Guard2 : $atm\_cardB(selfATM) = c$

checkBalance.Guard3 : $card\_account(c) = ac$

then

reqCB_SM_leaveState_recvdReqCB : $recvdReqCB := recvdReqCB \setminus \{selfATM\}$

processedCB_SM_enterState_processCB : $processCB := processCB \cup \{selfATM\}$

checkBalance.Action1 : $atm\_acbal(selfATM) := bal(ac)$

end

Event $sendRspWDFail \mathrel{\widehat{=}}$

refines $sendRspWDFail$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_processWDFail : $selfATM \in processWDFail$

sendRspWDFail.Guard1 : $selfATM \in atmB$

sendRspWDFail.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDFail_SM_leaveState_processWDFail : $processWDFail := processWDFail \setminus \{selfATM\}$

processedWDFail_SM_enterState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \cup \{selfATM\}$

sendRspWDFail.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDFail.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDFail.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event $sendRspWDOK \mathrel{\widehat{=}}$

refines $sendRspWDOK$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_processWDOK : $selfATM \in processWDOK$

sendRspWDOK.Guard1 : $selfATM \in atmB$

sendRspWDOK.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedWDOK_SM_leaveState_processWDOK : $processWDOK := processWDOK \setminus \{selfATM\}$

processedWDOK_SM_enterState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \cup \{selfATM\}$

sendRspWDOK.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDOK.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDOK.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event $sendRspCB \mathrel{\widehat{=}}$

refines $sendRspCB$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_processCB : $selfATM \in processCB$

sendRspCB.Guard1 : $selfATM \in atmB$

sendRspCB.Guard2 : $selfATM \in dom(atm\_acbal)$

then

processedCB_SM_leaveState_processCB : $processCB := processCB \setminus \{selfATM\}$

processedCB_SM_enterState_sentRspCB : $sentRspCB := sentRspCB \cup \{selfATM\}$

sendRspCB.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspCB.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspCB.Action2 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

END

B.7 Generated Event-B Seventh Refinement

MACHINE ATM_R7

REFINES ATM_R6

SEES ATM_R7_implicitContext

VARIABLES

`account` refined class instances

`atm` refined class instances

`atmB` refined class instances

`atmM` class instances

`bal` inherited attribute of account

`atm_card` inherited attribute of atm

`atm_cashA` inherited attribute of atm

`atm_wdam` inherited attribute of atm

`atm_acbalA` inherited attribute of atm

`idle` state from refined statemachine, ATM_SM

`validating` state from refined statemachine, ATM_SM

`trans` state from refined statemachine, ATM_SM

`invalidCard` state from refined statemachine, ATM_SM

`rspWDFail` state from refined statemachine, ATM_SM

`rspWDOK` state from refined statemachine, ATM_SM

`rspCB` state from refined statemachine, ATM_SM

`endTrans` state from refined statemachine, ATM_SM

`waitingResponse` state from refined statemachine, ATM_SM

`sentReqWD` state from refined statemachine, waitingResponseSM

`recvdReqWD` state from refined statemachine, waitingResponseSM

`sentReqCB` state from refined statemachine, waitingResponseSM

`recvdReqCB` state from refined statemachine, waitingResponseSM

`processWDFail` state from refined statemachine, waitingResponseSM

`sentRspWDFail` state from refined statemachine, waitingResponseSM

`processWDOK` state from refined statemachine, waitingResponseSM

`sentRspWDOK` state from refined statemachine, waitingResponseSM

`processCB` state from refined statemachine, waitingResponseSM

`sentRspCB` state from refined statemachine, waitingResponseSM

`atm_cardB` inherited attribute of atmB

`atm_acbalB` attribute of atmB

`atm_wdamB` inherited attribute of atmB

`atm_cardM` attribute of atmM

`atm_wdamM` attribute of atmM

`atm_acbalM` attribute of atmM

INVARIANTS

`atmM.type` : $atmM \in \mathbb{P}(atm)$

`atm_acbalB.type` : $atm\_acbalB \in atmB \pfun \mathbb{N}$

`atm_cardM.type` : $atm\_cardM \in atmM \fun ValidCard$

`atm_wdamM.type` : $atm\_wdamM \in atmM \pfun \mathbb{N}$

`atm_acbalM.type` : $atm\_acbalM \in atmM \pfun \mathbb{N}$

`Invariant3` : $\forall a \cdot a \in (processWDOK \cup processWDFail \cup processCB) \Rightarrow a \in dom(atm\_acbal)$

`Invariant9` : $\forall a \cdot a \in (sentRspWDFail \cup rspWDFail) \land a \in dom(atm\_acbal) \land a \in dom(atm\_acbalM) \Rightarrow atm\_acbal(a) = atm\_acbalM(a)$

`Invariant5` : $\forall a \cdot a \in processWDFail \land a \in dom(atm\_acbal) \land a \in dom(atm\_acbalB) \Rightarrow atm\_acbal(a) = atm\_acbalB(a)$

`Invariant7` : $\forall a \cdot a \in (sentRspWDOK \cup rspWDOK \cup sentRspWDFail \cup rspWDFail \cup sentRspCB \cup rspCB) \Rightarrow a \in dom(atm\_acbal)$

`Invariant8` : $\forall a \cdot a \in (sentRspWDOK \cup rspWDOK) \land a \in dom(atm\_acbal) \land a \in dom(atm\_acbalM) \Rightarrow atm\_acbal(a) = atm\_acbalM(a)$

`Invariant4` : $\forall a \cdot a \in processWDOK \land a \in dom(atm\_acbal) \land a \in dom(atm\_acbalB) \Rightarrow atm\_acbal(a) = atm\_acbalB(a)$

`Invariant10` : $\forall a \cdot a \in (sentRspCB \cup rspCB) \land a \in dom(atm\_acbal) \land a \in dom(atm\_acbalM) \Rightarrow atm\_acbal(a) = atm\_acbalM(a)$

`Invariant6` : $\forall a \cdot a \in processCB \land a \in dom(atm\_acbal) \land a \in dom(atm\_acbalB) \Rightarrow atm\_acbal(a) = atm\_acbalB(a)$

`Invariant1` : $\forall a \cdot a \in (rspWDFail \cup rspWDOK \cup rspCB) \land a \in dom(atm\_acbalA) \land a \in dom(atm\_acbal) \Rightarrow atm\_acbalA(a) = atm\_acbal(a)$

`Invariant2` : $\forall a \cdot a \in (sentReqWD \cup sentReqCB) \Rightarrow a \in dom(atm\_card)$

`Invariant11` : $\forall a \cdot a \in sentReqWD \Rightarrow a \in dom(atm\_wdam)$

`Invariant12` : $\forall a \cdot a \in (sentReqWD \cup sentReqCB) \land a \in dom(atm\_card) \land a \in dom(atm\_cardM) \Rightarrow atm\_card(a) = atm\_cardM(a)$

`Invariant13` : $\forall a \cdot a \in sentReqWD \land a \in dom(atm\_wdam) \land a \in dom(atm\_wdamM) \Rightarrow atm\_wdam(a) = atm\_wdamM(a)$

EVENTS

Initialisation

begin

account.init : $account := \varnothing$

atm.init : $atm := \varnothing$

atmB.init : $atmB := \varnothing$

atmM.init : $atmM := \varnothing$

bal.init : $bal := \varnothing$

atm_card.init : $atm\_card := \varnothing$

atm_cashA.init : $atm\_cashA := \varnothing$

atm_wdam.init : $atm\_wdam := \varnothing$

atm_acbalA.init : $atm\_acbalA := \varnothing$

idle.init : $idle := \varnothing$

validating.init : $validating := \varnothing$

trans.init : $trans := \varnothing$

invalidCard.init : $invalidCard := \varnothing$

rspWDFail.init : $rspWDFail := \varnothing$

rspWDOK.init : $rspWDOK := \varnothing$

rspCB.init : $rspCB := \varnothing$

endTrans.init : $endTrans := \varnothing$

waitingResponse.init : $waitingResponse := \varnothing$

sentReqWD.init : $sentReqWD := \varnothing$

recvdReqWD.init : $recvdReqWD := \varnothing$

sentReqCB.init : $sentReqCB := \varnothing$

recvdReqCB.init : $recvdReqCB := \varnothing$

processWDFail.init : $processWDFail := \varnothing$

sentRspWDFail.init : $sentRspWDFail := \varnothing$

processWDOK.init : $processWDOK := \varnothing$

sentRspWDOK.init : $sentRspWDOK := \varnothing$

processCB.init : $processCB := \varnothing$

sentRspCB.init : $sentRspCB := \varnothing$

atm_cardB.init : $atm\_cardB := \varnothing$

atm_acbalB.init : $atm\_acbalB := \varnothing$

atm_wdamB.init : $atm\_wdamB := \varnothing$

atm_cardM.init : $atm\_cardM := \varnothing$

atm_wdamM.init : $atm\_wdamM := \varnothing$

atm_acbalM.init : $atm\_acbalM := \varnothing$

end

Event *createAccount* $\widehat{=}$

refines *createAccount*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

Event *deposit* $\widehat{=}$

refines *deposit*

any

self contextual instance of refined class account

where

`self.type :` $self \in account$

then

`skip`

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

selfATM contextual instance of class atm

c

where

`selfATM.type :` $selfATM \in atm$

`c.type :` $c \in ValidCard$

`ATM_SM_isin_idle :` $selfATM \in idle$

`insertCard.Guard1 :` $selfATM \notin dom(atm\_card)$

then

`ATM_SM_leaveState_idle :` $idle := idle \setminus \{selfATM\}$

`active_atm_SM_enterState_validating :` $validating := validating \cup \{selfATM\}$

insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *reloadCash* $\widehat{=}$

refines *reloadCash*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

ATM_SM_isin_idle : $selfATM \in idle$

reloadCash.Guard1 : $atm\_cashA(selfATM) < MIN\_CASH$

then

reloadCash.Action1 : $atm\_cashA(selfATM) := MAX\_CASH$

end

Event *validateCardFail* $\widehat{=}$

refines *validateCardFail*

any

selfATM contextual instance of class atm

c

p

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

p.type : $p \in Pin$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

validateCardFail.Guard2 : $atm\_card(selfATM) = c$

validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

active_atm_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *validateCardOK* $\widehat{=}$

refines *validateCardOK*

any

selfATM contextual instance of class atm

c

p

where

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

active_atm_SM_isin_validating : $selfATM \in validating$

validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

validateCardOK.Guard2 : $atm\_card(selfATM) = c$

validateCardOK.Guard3 : $card\_pin(c) = p$

then

active_atm_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *ejectCard2* $\widehat{=}$

refines *ejectCard2*

any

selfATM contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

transOption_SM_isin_trans : $selfATM \in trans$

ejectCard2.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard2.Guard2 : $atm\_card(selfATM) = c$

then

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard2.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *sendReqWD* $\widehat{=}$

refines *sendReqWD*

any

selfATM contextual instance of class atm

am

where

selfATM.type : $selfATM \in atm$

am.type : $am \in \mathbb{N}$

transOption_SM_isin_trans : $selfATM \in trans$

sendReqWD.Guard3 : $selfATM \in dom(atm\_card)$

sendReqWD.Guard1 : $atm\_cashA(selfATM) > MIN\_CASH$

sendReqWD.Guard5 : $am \leq MIN\_CASH$

then

ATM_SM_enterSuperState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

reqWD_SM_enterState_sentReqWD : $sentReqWD := sentReqWD \cup \{selfATM\}$

sendReqWD.Action1 : $atm\_wdam(selfATM) := am$

sendReqWD.Action2 : $atm\_cardM(selfATM) := atm\_card(selfATM)$

sendReqWD.Action3 : $atmM := atmM \cup \{selfATM\}$

sendReqWD.Action4 : $atm\_wdamM(selfATM) := am$

end

Event *sendReqCB* $\widehat{=}$

refines *sendReqCB*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

transOption_SM_isin_trans : $selfATM \in trans$

sendReqCB.Guard1 : $selfATM \in dom(atm\_card)$

then

ATM_SM_enterSuperState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

transOption_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

reqCB_SM_enterState_sentReqCB : $sentReqCB := sentReqCB \cup \{selfATM\}$

sendReqCB.Action1 : $atm\_cardM(selfATM) := atm\_card(selfATM)$

sendReqCB.Action3 : $atmM := atmM \cup \{selfATM\}$

end

Event *ejectCard1* $\widehat{=}$

refines *ejectCard1*

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *retry* $\widehat{=}$

refines *retry*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

active_atm_SM_isin_invalidCard : $selfATM \in invalidCard$

retry.Guard1 : $selfATM \in dom(atm\_card)$

then

active_atm_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

active_atm_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *withdrawATMFail* $\widehat{=}$

refines *withdrawATMFail*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDFail : $selfATM \in rspWDFail$

withdrawATMFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMFail.Guard2 : $selfATM \in dom(atm\_acbalA)$

then

performTrans_SM_leaveState_rspWDFail : $rspWDFail := rspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event $withdrawATMOK \mathrel{\widehat{=}}$

refines *withdrawATMOK*

any

$selfATM$ contextual instance of class atm

am

where

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspWDOK : $selfATM \in rspWDOK$

withdrawATMOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMOK.Guard2 : $selfATM \in dom(atm\_wdam)$

withdrawATMOK.Guard3 : $selfATM \in dom(atm\_acbalA)$

withdrawATMOK.Guard5 : $atm\_wdam(selfATM) = am$

withdrawATMOK.Guard7 : $atm\_cashA(selfATM) \geq am$

then

performTrans_SM_leaveState_rspWDOK : $rspWDOK := rspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

withdrawATMOK.Action2 : $atm\_cashA(selfATM) := atm\_cashA(selfATM) - am$

end

Event $checkBalATM \mathrel{\widehat{=}}$

refines *checkBalATM*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_rspCB : $selfATM \in rspCB$

checkBalATM.Guard1 : $selfATM \in dom(atm\_card)$

checkBalATM.Guard2 : $selfATM \in dom(atm\_acbalA)$

then

performTrans_SM_leaveState_rspCB : $rspCB := rspCB \setminus \{selfATM\}$

performTrans_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *ejectCard3* $\widehat{=}$

refines *ejectCard3*

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *doAnother* $\widehat{=}$

refines *doAnother*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

performTrans_SM_isin_endTrans : $selfATM \in endTrans$

doAnother.Guard1 : $selfATM \in dom(atm\_card)$

then

performTrans_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

transOption_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *start* $\widehat{=}$

refines *start*

any

$selfATM$ constructed instance of class atm

where

selfATM.type : $selfATM \in ATM \setminus atm$

then

atm_constructor : $atm := atm \cup \{selfATM\}$

atm.atm_cashA_initialise : $atm\_cashA(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *recvRspWDFail* $\widehat{=}$

refines *recvRspWDFail*

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_sentRspWDFail : $selfATM \in sentRspWDFail$

recvRspWDFail.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDFail.Guard2 : $selfATM \in dom(atm\_acbalM)$

recvRspWDFail.Guard3 : $selfATM \in atmM$

then

ATM_SM_leaveSuperState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

processedWDFail_SM_leaveState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDFail : $rspWDFail := rspWDFail \cup \{selfATM\}$

recvRspWDFail.Action1 : $atm\_acbalA(selfATM) := atm\_acbalM(selfATM)$

recvRspWDFail.Action4 : $atmM := atmM \setminus \{selfATM\}$

recvRspWDFail.Action5 : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

recvRspWDFail.Action6 : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

recvRspWDFail.Action7 : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

end

Event *recvRspWDOK* $\widehat{=}$

refines *recvRspWDOK*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_sentRspWDOK : $selfATM \in sentRspWDOK$

recvRspWDOK.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDOK.Guard2 : $selfATM \in dom(atm\_acbalM)$

recvRspWDOK.Guard3 : $selfATM \in atmM$

then

ATM_SM_leaveSuperState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

processedWDOK_SM_leaveState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \setminus \{selfATM\}$

performTrans_SM_enterState_rspWDOK : $rspWDOK := rspWDOK \cup \{selfATM\}$

recvRspWDOK.Action1 : $atm\_acbalA(selfATM) := atm\_acbalM(selfATM)$

recvRspWDOK.Action4 : $atmM := atmM \setminus \{selfATM\}$

recvRspWDOK.Action5 : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

recvRspWDOK.Action6 : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

recvRspWDOK.Action7 : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

end

Event *recvdRspCB* $\widehat{=}$

refines *recvdRspCB*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_sentRspCB : $selfATM \in sentRspCB$

recvdRspCB.Guard1 : $selfATM \in dom(atm\_card)$

recvdRspCB.Guard2 : $selfATM \in dom(atm\_acbalM)$

recvdRspCB.Guard3 : $selfATM \in atmM$

then

ATM_SM_leaveSuperState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

processedCB_SM_leaveState_sentRspCB : $sentRspCB := sentRspCB \setminus \{selfATM\}$

performTrans_SM_enterState_rspCB : $rspCB := rspCB \cup \{selfATM\}$

recvdRspCB.Action1 : $atm\_acbalA(selfATM) := atm\_acbalM(selfATM)$

recvdRspCB.Action4 : $atmM := atmM \setminus \{selfATM\}$

recvdRspCB.Action5 : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

recvdRspCB.Action6 : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

recvdRspCB.Action7 : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

end

Event $recvReqWD \mathrel{\widehat{=}}$

refines $recvReqWD$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqWD_SM_isin_sentReqWD : $selfATM \in sentReqWD$

recvReqWD.Guard1 : $selfATM \in dom(atm\_wdamM)$

recvReqWD.Guard2 : $selfATM \in dom(atm\_cardM)$

then

reqWD_SM_leaveState_sentReqWD : $sentReqWD := sentReqWD \setminus \{selfATM\}$

reqWD_SM_enterState_recvdReqWD : $recvdReqWD := recvdReqWD \cup \{selfATM\}$

recvReqWD.Action1 : $atmB := atmB \cup \{selfATM\}$

recvReqWD.Action2 : $atm\_cardB(selfATM) := atm\_cardM(selfATM)$

recvReqWD.Action3 : $atm\_wdamB(selfATM) := atm\_wdamM(selfATM)$

recvReqWD.Action4 : $atmM := atmM \setminus \{selfATM\}$

recvReqWD.Action5 : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

recvReqWD.Action6 : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

recvReqWD.Action7 : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

end

Event $withdrawOK \mathrel{\widehat{=}}$

refines $withdrawOK$

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawOK.Guard1 : $selfATM \in atmB$

withdrawOK.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawOK.Guard2 : $atm\_cardB(selfATM) = c$

withdrawOK.Guard3 : $bal(ac) \geq am$

withdrawOK.Guard4 : $card\_account(c) = ac$

withdrawOK.Guard5 : $am = atm\_wdamB(selfATM)$

then

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDOK_SM_enterState_processWDOK : $processWDOK := processWDOK \cup \{selfATM\}$

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbalB(selfATM) := bal(ac) - am$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

$selfATM$ contextual instance of class atm

c

am

ac

where

am.type : $am \in \mathbb{N}$

ac.type : $ac \in account$

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

reqWD_SM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawFail.Guard1 : $selfATM \in atmB$

withdrawFail.Guard6 : $selfATM \in dom(atm\_wdamB)$

withdrawFail.Guard2 : $atm\_cardB(selfATM) = c$

withdrawFail.Guard3 : $card\_account(c) = ac$

withdrawFail.Guard4 : $bal(ac) < am$

withdrawFail.Guard5 : $am = atm\_wdamB(selfATM)$

then

reqWD_SM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

processedWDFail_SM_enterState_processWDFail : $processWDFail := processWDFail \cup \{selfATM\}$

withdrawFail.Action1 : $atm\_acbalB(selfATM) := bal(ac)$

end

Event *recvReqCB* $\widehat{=}$

refines *recvReqCB*

any

selfATM contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

reqCB_SM_isin_sentReqCB : $selfATM \in sentReqCB$

recvReqCB.Guard1 : $selfATM \in dom(atm\_cardM)$

then

reqCB_SM_leaveState_sentReqCB : $sentReqCB := sentReqCB \setminus \{selfATM\}$

reqCB_SM_enterState_recvdReqCB : $recvdReqCB := recvdReqCB \cup \{selfATM\}$

recvReqCB.Action1 : $atmB := atmB \cup \{selfATM\}$

recvReqCB.Action2 : $atm\_cardB(selfATM) := atm\_cardM(selfATM)$

recvReqCB.Action4 : $atmM := atmM \setminus \{selfATM\}$

recvReqCB.Action5 : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

recvReqCB.Action6 : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

recvReqCB.Action7 : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

selfATM contextual instance of class atm

c

ac

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ac.type : $ac \in account$

reqCB_SM_isin_recvdReqCB : $selfATM \in recvdReqCB$

checkBalance.Guard1 : $selfATM \in atmB$

checkBalance.Guard2 : $atm\_cardB(selfATM) = c$

checkBalance.Guard3 : $card\_account(c) = ac$

then

reqCB_SM_leaveState_recvdReqCB : $recvdReqCB := recvdReqCB \setminus \{selfATM\}$

processedCB_SM_enterState_processCB : $processCB := processCB \cup \{selfATM\}$

checkBalance.Action1 : $atm\_acbalB(selfATM) := bal(ac)$

end

Event $sendRspWDFail \mathrel{\widehat{=}}$

refines $sendRspWDFail$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDFail_SM_isin_processWDFail : $selfATM \in processWDFail$

sendRspWDFail.Guard1 : $selfATM \in dom(atm\_acbalB)$

sendRspWDFail.Guard2 : $selfATM \in atmB$

then

processedWDFail_SM_leaveState_processWDFail : $processWDFail := processWDFail \setminus \{selfATM\}$

processedWDFail_SM_enterState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \cup \{selfATM\}$

sendRspWDFail.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDFail.Action2 : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

sendRspWDFail.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDFail.Action4 : $atm\_acbalM(selfATM) := atm\_acbalB(selfATM)$

sendRspWDFail.Action5 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

sendRspWDFail.Action6 : $atmM := atmM \cup \{selfATM\}$

sendRspWDFail.Action7 : $atm\_cardM(selfATM) := atm\_cardB(selfATM)$

end

Event $sendRspWDOK \widehat{=}$

refines $sendRspWDOK$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedWDOK_SM_isin_processWDOK : $selfATM \in processWDOK$

sendRspWDOK.Guard1 : $selfATM \in dom(atm\_acbalB)$

sendRspWDOK.Guard2 : $selfATM \in atmB$

then

processedWDOK_SM_leaveState_processWDOK : $processWDOK := processWDOK \setminus \{selfATM\}$

processedWDOK_SM_enterState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \cup \{selfATM\}$

sendRspWDOK.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspWDOK.Action2 : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

sendRspWDOK.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspWDOK.Action4 : $atm\_acbalM(selfATM) := atm\_acbalB(selfATM)$

sendRspWDOK.Action5 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

sendRspWDOK.Action6 : $atmM := atmM \cup \{selfATM\}$

sendRspWDOK.Action7 : $atm\_cardM(selfATM) := atm\_cardB(selfATM)$

end

Event $sendRspCB \widehat{=}$

refines $sendRspCB$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

processedCB_SM_isin_processCB : $selfATM \in processCB$

sendRspCB.Guard2 : $selfATM \in dom(atm\_acbalB)$

sendRspCB.Guard1 : $selfATM \in atmB$

then

processedCB_SM_leaveState_processCB : $processCB := processCB \setminus \{selfATM\}$

processedCB_SM_enterState_sentRspCB : $sentRspCB := sentRspCB \cup \{selfATM\}$

sendRspCB.Action1 : $atmB := atmB \setminus \{selfATM\}$

sendRspCB.Action2 : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

sendRspCB.Action3 : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

sendRspCB.Action4 : $atm\_acbalM(selfATM) := atm\_acbalB(selfATM)$

sendRspCB.Action5 : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

sendRspCB.Action6 : $atmM := atmM \cup \{selfATM\}$

sendRspCB.Action7 : $atm\_cardM(selfATM) := atm\_cardB(selfATM)$

end

END

B.8 Generated Event-B Eight Refinement

MACHINE ATM_R8

REFINES ATM_R7

SEES mATM_implicitContext, mBank_implicitContext, mMW_implicitContext

VARIABLES

`atm` class instances

`atm_cashA` attribute of atm

`atm_wdam` attribute of atm

`atm_card` attribute of atm

`atm_acbalA` attribute of atm

`idle` state from statemachine, ATM_SM

`validating` state from statemachine, ATM_SM

`trans` state from statemachine, ATM_SM

`invalidCard` state from statemachine, ATM_SM

`waitingResponse` state from statemachine, ATM_SM

`rspCB` state from statemachine, ATM_SM

`rspWDOK` state from statemachine, ATM_SM

`rspWDFail` state from statemachine, ATM_SM

`endTrans` state from statemachine, ATM_SM

`account` class instances

`atmB` class instances

`bal` attribute of account

`atm_acbalB` attribute of atmB

`atm_cardB` attribute of atmB

`atm_wdamB` attribute of atmB

`recvdReqCB` state from statemachine, bankSM

`recvdReqWD` state from statemachine, bankSM

`processCB` state from statemachine, bankSM

`processWDOK` state from statemachine, bankSM

`processWDFail` state from statemachine, bankSM

`atmM` class instances

`atm_acbalM` attribute of atmM

`atm_wdamM` attribute of atmM

`atm_cardM` attribute of atmM

`sentReqWD` state from statemachine, reqWD_SM

`sentReqCB` state from statemachine, reqCB_SM

`sentRspWDOK` state from statemachine, rspWDOK_SM

`sentRspWDFail` state from statemachine, rspWDFail_SM

`sentRspCB` state from statemachine, rspCB_SM

EVENTS

Initialisation

begin

`mATM/atm.init` : $atm := \varnothing$

`mATM/atm_cashA.init` : $atm\_cashA := \varnothing$

`mATM/atm_wdam.init` : $atm\_wdam := \varnothing$

`mATM/atm_card.init` : $atm\_card := \varnothing$

`mATM/atm_acbalA.init` : $atm\_acbalA := \varnothing$

`mATM/idle.init` : $idle := \varnothing$

`mATM/validating.init` : $validating := \varnothing$

`mATM/trans.init` : $trans := \varnothing$

`mATM/invalidCard.init` : $invalidCard := \varnothing$

`mATM/waitingResponse.init` : $waitingResponse := \varnothing$

`mATM/rspCB.init` : $rspCB := \varnothing$

`mATM/rspWDOK.init` : $rspWDOK := \varnothing$

`mATM/rspWDFail.init` : $rspWDFail := \varnothing$

`mATM/endTrans.init` : $endTrans := \varnothing$

`mBank/account.init` : $account := \varnothing$

`mBank/atmB.init` : $atmB := \varnothing$

`mBank/bal.init` : $bal := \varnothing$

`mBank/atm_acbalB.init` : $atm\_acbalB := \varnothing$

`mBank/atm_cardB.init` : $atm\_cardB := \varnothing$

`mBank/atm_wdamB.init` : $atm\_wdamB := \varnothing$

`mBank/recvdReqCB.init` : $recvdReqCB := \varnothing$

`mBank/recvdReqWD.init` : $recvdReqWD := \varnothing$

`mBank/processCB.init` : $processCB := \varnothing$

`mBank/processWDOK.init` : $processWDOK := \varnothing$

`mBank/processWDFail.init` : $processWDFail := \varnothing$

`mMW/atmM.init` : $atmM := \varnothing$

mMW/atm_acbalM.init : $atm\_acbalM := \varnothing$

mMW/atm_wdamM.init : $atm\_wdamM := \varnothing$

mMW/atm_cardM.init : $atm\_cardM := \varnothing$

mMW/sentReqWD.init : $sentReqWD := \varnothing$

mMW/sentReqCB.init : $sentReqCB := \varnothing$

mMW/sentRspWDOK.init : $sentRspWDOK := \varnothing$

mMW/sentRspWDFail.init : $sentRspWDFail := \varnothing$

mMW/sentRspCB.init : $sentRspCB := \varnothing$

end

Event *start* $\widehat{=}$

refines *start*

any

$selfATM$ constructed instance of class atm

where

mATM/selfATM.type : $selfATM \in ATM \setminus atm$

then

mATM/atm_constructor : $atm := atm \cup \{selfATM\}$

mATM/atm.atm_cashA_initialise : $atm\_cashA(selfATM) := MAX\_CASH$

mATM/ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *reloadCash* $\widehat{=}$

refines *reloadCash*

any

$selfATM$ contextual instance of class atm

where

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_idle : $selfATM \in idle$

mATM/reloadCash.Guard1 : $atm\_cashA(selfATM) < MIN\_CASH$

then

mATM/reloadCash.Action1 : $atm\_cashA(selfATM) := MAX\_CASH$

end

Event *insertCard* $\widehat{=}$

refines *insertCard*

any

$selfATM$ contextual instance of class atm

c

where

mATM/selfATM.type : $selfATM \in atm$

mATM/c.type : $c \in ValidCard$

mATM/ATM_SM_isin_idle : $selfATM \in idle$

mATM/insertCard.Guard1 : $selfATM \notin dom(atm\_card)$

then

mATM/ATM_SM_leaveState_idle : $idle := idle \setminus \{selfATM\}$

mATM/ATM_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

mATM/insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event $validateCardOK \mathrel{\widehat{=}}$

refines $validateCardOK$

any

$selfATM$ contextual instance of class atm

c

p

where

mATM/c.type : $c \in ValidCard$

mATM/p.type : $p \in Pin$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_validating : $selfATM \in validating$

mATM/validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

mATM/validateCardOK.Guard2 : $atm\_card(selfATM) = c$

mATM/validateCardOK.Guard3 : $card\_pin(c) = p$

then

mATM/ATM_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

mATM/ATM_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event $validateCardFail \mathrel{\widehat{=}}$

refines $validateCardFail$

any

selfATM contextual instance of class atm

c

p

where

mATM/c.type : $c \in ValidCard$

mATM/p.type : $p \in Pin$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_validating : $selfATM \in validating$

mATM/validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

mATM/validateCardFail.Guard2 : $atm\_card(selfATM) = c$

mATM/validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

mATM/ATM_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

mATM/ATM_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *retry* $\widehat{=}$

refines *retry*

any

selfATM contextual instance of class atm

c

where

mATM/c.type : $c \in ValidCard$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_invalidCard : $selfATM \in invalidCard$

mATM/retry.Guard1 : $selfATM \in dom(atm\_card)$

mATM/retry.Guard2 : $atm\_card(selfATM) = c$

then

mATM/ATM_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

mATM/ATM_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *sendReqWD* $\widehat{=}$

refines *sendReqWD*

any

selfATM contextual instance of class atm

c

am

where

mATM/c.type : $c \in ValidCard$

mATM/am.type : $am \in \mathbb{N}$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_trans : $selfATM \in trans$

mATM/sendReqWD.Guard1 : $selfATM \in dom(atm\_card)$

mATM/sendReqWD.Guard2 : $atm\_card(selfATM) = c$

mATM/sendReqWD.Guard5 : $atm\_cashA(selfATM) > MIN\_CASH$

mATM/sendReqWD.Guard7 : $am \leq MIN\_CASH$

mMW/am.type : $am \in \mathbb{N}$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in ATM \setminus atmM$

then

mATM/ATM_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

mATM/ATM_SM_enterState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

mATM/sendReqWD.Action1 : $atm\_wdam(selfATM) := am$

mMW/atmM_constructor : $atmM := atmM \cup \{selfATM\}$

mMW/reqWD_SM_enterState_sentReqWD : $sentReqWD := sentReqWD \cup \{selfATM\}$

mMW/sendReqWD.Action2 : $atm\_wdamM(selfATM) := am$

mMW/sendReqWD.Action1 : $atm\_cardM(selfATM) := c$

end

Event $sendReqCB \mathrel{\widehat{=}}$

refines $sendReqCB$

any

$selfATM$ contextual instance of class atm

c

where

mATM/c.type : $c \in ValidCard$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_trans : $selfATM \in trans$

mATM/sendReqCB.Guard1 : $selfATM \in dom(atm\_card)$

mATM/sendReqCB.Guard2 : $atm\_card(selfATM) = c$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in ATM \setminus atmM$

then

mATM/ATM_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

mATM/ATM_SM_enterState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

mMW/atmM_constructor : $atmM := atmM \cup \{selfATM\}$

mMW/reqCB_SM_enterState_sentReqCB : $sentReqCB := sentReqCB \cup \{selfATM\}$

mMW/sendReqCB.Action1 : $atm\_cardM(selfATM) := c$

end

Event $recvReqWD \mathrel{\widehat{=}}$

refines $recvReqWD$

any

$selfATM$ contextual instance of class atmM

c

am

where

mMW/c.type : $c \in ValidCard$

mMW/am.type : $am \in \mathbb{N}$

mMW/selfATM.type : $selfATM \in atmM$

mMW/reqWD_SM_isin_sentReqWD : $selfATM \in sentReqWD$

mMW/recvReqWD.Guard1 : $atm\_cardM(selfATM) = c$

mMW/recvReqWD.Guard2 : $selfATM \in dom(atm\_wdamM)$

mMW/recvReqWD.Guard3 : $atm\_wdamM(selfATM) = am$

mBank/c.type : $c \in ValidCard$

mBank/am.type : $am \in \mathbb{N}$

mBank/selfATM.type : $selfATM \in ATM \setminus atmB$

then

mMW/reqWD_SM_leaveState_sentReqWD : $sentReqWD := sentReqWD \setminus \{selfATM\}$

mMW/atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

mMW/atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

mMW/atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

mMW/atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

mBank/atmB_constructor : $atmB := atmB \cup \{selfATM\}$

mBank/bankSM_enterState_recvdReqWD : $recvdReqWD := recvdReqWD \cup \{selfATM\}$

mBank/recvReqWD.Action2 : $atm\_cardB(selfATM) := c$

mBank/recvReqWD.Action3 : $atm\_wdamB(selfATM) := am$

end

Event *recvReqCB* $\widehat{=}$

refines *recvReqCB*

any

selfATM contextual instance of class atmM

c

where

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in atmM$

mMW/reqCB_SM_isin_sentReqCB : $selfATM \in sentReqCB$

mMW/recvReqCB.Guard1 : $atm\_cardM(selfATM) = c$

mBank/c.type : $c \in ValidCard$

mBank/selfATM.type : $selfATM \in ATM \setminus atmB$

then

mMW/reqCB_SM_leaveState_sentReqCB : $sentReqCB := sentReqCB \setminus \{selfATM\}$

mMW/atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

mMW/atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

mMW/atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

mMW/atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

mBank/atmB_constructor : $atmB := atmB \cup \{selfATM\}$

mBank/bankSM_enterState_recvdReqCB : $recvdReqCB := recvdReqCB \cup \{selfATM\}$

mBank/recvReqCB.Action2 : $atm\_cardB(selfATM) := c$

end

Event *withdrawOK* $\widehat{=}$

refines *withdrawOK*

any

selfATM contextual instance of class atmB

c

ac

am

where

mBank/c.type : $c \in ValidCard$

mBank/ac.type : $ac \in account$

mBank/am.type : $am \in \mathbb{N}$

mBank/selfATM.type : $selfATM \in atmB$

mBank/bankSM_isin_recvdReqWD : $selfATM \in recvdReqWD$

mBank/withdrawOK.Guard1 : $selfATM \in dom(atm\_wdamB)$

mBank/withdrawOK.Guard4 : $bal(ac) \geq am$

mBank/withdrawOK.Guard2 : $card\_account(c) = ac$

mBank/withdrawOK.Guard7 : $atm\_cardB(selfATM) = c$

mBank/withdrawOK.Guard5 : $atm\_wdamB(selfATM) = am$

then

mBank/bankSM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

mBank/bankSM_enterState_processWDOK : $processWDOK := processWDOK \cup \{selfATM\}$

mBank/withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

mBank/withdrawOK.Action2 : $atm\_acbalB(selfATM) := bal(ac) - am$

end

Event *withdrawFail* $\widehat{=}$

refines *withdrawFail*

any

$selfATM$ contextual instance of class atmB

c

ac

am

where

mBank/c.type : $c \in ValidCard$

mBank/ac.type : $ac \in account$

mBank/am.type : $am \in \mathbb{N}$

mBank/selfATM.type : $selfATM \in atmB$

mBank/bankSM_isin_recvdReqWD : $selfATM \in recvdReqWD$

mBank/withdrawFail.Guard1 : $selfATM \in dom(atm\_wdamB)$

mBank/withdrawFail.Guard5 : $bal(ac) < am$

mBank/withdrawFail.Guard2 : $card\_account(c) = ac$

mBank/withdrawFail.Guard7 : $atm\_cardB(selfATM) = c$

mBank/withdrawFail.Guard4 : $atm\_wdamB(selfATM) = am$

then

mBank/bankSM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

mBank/bankSM_enterState_processWDFail : $processWDFail := processWDFail \cup \{selfATM\}$

mBank/withdrawFail.Action2 : $atm\_acbalB(selfATM) := bal(ac)$

end

Event *checkBalance* $\widehat{=}$

refines *checkBalance*

any

selfATM contextual instance of class atmB

c

ac

b

where

mBank/c.type : $c \in ValidCard$

mBank/ac.type : $ac \in account$

mBank/b.type : $b \in \mathbb{N}$

mBank/selfATM.type : $selfATM \in atmB$

mBank/bankSM_isin_recvdReqCB : $selfATM \in recvdReqCB$

mBank/checkBalance.Guard5 : $b = bal(ac)$

mBank/checkBalance.Guard2 : $card\_account(c) = ac$

mBank/checkBalance.Guard6 : $atm\_cardB(selfATM) = c$

then

mBank/bankSM_leaveState_recvdReqCB : $recvdReqCB := recvdReqCB \setminus \{selfATM\}$

mBank/bankSM_enterState_processCB : $processCB := processCB \cup \{selfATM\}$

mBank/checkBalance.Action2 : $atm\_acbalB(selfATM) := bal(ac)$

end

Event *sendRspWDOK* $\widehat{=}$

refines *sendRspWDOK*

any

selfATM contextual instance of class atmB

c

b

where

mBank/b.type : $b \in \mathbb{N}$

mBank/selfATM.type : $selfATM \in atmB$

mBank/c.type : $c \in ValidCard$

mBank/bankSM_isin_processWDOK : $selfATM \in processWDOK$

mBank/sendRspWDOK.Guard5 : $selfATM \in dom(atm\_acbalB)$

mBank/sendRspWDOK.Guard3 : $atm\_acbalB(selfATM) = b$

mBank/sendRspWDOK.Guard1 : $atm\_cardB(selfATM) = c$

mMW/b.type : $b \in \mathbb{N}$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in ATM \setminus atmM$

then

mBank/bankSM_leaveState_processWDOK : $processWDOK := processWDOK \setminus \{selfATM\}$

mBank/atmB_destructor : $atmB := atmB \setminus \{selfATM\}$

mBank/atmB.atm_acbalB_destructor : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

mBank/atmB.atm_cardB_destructor : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

mBank/atmB.atm_wdamB_destructor : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

mMW/atmM_constructor : $atmM := atmM \cup \{selfATM\}$

mMW/rspWDOK_SM_enterState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \cup \{selfATM\}$

mMW/sendRspWDOK.Action1 : $atm\_acbalM(selfATM) := b$

mMW/sendRspWDOK.Action2 : $atm\_cardM(selfATM) := c$

end

Event $sendRspWDFail \mathrel{\widehat{=}}$

refines $sendRspWDFail$

any

$selfATM$ contextual instance of class atmB

c

b

where

mBank/b.type : $b \in \mathbb{N}$

mBank/selfATM.type : $selfATM \in atmB$

mBank/c.type : $c \in ValidCard$

mBank/bankSM_isin_processWDFail : $selfATM \in processWDFail$

mBank/sendRspWDFail.Guard5 : $selfATM \in dom(atm\_acbalB)$

mBank/sendRspWDFail.Guard3 : $atm\_acbalB(selfATM) = b$

mBank/sendRspWDFail.Guard1 : $atm\_cardB(selfATM) = c$

mMW/b.type : $b \in \mathbb{N}$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in ATM \setminus atmM$

then

mBank/bankSM_leaveState_processWDFail : $processWDFail := processWDFail \setminus \{selfATM\}$

mBank/atmB_destructor : $atmB := atmB \setminus \{selfATM\}$

mBank/atmB.atm_acbalB_destructor : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

mBank/atmB.atm_cardB_destructor : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

mBank/atmB.atm_wdamB_destructor : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

mMW/atmM_constructor : $atmM := atmM \cup \{selfATM\}$

mMW/rspWDFail_SM_enterState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \cup \{selfATM\}$

mMW/sendRspWDFail.Action1 : $atm\_acbalM(selfATM) := b$

mMW/sendRspWDFail.Action2 : $atm\_cardM(selfATM) := c$

end

Event $sendRspCB \mathrel{\widehat{=}}$

refines $sendRspCB$

any

$selfATM$ contextual instance of class atmB

c

b

where

mBank/b.type : $b \in \mathbb{N}$

mBank/selfATM.type : $selfATM \in atmB$

mBank/c.type : $c \in ValidCard$

mBank/bankSM_isin_processCB : $selfATM \in processCB$

mBank/sendRspCB.Guard4 : $selfATM \in dom(atm\_acbalB)$

mBank/sendRspCB.Guard7 : $atm\_acbalB(selfATM) = b$

mBank/sendRspCB.Guard1 : $atm\_cardB(selfATM) = c$

mMW/b.type : $b \in \mathbb{N}$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in ATM \setminus atmM$

then

mBank/bankSM_leaveState_processCB : $processCB := processCB \setminus \{selfATM\}$

mBank/atmB_destructor : $atmB := atmB \setminus \{selfATM\}$

mBank/atmB.atm_acbalB_destructor : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

mBank/atmB.atm_cardB_destructor : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

mBank/atmB.atm_wdamB_destructor : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

mMW/atmM_constructor : $atmM := atmM \cup \{selfATM\}$

mMW/rspCB_SM_enterState_sentRspCB : $sentRspCB := sentRspCB \cup \{selfATM\}$

mMW/sendRspCB.Action1 : $atm\_acbalM(selfATM) := b$

mMW/sendRspCB.Action2 : $atm\_cardM(selfATM) := c$

end

Event $recvRspWDOK \mathrel{\widehat{=}}$

refines $recvRspWDOK$

any

$selfATM$ contextual instance of class atmM

b

c

where

mMW/b.type : $b \in \mathbb{N}$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in atmM$

mMW/rspWDOK_SM_isin_sentRspWDOK : $selfATM \in sentRspWDOK$

mMW/recvRspWDOK.Guard1 : $selfATM \in dom(atm\_acbalM)$

mMW/recvRspWDOK.Guard2 : $atm\_acbalM(selfATM) = b$

mMW/recvRspWDOK.Guard3 : $atm\_cardM(selfATM) = c$

mATM/c.type : $c \in ValidCard$

mATM/b.type : $b \in \mathbb{N}$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_waitingResponse : $selfATM \in waitingResponse$

mATM/recvRspWDOK.Guard1 : $selfATM \in dom(atm\_card)$

mATM/recvRspWDOK.Guard2 : $atm\_card(selfATM) = c$

then

mMW/rspWDOK_SM_leaveState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \setminus \{selfATM\}$

mMW/atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

mMW/atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

mMW/atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

mMW/atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

mATM/ATM_SM_leaveState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

mATM/ATM_SM_enterState_rspWDOK : $rspWDOK := rspWDOK \cup \{selfATM\}$

mATM/recvRspWDOK.Action1 : $atm\_acbalA(selfATM) := b$

end

Event *recvRspWDFail* $\widehat{=}$

refines *recvRspWDFail*

any

selfATM contextual instance of class atmM

b

c

where

mMW/b.type : $b \in \mathbb{N}$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in atmM$

mMW/rspWDFail_SM_isin_sentRspWDFail : $selfATM \in sentRspWDFail$

mMW/recvRspWDFail.Guard1 : $selfATM \in dom(atm\_acbalM)$

mMW/recvRspWDFail.Guard2 : $atm\_acbalM(selfATM) = b$

mMW/recvRspWDFail.Guard3 : $atm\_cardM(selfATM) = c$

mATM/c.type : $c \in ValidCard$

mATM/b.type : $b \in \mathbb{N}$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_waitingResponse : $selfATM \in waitingResponse$

mATM/recvRspWDFail.Guard1 : $selfATM \in dom(atm\_card)$

mATM/recvRspWDFail.Guard2 : $atm\_card(selfATM) = c$

then

mMW/rspWDFail_SM_leaveState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \setminus \{selfATM\}$

mMW/atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

mMW/atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

mMW/atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

mMW/atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

mATM/ATM_SM_leaveState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

mATM/ATM_SM_enterState_rspWDFail : $rspWDFail := rspWDFail \cup \{selfATM\}$

mATM/recvRspWDFail.Action1 : $atm\_acbalA(selfATM) := b$

end

Event *recvdRspCB* $\widehat{=}$

refines *recvdRspCB*

any

selfATM contextual instance of class atmM

b

c

where

mMW/b.type : $b \in \mathbb{N}$

mMW/c.type : $c \in ValidCard$

mMW/selfATM.type : $selfATM \in atmM$

mMW/rspCB_SM_isin_sentRspCB : $selfATM \in sentRspCB$

mMW/recvRspCB.Guard1 : $selfATM \in dom(atm\_acbalM)$

mMW/recvRspCB.Guard2 : $atm\_acbalM(selfATM) = b$

mMW/recvRspCB.Guard3 : $atm\_cardM(selfATM) = c$

mATM/c.type : $c \in ValidCard$

mATM/b.type : $b \in \mathbb{N}$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_waitingResponse : $selfATM \in waitingResponse$

mATM/recvRspCB.Guard1 : $selfATM \in dom(atm\_card)$

mATM/recvRspCB.Guard2 : $atm\_card(selfATM) = c$

then

mMW/rspCB_SM_leaveState_sentRspCB : $sentRspCB := sentRspCB \setminus \{selfATM\}$

mMW/atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

mMW/atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

mMW/atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

mMW/atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

mATM/ATM_SM_leaveState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

mATM/ATM_SM_enterState_rspCB : $rspCB := rspCB \cup \{selfATM\}$

mATM/recvRspCB.Action1 : $atm\_acbalA(selfATM) := b$

end

Event *withdrawATMOK* $\widehat{=}$

refines *withdrawATMOK*

any

selfATM contextual instance of class atm

c

am

b

where

mATM/c.type : $c \in ValidCard$

mATM/am.type : $am \in \mathbb{N}$

mATM/b.type : $b \in \mathbb{N}$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_rspWDOK : $selfATM \in rspWDOK$

mATM/withdrawATMOK.Guard1 : $selfATM \in dom(atm\_card)$

mATM/withdrawATMOK.Guard2 : $selfATM \in dom(atm\_acbalA)$

mATM/withdrawATMOK.Guard3 : $atm\_cashA(selfATM) \geq am$

mATM/withdrawATMOK.Guard4 : $atm\_acbalA(selfATM) = b$

mATM/withdrawATMOK.Guard5 : $selfATM \in dom(atm\_wdam)$

mATM/withdrawATMOK.Guard6 : $atm\_wdam(selfATM) = am$

then

mATM/ATM_SM_leaveState_rspWDOK : $rspWDOK := rspWDOK \setminus \{selfATM\}$

mATM/ATM_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

mATM/withdrawATMOK.Action1 : $atm\_cashA(selfATM) := atm\_cashA(selfATM) - am$

end

Event *withdrawATMFail* $\widehat{=}$

refines *withdrawATMFail*

any

$selfATM$ contextual instance of class atm

c

b

where

mATM/c.type : $c \in ValidCard$

mATM/b.type : $b \in \mathbb{N}$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_rspWDFail : $selfATM \in rspWDFail$

mATM/withdrawATMFail.Guard1 : $selfATM \in dom(atm\_card)$

mATM/withdrawATMFail.Guard2 : $selfATM \in dom(atm\_acbalA)$

mATM/withdrawATMFail.Guard3 : $atm\_acbalA(selfATM) = b$

then

mATM/ATM_SM_leaveState_rspWDFail : $rspWDFail := rspWDFail \setminus \{selfATM\}$

mATM/ATM_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *checkBalATM* $\widehat{=}$

refines *checkBalATM*

any

selfATM contextual instance of class atm

c

b

where

mATM/c.type : $c \in ValidCard$

mATM/b.type : $b \in \mathbb{N}$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_rspCB : $selfATM \in rspCB$

mATM/checkBalATM.Guard1 : $selfATM \in dom(atm\_card)$

mATM/checkBalATM.Guard2 : $selfATM \in dom(atm\_acbalA)$

mATM/checkBalATM.Guard3 : $atm\_acbalA(selfATM) = b$

then

mATM/ATM_SM_leaveState_rspCB : $rspCB := rspCB \setminus \{selfATM\}$

mATM/ATM_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *doAnother* $\widehat{=}$

refines *doAnother*

any

selfATM contextual instance of class atm

c

where

mATM/c.type : $c \in ValidCard$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_endTrans : $selfATM \in endTrans$

mATM/doAnother.Guard1 : $selfATM \in dom(atm\_card)$

mATM/doAnother.Guard2 : $atm\_card(selfATM) = c$

then

mATM/ATM_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

mATM/ATM_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *ejectCard1* $\widehat{=}$

refines *ejectCard1*

any

selfATM contextual instance of class atm

c

where

mATM/c.type : $c \in ValidCard$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_invalidCard : $selfATM \in invalidCard$

mATM/ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

mATM/ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

mATM/ATM_SM_leaveState_invalidCard : $invalidCard := invalidCard \backslash \{selfATM\}$

mATM/ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

mATM/ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *ejectCard2* $\widehat{=}$

refines *ejectCard2*

any

selfATM contextual instance of class atm

c

where

mATM/c.type : $c \in ValidCard$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_trans : $selfATM \in trans$

mATM/ejectCard2.Guard1 : $selfATM \in dom(atm\_card)$

mATM/ejectCard2.Guard2 : $atm\_card(selfATM) = c$

then

mATM/ATM_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

mATM/ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

mATM/ejectCard2.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *ejectCard3* $\widehat{=}$

refines *ejectCard3*

any

$selfATM$ contextual instance of class atm

c

where

mATM/c.type : $c \in ValidCard$

mATM/selfATM.type : $selfATM \in atm$

mATM/ATM_SM_isin_endTrans : $selfATM \in endTrans$

mATM/ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

mATM/ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

mATM/ATM_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

mATM/ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

mATM/ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event $createAccount \mathrel{\widehat{=}}$

refines $createAccount$

any

$self$ contextual instance of class account

where

mBank/self.type : $self \in account$

then

skip

end

Event $deposit \mathrel{\widehat{=}}$

refines $deposit$

any

$self$ contextual instance of class account

where

mBank/self.type : $self \in account$

then

skip

end

END

B.9 Generated Event-B of the Machine *mATM*

MACHINE mATM

SEES mATM_implicitContext

VARIABLES

`atm` class instances

`atm_cashA` attribute of atm

`atm_wdam` attribute of atm

`atm_card` attribute of atm

`atm_acbalA` attribute of atm

`idle` state from statemachine, ATM_SM

`validating` state from statemachine, ATM_SM

`trans` state from statemachine, ATM_SM

`invalidCard` state from statemachine, ATM_SM

`waitingResponse` state from statemachine, ATM_SM

`rspCB` state from statemachine, ATM_SM

`rspWDOK` state from statemachine, ATM_SM

`rspWDFail` state from statemachine, ATM_SM

`endTrans` state from statemachine, ATM_SM

INVARIANTS

`atm.type` : $atm \in \mathbb{P}(ATM)$

`atm_cashA.type` : $atm\_cashA \in atm \rightarrow \mathbb{N}$

`atm_wdam.type` : $atm\_wdam \in atm \nrightarrow \mathbb{N}$

`atm_card.type` : $atm\_card \in atm \nrightarrow ValidCard$

`atm_acbalA.type` : $atm\_acbalA \in atm \nrightarrow \mathbb{N}$

`idle.type` : $idle \in \mathbb{P}(atm)$

`validating.type` : $validating \in \mathbb{P}(atm)$

`trans.type` : $trans \in \mathbb{P}(atm)$

`invalidCard.type` : $invalidCard \in \mathbb{P}(atm)$

`waitingResponse.type` : $waitingResponse \in \mathbb{P}(atm)$

`rspCB.type` : $rspCB \in \mathbb{P}(atm)$

`rspWDOK.type` : $rspWDOK \in \mathbb{P}(atm)$

`rspWDFail.type` : $rspWDFail \in \mathbb{P}(atm)$

endTrans.type : $endTrans \in \mathbb{P}(atm)$

ATM_SM_partitions_atm : $partition(atm, idle, validating, trans, invalidCard, waitingResponse, rspCB, rspWDOK, rspWDFail, endTrans)$

EVENTS

Initialisation

begin

atm.init : $atm := \varnothing$

atm_cashA.init : $atm\_cashA := \varnothing$

atm_wdam.init : $atm\_wdam := \varnothing$

atm_card.init : $atm\_card := \varnothing$

atm_acbalA.init : $atm\_acbalA := \varnothing$

idle.init : $idle := \varnothing$

validating.init : $validating := \varnothing$

trans.init : $trans := \varnothing$

invalidCard.init : $invalidCard := \varnothing$

waitingResponse.init : $waitingResponse := \varnothing$

rspCB.init : $rspCB := \varnothing$

rspWDOK.init : $rspWDOK := \varnothing$

rspWDFail.init : $rspWDFail := \varnothing$

endTrans.init : $endTrans := \varnothing$

end

Event $insertCard \mathrel{\widehat{=}}$

any

$selfATM$ contextual instance of class atm

c

where

selfATM.type : $selfATM \in atm$

c.type : $c \in ValidCard$

ATM_SM_isin_idle : $selfATM \in idle$

insertCard.Guard1 : $selfATM \notin dom(atm\_card)$

then

ATM_SM_leaveState_idle : $idle := idle \setminus \{selfATM\}$

ATM_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

insertCard.Action1 : $atm\_card := atm\_card \cup \{selfATM \mapsto c\}$

end

Event *reloadCash* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

where

selfATM.type : $selfATM \in atm$

ATM_SM_isin_idle : $selfATM \in idle$

reloadCash.Guard1 : $atm\_cashA(selfATM) < MIN\_CASH$

then

reloadCash.Action1 : $atm\_cashA(selfATM) := MAX\_CASH$

end

Event *start* $\widehat{=}$

any

$selfATM$ constructed instance of class atm

where

selfATM.type : $selfATM \in ATM \setminus atm$

then

atm_constructor : $atm := atm \cup \{selfATM\}$

atm.atm_cashA_initialise : $atm\_cashA(selfATM) := MAX\_CASH$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

end

Event *validateCardOK* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

p

where

c.type : $c \in ValidCard$

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_validating : $selfATM \in validating$

validateCardOK.Guard1 : $selfATM \in dom(atm\_card)$

validateCardOK.Guard2 : $atm\_card(selfATM) = c$

validateCardOK.Guard3 : $card\_pin(c) = p$

then

ATM_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

ATM_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *validateCardFail* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

p

where

c.type : $c \in ValidCard$

p.type : $p \in Pin$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_validating : $selfATM \in validating$

validateCardFail.Guard1 : $selfATM \in dom(atm\_card)$

validateCardFail.Guard2 : $atm\_card(selfATM) = c$

validateCardFail.Guard3 : $card\_pin(c) \neq p$

then

ATM_SM_leaveState_validating : $validating := validating \setminus \{selfATM\}$

ATM_SM_enterState_invalidCard : $invalidCard := invalidCard \cup \{selfATM\}$

end

Event *sendReqCB* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_trans : $selfATM \in trans$

sendReqCB.Guard1 : $selfATM \in dom(atm\_card)$

sendReqCB.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

ATM_SM_enterState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

end

Event *sendReqWD* $\widehat{=}$

any

selfATM contextual instance of class atm

c

am

where

c.type : $c \in ValidCard$

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_trans : $selfATM \in trans$

sendReqWD.Guard1 : $selfATM \in dom(atm\_card)$

sendReqWD.Guard2 : $atm\_card(selfATM) = c$

sendReqWD.Guard5 : $atm\_cashA(selfATM) > MIN\_CASH$

sendReqWD.Guard7 : $am \leq MIN\_CASH$

then

ATM_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

ATM_SM_enterState_waitingResponse : $waitingResponse := waitingResponse \cup \{selfATM\}$

sendReqWD.Action1 : $atm\_wdam(selfATM) := am$

end

Event *ejectCard2* $\widehat{=}$

any

selfATM contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_trans : $selfATM \in trans$

ejectCard2.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard2.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_trans : $trans := trans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard2.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *retry* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_invalidCard : $selfATM \in invalidCard$

retry.Guard1 : $selfATM \in dom(atm\_card)$

retry.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_validating : $validating := validating \cup \{selfATM\}$

end

Event *ejectCard1* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_invalidCard : $selfATM \in invalidCard$

ejectCard1.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard1.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_invalidCard : $invalidCard := invalidCard \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard1.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

Event *recvRspCB* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

b

where

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_waitingResponse : $selfATM \in waitingResponse$

recvRspCB.Guard1 : $selfATM \in dom(atm\_card)$

recvRspCB.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

ATM_SM_enterState_rspCB : $rspCB := rspCB \cup \{selfATM\}$

recvRspCB.Action1 : $atm\_acbalA(selfATM) := b$

end

Event *recvRspWDOK* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

b

where

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_waitingResponse : $selfATM \in waitingResponse$

recvRspWDOK.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDOK.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

ATM_SM_enterState_rspWDOK : $rspWDOK := rspWDOK \cup \{selfATM\}$

recvRspWDOK.Action1 : $atm\_acbalA(selfATM) := b$

end

Event *recvRspWDFail* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

b

where

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_waitingResponse : $selfATM \in waitingResponse$

recvRspWDFail.Guard1 : $selfATM \in dom(atm\_card)$

recvRspWDFail.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_waitingResponse : $waitingResponse := waitingResponse \setminus \{selfATM\}$

ATM_SM_enterState_rspWDFail : $rspWDFail := rspWDFail \cup \{selfATM\}$

recvRspWDFail.Action1 : $atm\_acbalA(selfATM) := b$

end

Event *checkBalATM* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

b

where

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_rspCB : $selfATM \in rspCB$

checkBalATM.Guard1 : $selfATM \in dom(atm\_card)$

checkBalATM.Guard2 : $selfATM \in dom(atm\_acbalA)$

checkBalATM.Guard3 : $atm\_acbalA(selfATM) = b$

then

ATM_SM_leaveState_rspCB : $rspCB := rspCB \setminus \{selfATM\}$

ATM_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *withdrawATMOK* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

am

b

where

c.type : $c \in ValidCard$

am.type : $am \in \mathbb{N}$

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_rspWDOK : $selfATM \in rspWDOK$

withdrawATMOK.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMOK.Guard2 : $selfATM \in dom(atm\_acbalA)$

withdrawATMOK.Guard3 : $atm\_cashA(selfATM) \geq am$

withdrawATMOK.Guard4 : $atm\_acbalA(selfATM) = b$

withdrawATMOK.Guard5 : $selfATM \in dom(atm\_wdam)$

withdrawATMOK.Guard6 : $atm\_wdam(selfATM) = am$

then

ATM_SM_leaveState_rspWDOK : $rspWDOK := rspWDOK \setminus \{selfATM\}$

ATM_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

withdrawATMOK.Action1 : $atm\_cashA(selfATM) := atm\_cashA(selfATM) - am$

end

Event *withdrawATMFail* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

b

where

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_rspWDFail : $selfATM \in rspWDFail$

withdrawATMFail.Guard1 : $selfATM \in dom(atm\_card)$

withdrawATMFail.Guard2 : $selfATM \in dom(atm\_acbalA)$

withdrawATMFail.Guard3 : $atm\_acbalA(selfATM) = b$

then

ATM_SM_leaveState_rspWDFail : $rspWDFail := rspWDFail \setminus \{selfATM\}$

ATM_SM_enterState_endTrans : $endTrans := endTrans \cup \{selfATM\}$

end

Event *doAnother* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_endTrans : $selfATM \in endTrans$

doAnother.Guard1 : $selfATM \in dom(atm\_card)$

doAnother.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

ATM_SM_enterState_trans : $trans := trans \cup \{selfATM\}$

end

Event *ejectCard3* $\widehat{=}$

any

$selfATM$ contextual instance of class atm

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atm$

ATM_SM_isin_endTrans : $selfATM \in endTrans$

ejectCard3.Guard1 : $selfATM \in dom(atm\_card)$

ejectCard3.Guard2 : $atm\_card(selfATM) = c$

then

ATM_SM_leaveState_endTrans : $endTrans := endTrans \setminus \{selfATM\}$

ATM_SM_enterState_idle : $idle := idle \cup \{selfATM\}$

ejectCard3.Action1 : $atm\_card := atm\_card \setminus \{selfATM \mapsto c\}$

end

END

B.10 Generated Event-B of the Machine *mBank*

MACHINE mBank

SEES mBank_implicitContext

VARIABLES

`account` class instances

`atmB` class instances

`bal` attribute of account

`atm_acbalB` attribute of atmB

`atm_cardB` attribute of atmB

`atm_wdamB` attribute of atmB

`recvdReqCB` state from statemachine, bankSM

`recvdReqWD` state from statemachine, bankSM

`processCB` state from statemachine, bankSM

`processWDOK` state from statemachine, bankSM

`processWDFail` state from statemachine, bankSM

INVARIANTS

`account.type` : $account \in \mathbb{P}(Account)$

`atmB.type` : $atmB \in \mathbb{P}(ATM)$

`bal.type` : $bal \in account \rightarrow \mathbb{N}$

`atm_acbalB.type` : $atm\_acbalB \in atmB \nrightarrow \mathbb{N}$

`atm_cardB.type` : $atm\_cardB \in atmB \rightarrow ValidCard$

`atm_wdamB.type` : $atm\_wdamB \in atmB \nrightarrow \mathbb{N}$

`recvdReqCB.type` : $recvdReqCB \in \mathbb{P}(atmB)$

`recvdReqWD.type` : $recvdReqWD \in \mathbb{P}(atmB)$

`processCB.type` : $processCB \in \mathbb{P}(atmB)$

`processWDOK.type` : $processWDOK \in \mathbb{P}(atmB)$

`processWDFail.type` : $processWDFail \in \mathbb{P}(atmB)$

`bankSM_partitions_atmB` : $partition(atmB, recvdReqCB, recvdReqWD, processCB, processWDOK, processWDFail)$

EVENTS

Initialisation

begin

`account.init` : $account := \varnothing$

`atmB.init` : $atmB := \varnothing$

`bal.init` : $bal := \varnothing$

`atm_acbalB.init` : $atm\_acbalB := \varnothing$

`atm_cardB.init` : $atm\_cardB := \varnothing$

`atm_wdamB.init` : $atm\_wdamB := \varnothing$

`recvdReqCB.init` : $recvdReqCB := \varnothing$

`recvdReqWD.init` : $recvdReqWD := \varnothing$

`processCB.init` : $processCB := \varnothing$

`processWDOK.init` : $processWDOK := \varnothing$

`processWDFail.init` : $processWDFail := \varnothing$

end

Event *createAccount* $\widehat{=}$

any

self contextual instance of class account

where

`self.type` : $self \in account$

then

`skip`

end

Event *deposit* $\widehat{=}$

any

self contextual instance of class account

where

`self.type` : $self \in account$

then

`skip`

end

Event *recvReqCB* $\widehat{=}$

any

selfATM constructed instance of class atmB

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in ATM \setminus atmB$

then

atmB_constructor : $atmB := atmB \cup \{selfATM\}$

bankSM_enterState_recvdReqCB : $recvdReqCB := recvdReqCB \cup \{selfATM\}$

recvReqCB.Action2 : $atm\_cardB(selfATM) := c$

end

Event *checkBalance* $\widehat{=}$

any

$selfATM$ contextual instance of class atmB

c

ac

b

where

c.type : $c \in ValidCard$

ac.type : $ac \in account$

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atmB$

bankSM_isin_recvdReqCB : $selfATM \in recvdReqCB$

checkBalance.Guard5 : $b = bal(ac)$

checkBalance.Guard2 : $card\_account(c) = ac$

checkBalance.Guard6 : $atm\_cardB(selfATM) = c$

then

bankSM_leaveState_recvdReqCB : $recvdReqCB := recvdReqCB \setminus \{selfATM\}$

bankSM_enterState_processCB : $processCB := processCB \cup \{selfATM\}$

checkBalance.Action2 : $atm\_acbalB(selfATM) := bal(ac)$

end

Event *withdrawOK* $\widehat{=}$

any

$selfATM$ contextual instance of class atmB

c

ac

am

where

c.type : $c \in ValidCard$

ac.type : $ac \in account$

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atmB$

bankSM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawOK.Guard1 : $selfATM \in dom(atm\_wdamB)$

withdrawOK.Guard4 : $bal(ac) \geq am$

withdrawOK.Guard2 : $card\_account(c) = ac$

withdrawOK.Guard7 : $atm\_cardB(selfATM) = c$

withdrawOK.Guard5 : $atm\_wdamB(selfATM) = am$

then

bankSM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

bankSM_enterState_processWDOK : $processWDOK := processWDOK \cup \{selfATM\}$

withdrawOK.Action1 : $bal(ac) := bal(ac) - am$

withdrawOK.Action2 : $atm\_acbalB(selfATM) := bal(ac) - am$

end

Event *withdrawFail* $\widehat{=}$

any

$selfATM$ contextual instance of class atmB

c

ac

am

where

c.type : $c \in ValidCard$

ac.type : $ac \in account$

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in atmB$

bankSM_isin_recvdReqWD : $selfATM \in recvdReqWD$

withdrawFail.Guard1 : $selfATM \in dom(atm\_wdamB)$

withdrawFail.Guard5 : $bal(ac) < am$

withdrawFail.Guard2 : $card\_account(c) = ac$

withdrawFail.Guard7 : $atm\_cardB(selfATM) = c$

withdrawFail.Guard4 : $atm\_wdamB(selfATM) = am$

then

bankSM_leaveState_recvdReqWD : $recvdReqWD := recvdReqWD \setminus \{selfATM\}$

bankSM_enterState_processWDFail : $processWDFail := processWDFail \cup \{selfATM\}$

withdrawFail.Action2 : $atm\_acbalB(selfATM) := bal(ac)$

end

Event *recvReqWD* $\widehat{=}$

any

selfATM constructed instance of class atmB

c

am

where

c.type : $c \in ValidCard$

am.type : $am \in \mathbb{N}$

selfATM.type : $selfATM \in ATM \setminus atmB$

then

atmB_constructor : $atmB := atmB \cup \{selfATM\}$

bankSM_enterState_recvdReqWD : $recvdReqWD := recvdReqWD \cup \{selfATM\}$

recvReqWD.Action2 : $atm\_cardB(selfATM) := c$

recvReqWD.Action3 : $atm\_wdamB(selfATM) := am$

end

Event *sendRspCB* $\widehat{=}$

any

selfATM contextual instance of class atmB

c

b

where

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atmB$

c.type : $c \in ValidCard$

bankSM_isin_processCB : $selfATM \in processCB$

sendRspCB.Guard4 : $selfATM \in dom(atm\_acbalB)$

sendRspCB.Guard7 : $atm\_acbalB(selfATM) = b$

sendRspCB.Guard1 : $atm\_cardB(selfATM) = c$

then

bankSM_leaveState_processCB : $processCB := processCB \setminus \{selfATM\}$

atmB_destructor : $atmB := atmB \setminus \{selfATM\}$

atmB.atm_acbalB_destructor : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

atmB.atm_cardB_destructor : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

atmB.atm_wdamB_destructor : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event *sendRspWDOK* $\widehat{=}$

any

selfATM contextual instance of class atmB

c

b

where

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atmB$

c.type : $c \in ValidCard$

bankSM_isin_processWDOK : $selfATM \in processWDOK$

sendRspWDOK.Guard5 : $selfATM \in dom(atm\_acbalB)$

sendRspWDOK.Guard3 : $atm\_acbalB(selfATM) = b$

sendRspWDOK.Guard1 : $atm\_cardB(selfATM) = c$

then

bankSM_leaveState_processWDOK : $processWDOK := processWDOK \setminus \{selfATM\}$

atmB_destructor : $atmB := atmB \setminus \{selfATM\}$

atmB.atm_acbalB_destructor : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

atmB.atm_cardB_destructor : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

atmB.atm_wdamB_destructor : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

Event *sendRspWDFail* $\widehat{=}$

any

selfATM contextual instance of class atmB

c

b

where

b.type : $b \in \mathbb{N}$

selfATM.type : $selfATM \in atmB$

c.type : $c \in ValidCard$

bankSM_isin_processWDFail : $selfATM \in processWDFail$

sendRspWDFail.Guard5 : $selfATM \in dom(atm\_acbalB)$

sendRspWDFail.Guard3 : $atm\_acbalB(selfATM) = b$

sendRspWDFail.Guard1 : $atm\_cardB(selfATM) = c$

then

bankSM_leaveState_processWDFail : $processWDFail := processWDFail \setminus \{selfATM\}$

atmB_destructor : $atmB := atmB \setminus \{selfATM\}$

atmB.atm_acbalB_destructor : $atm\_acbalB := \{selfATM\} \lhd atm\_acbalB$

atmB.atm_cardB_destructor : $atm\_cardB := \{selfATM\} \lhd atm\_cardB$

atmB.atm_wdamB_destructor : $atm\_wdamB := \{selfATM\} \lhd atm\_wdamB$

end

END

B.11 Generated Event-B of the Machine *mMW*

MACHINE mMW

SEES mMW_implicitContext

VARIABLES

`atmM` class instances

`atm_acbalM` attribute of atmM

`atm_wdamM` attribute of atmM

`atm_cardM` attribute of atmM

`sentReqWD` state from statemachine, reqWD_SM

`sentReqCB` state from statemachine, reqCB_SM

`sentRspWDOK` state from statemachine, rspWDOK_SM

`sentRspWDFail` state from statemachine, rspWDFail_SM

`sentRspCB` state from statemachine, rspCB_SM

INVARIANTS

`atmM.type` : $atmM \in \mathbb{P}(ATM)$

`atm_acbalM.type` : $atm\_acbalM \in atmM \pfun \mathbb{N}$

`atm_wdamM.type` : $atm\_wdamM \in atmM \pfun \mathbb{N}$

`atm_cardM.type` : $atm\_cardM \in atmM \fun ValidCard$

`sentReqWD.type` : $sentReqWD \in \mathbb{P}(atmM)$

`sentReqCB.type` : $sentReqCB \in \mathbb{P}(atmM)$

`sentRspWDOK.type` : $sentRspWDOK \in \mathbb{P}(atmM)$

`sentRspWDFail.type` : $sentRspWDFail \in \mathbb{P}(atmM)$

`sentRspCB.type` : $sentRspCB \in \mathbb{P}(atmM)$

`partitions_atmM` : $partition(atmM, sentReqWD, sentReqCB, sentRspWDOK, sentRspWDFail, sentRspCB)$

EVENTS

Initialisation

begin

`atmM.init` : $atmM := \varnothing$

`atm_acbalM.init` : $atm\_acbalM := \varnothing$

`atm_wdamM.init` : $atm\_wdamM := \varnothing$

`atm_cardM.init` : $atm\_cardM := \varnothing$

`sentReqWD.init` : $sentReqWD := \varnothing$

`sentReqCB.init` : $sentReqCB := \varnothing$

`sentRspWDOK.init` : $sentRspWDOK := \varnothing$

`sentRspWDFail.init` : $sentRspWDFail := \varnothing$

`sentRspCB.init` : $sentRspCB := \varnothing$

end

Event $sendReqWD \mathrel{\widehat{=}}$

any

$selfATM$ constructed instance of class atmM

am

c

where

`am.type` : $am \in \mathbb{N}$

`c.type` : $c \in ValidCard$

`selfATM.type` : $selfATM \in ATM \setminus atmM$

then

`atmM_constructor` : $atmM := atmM \cup \{selfATM\}$

`reqWD_SM_enterState_sentReqWD` : $sentReqWD := sentReqWD \cup \{selfATM\}$

`sendReqWD.Action2` : $atm\_wdamM(selfATM) := am$

`sendReqWD.Action1` : $atm\_cardM(selfATM) := c$

end

Event $recvReqWD \mathrel{\widehat{=}}$

any

$selfATM$ contextual instance of class atmM

c

am

where

`c.type` : $c \in ValidCard$

`am.type` : $am \in \mathbb{N}$

`selfATM.type` : $selfATM \in atmM$

`reqWD_SM_isin_sentReqWD` : $selfATM \in sentReqWD$

`recvReqWD.Guard1` : $atm\_cardM(selfATM) = c$

`recvReqWD.Guard2` : $selfATM \in dom(atm\_wdamM)$

`recvReqWD.Guard3` : $atm\_wdamM(selfATM) = am$

then

reqWD_SM_leaveState_sentReqWD : $sentReqWD := sentReqWD \setminus \{selfATM\}$

atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

end

Event *sendReqCB* $\widehat{=}$

any

$selfATM$ constructed instance of class atmM

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in ATM \setminus atmM$

then

atmM_constructor : $atmM := atmM \cup \{selfATM\}$

reqCB_SM_enterState_sentReqCB : $sentReqCB := sentReqCB \cup \{selfATM\}$

sendReqCB.Action1 : $atm\_cardM(selfATM) := c$

end

Event *recvReqCB* $\widehat{=}$

any

$selfATM$ contextual instance of class atmM

c

where

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atmM$

reqCB_SM_isin_sentReqCB : $selfATM \in sentReqCB$

recvReqCB.Guard1 : $atm\_cardM(selfATM) = c$

then

reqCB_SM_leaveState_sentReqCB : $sentReqCB := sentReqCB \setminus \{selfATM\}$

atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

end

Event *sendRspWDOK* $\widehat{=}$

any

selfATM constructed instance of class atmM

b

c

where

b.type : $b \in \mathbb{N}$

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in ATM \setminus atmM$

then

atmM_constructor : $atmM := atmM \cup \{selfATM\}$

rspWDOK_SM_enterState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \cup \{selfATM\}$

sendRspWDOK.Action1 : $atm\_acbalM(selfATM) := b$

sendRspWDOK.Action2 : $atm\_cardM(selfATM) := c$

end

Event *recvRspWDOK* $\widehat{=}$

any

selfATM contextual instance of class atmM

b

c

where

b.type : $b \in \mathbb{N}$

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atmM$

rspWDOK_SM_isin_sentRspWDOK : $selfATM \in sentRspWDOK$

recvRspWDOK.Guard1 : $selfATM \in dom(atm\_acbalM)$

recvRspWDOK.Guard2 : $atm\_acbalM(selfATM) = b$

recvRspWDOK.Guard3 : $atm\_cardM(selfATM) = c$

then

rspWDOK_SM_leaveState_sentRspWDOK : $sentRspWDOK := sentRspWDOK \setminus \{selfATM\}$

atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

end

Event *sendRspWDFail* $\widehat{=}$

any

selfATM constructed instance of class atmM

b

c

where

b.type : $b \in \mathbb{N}$

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in ATM \setminus atmM$

then

atmM_constructor : $atmM := atmM \cup \{selfATM\}$

rspWDFail_SM_enterState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \cup \{selfATM\}$

sendRspWDFail.Action1 : $atm\_acbalM(selfATM) := b$

sendRspWDFail.Action2 : $atm\_cardM(selfATM) := c$

end

Event *recvRspWDFail* $\widehat{=}$

any

selfATM contextual instance of class atmM

b

c

where

b.type : $b \in \mathbb{N}$

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atmM$

rspWDFail_SM_isin_sentRspWDFail : $selfATM \in sentRspWDFail$

recvRspWDFail.Guard1 : $selfATM \in dom(atm\_acbalM)$

recvRspWDFail.Guard2 : $atm\_acbalM(selfATM) = b$

recvRspWDFail.Guard3 : $atm\_cardM(selfATM) = c$

then

rspWDFail_SM_leaveState_sentRspWDFail : $sentRspWDFail := sentRspWDFail \setminus \{selfATM\}$

atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

end

Event *sendRspCB* $\widehat{=}$

any

selfATM constructed instance of class atmM

b

c

where

b.type : $b \in \mathbb{N}$

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in ATM \setminus atmM$

then

atmM_constructor : $atmM := atmM \cup \{selfATM\}$

rspCB_SM_enterState_sentRspCB : $sentRspCB := sentRspCB \cup \{selfATM\}$

sendRspCB.Action1 : $atm\_acbalM(selfATM) := b$

sendRspCB.Action2 : $atm\_cardM(selfATM) := c$

end

Event *recvRspCB* $\widehat{=}$

any

selfATM contextual instance of class atmM

b

c

where

b.type : $b \in \mathbb{N}$

c.type : $c \in ValidCard$

selfATM.type : $selfATM \in atmM$

rspCB_SM_isin_sentRspCB : $selfATM \in sentRspCB$

recvRspCB.Guard1 : $selfATM \in dom(atm\_acbalM)$

recvRspCB.Guard2 : $atm\_acbalM(selfATM) = b$

recvRspCB.Guard3 : $atm\_cardM(selfATM) = c$

then

rspCB_SM_leaveState_sentRspCB : $sentRspCB := sentRspCB \setminus \{selfATM\}$

atmM_destructor : $atmM := atmM \setminus \{selfATM\}$

atmM.atm_acbalM_destructor : $atm\_acbalM := \{selfATM\} \lhd atm\_acbalM$

atmM.atm_wdamM_destructor : $atm\_wdamM := \{selfATM\} \lhd atm\_wdamM$

atmM.atm_cardM_destructor : $atm\_cardM := \{selfATM\} \lhd atm\_cardM$

end

END

B.12 Generated Event-B of the Context *ATM_CXR3*

CONTEXT ATM_CXR3

EXTENDS ATM_CXR2

SETS

`MSG` ClassType

`STATUS` ClassType

CONSTANTS

`REQ_MSG` classType instances

`RSP_MSG` classType instances

`REQ_WD_MSG` classType instances

`REQ_CB_MSG` classType instances

`RSP_WDOK_MSG` classType instances

`RSP_WDFAIL_MSG` classType instances

`RSP_CB_MSG` classType instances

`OK` enumeration constant

`NOT_OK` enumeration constant

`msg_atm` attribute of MSG

`msg_card` attribute of MSG

`msg_status` attribute of RSP_MSG

`msg_bal` attribute of RSP_MSG

`msg_wdAmount` attribute of REQ_WD_MSG

AXIOMS

`REQ_MSG.type` : $REQ\_MSG \in \mathbb{P}(MSG)$

`RSP_MSG.type` : $RSP\_MSG \in \mathbb{P}(MSG)$

`REQ_WD_MSG.type` : $REQ\_WD\_MSG \in \mathbb{P}(REQ\_MSG)$

`REQ_CB_MSG.type` : $REQ\_CB\_MSG \in \mathbb{P}(REQ\_MSG)$

`RSP_WDOK_MSG.type` : $RSP\_WDOK\_MSG \in \mathbb{P}(RSP\_MSG)$

`RSP_WDFAIL_MSG.type` : $RSP\_WDFAIL\_MSG \in \mathbb{P}(RSP\_MSG)$

`RSP_CB_MSG.type` : $RSP\_CB\_MSG \in \mathbb{P}(RSP\_MSG)$

`OK.type` : $OK \in STATUS$

`NOT_OK.type` : $NOT\_OK \in STATUS$

`msg_atm.type` : $msg\_atm \in MSG \rightarrow ATM$

msg_card.type : $msg\_card \in MSG \rightarrow ValidCard$

msg_status.type : $msg\_status \in RSP\_MSG \rightarrow STATUS$

msg_bal.type : $msg\_bal \in RSP\_MSG \rightarrow \mathbb{N}$

msg_wdAmount.type : $msg\_wdAmount \in REQ\_WD\_MSG \rightarrow \mathbb{N}$

Axiom1 : $partition(MSG, REQ\_MSG, RSP\_MSG)$

Axiom2 : $partition(REQ\_MSG, REQ\_WD\_MSG, REQ\_CB\_MSG)$

Axiom3 : $partition(RSP\_MSG, RSP\_WDOK\_MSG, RSP\_WDFAIL\_MSG, RSP\_CB\_MSG)$

partition enumeration of STATUS : $partition(STATUS, \{OK\}, \{NOT\_OK\})$

END

B.13 Generated Event-B of the Machine *mMW_R1*

MACHINE mMW_R1

REFINES mMW

SEES mMW_R1_implicitContext

VARIABLES

`msg` class instances

`reqwdmsg` state from statemachine, reqwd_SM

`reqcbmsg` state from statemachine, reqCB_SM

`rspwdokmsg` state from statemachine, rspwdok_SM

`rspwdfailmsg` state from statemachine, rspwdfail_SM

`rspcbmsg` state from statemachine, rspcb_SM

INVARIANTS

`msg.type` : $msg \in \mathbb{P}(MSG)$

`reqwdmsg.type` : $reqwdmsg \in \mathbb{P}(msg)$

`reqcbmsg.type` : $reqcbmsg \in \mathbb{P}(msg)$

`rspwdokmsg.type` : $rspwdokmsg \in \mathbb{P}(msg)$

`rspwdfailmsg.type` : $rspwdfailmsg \in \mathbb{P}(msg)$

`rspcbmsg.type` : $rspcbmsg \in \mathbb{P}(msg)$

`partitions_msg` : $partition(msg, reqwdmsg, reqcbmsg, rspwdokmsg, rspwdfailmsg, rspcbmsg)$

`Invariant1` : $reqwdmsg \in \mathbb{P}(REQ\_WD\_MSG)$

`Invariant2` : $reqcbmsg \in \mathbb{P}(REQ\_CB\_MSG)$

`Invariant3` : $rspwdokmsg \in \mathbb{P}(RSP\_WDOK\_MSG)$

`Invariant4` : $rspwdfailmsg \in \mathbb{P}(RSP\_WDFAIL\_MSG)$

`Invariant5` : $rspcbmsg \in \mathbb{P}(RSP\_CB\_MSG)$

`Invariant6` : $\forall m \cdot m \in msg \Rightarrow msg\_atm(m) \mapsto msg\_card(m) \in atm\_cardM$

`Invariant7` : $\forall m \cdot m \in reqwdmsg \Rightarrow msg\_atm(m) \mapsto msg\_wdAmount(m) \in atm\_wdamM$

`Invariant8` : $\forall m \cdot m \in (rspwdokmsg \cup rspwdfailmsg \cup rspcbmsg) \Rightarrow msg\_atm(m) \mapsto msg\_bal(m) \in atm\_acbalM$

`Invariant9` : $sentReqWD = msg\_atm[reqwdmsg]$

`Invariant10` : $sentReqCB = msg\_atm[reqcbmsg]$

`Invariant11` : $sentRspWDOK = msg\_atm[rspwdokmsg]$

`Invariant12` : $sentRspWDFail = msg\_atm[rspwdfailmsg]$

Invariant13 : $sentRspCB = msg\_atm[rspcbmsg]$

Invariant14 : $\forall m, m0 \cdot m \in msg \land m0 \in msg \land msg\_atm(m) = msg\_atm(m0) \Rightarrow m = m0$

EVENTS

Initialisation

begin

msg.init : $msg := \varnothing$

reqwdmsg.init : $reqwdmsg := \varnothing$

reqcbmsg.init : $reqcbmsg := \varnothing$

rspwdokmsg.init : $rspwdokmsg := \varnothing$

rspwdfailmsg.init : $rspwdfailmsg := \varnothing$

rspcbmsg.init : $rspcbmsg := \varnothing$

end

Event $sendReqWD \mathrel{\widehat{=}}$

refines $sendReqWD$

any

$selfMsg$ constructed instance of class msg

$selfATM$

c

am

where

selfMsg.type : $selfMsg \in MSG \setminus msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

am.type : $am \in \mathbb{N}$

sendReqWD.Guard1 : $selfMsg \in REQ\_WD\_MSG$

sendReqWD.Guard2 : $selfATM \notin msg\_atm[msg]$

sendReqWD.Guard3 : $msg\_atm(selfMsg) = selfATM$

sendReqWD.Guard4 : $msg\_card(selfMsg) = c$

sendReqWD.Guard5 : $msg\_wdAmount(selfMsg) = am$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

reqwd_SM_enterState_reqwdmsg : $reqwdmsg := reqwdmsg \cup \{selfMsg\}$

end

Event *recvReqWD* $\widehat{=}$

refines *recvReqWD*

any

selfMsg contextual instance of class msg

selfATM

c

am

where

`selfMsg.type :` $selfMsg \in msg$

`selfATM.type :` $selfATM \in ATM$

`c.type :` $c \in ValidCard$

`am.type :` $am \in \mathbb{N}$

`reqwd_SM_isin_reqwdmsg :` $selfMsg \in reqwdmsg$

`recvReqWD.Guard1 :` $msg\_atm(selfMsg) = selfATM$

`recvReqWD.Guard2 :` $msg\_card(selfMsg) = c$

`recvReqWD.Guard3 :` $msg\_wdAmount(selfMsg) = am$

then

`reqwd_SM_leaveState_reqwdmsg :` $reqwdmsg := reqwdmsg \setminus \{selfMsg\}$

`msg_destructor :` $msg := msg \setminus \{selfMsg\}$

end

Event *sendReqCB* $\widehat{=}$

refines *sendReqCB*

any

selfMsg constructed instance of class msg

selfATM

c

where

`selfMsg.type :` $selfMsg \in MSG \setminus msg$

`selfATM.type :` $selfATM \in ATM$

`c.type :` $c \in ValidCard$

`sendReqCB.Guard1 :` $selfMsg \in REQ\_CB\_MSG$

`sendReqCB.Guard2 :` $selfATM \notin msg\_atm[msg]$

`sendReqCB.Guard3 :` $msg\_atm(selfMsg) = selfATM$

`sendReqCB.Guard4 :` $msg\_card(selfMsg) = c$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

reqCB_SM_enterState_reqcbmsg : $reqcbmsg := reqcbmsg \cup \{selfMsg\}$

end

Event *recvReqCB* $\widehat{=}$

refines *recvReqCB*

any

selfMsg contextual instance of class msg

selfATM

c

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

reqCB_SM_isin_reqcbmsg : $selfMsg \in reqcbmsg$

recvReqCB.Guard1 : $msg\_atm(selfMsg) = selfATM$

recvReqCB.Guard2 : $msg\_card(selfMsg) = c$

then

reqCB_SM_leaveState_reqcbmsg : $reqcbmsg := reqcbmsg \setminus \{selfMsg\}$

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

Event *sendRspWDOK* $\widehat{=}$

refines *sendRspWDOK*

any

selfMsg constructed instance of class msg

selfATM

c

b

where

selfMsg.type : $selfMsg \in MSG \setminus msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

sendRspWDOK.Guard1 : $selfMsg \in RSP\_WDOK\_MSG$

sendRspWDOK.Guard2 : $selfATM \notin msg\_atm[msg]$

sendRspWDOK.Guard3 : $msg\_atm(selfMsg) = selfATM$

sendRspWDOK.Guard4 : $msg\_card(selfMsg) = c$

sendRspWDOK.Guard5 : $msg\_bal(selfMsg) = b$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

rspwdok_SM_enterState_rspwdokmsg : $rspwdokmsg := rspwdokmsg \cup \{selfMsg\}$

end

Event *recvRspWDOK* $\widehat{=}$

refines *recvRspWDOK*

any

selfMsg contextual instance of class msg

selfATM

c

b

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

rspwdok_SM_isin_rspwdokmsg : $selfMsg \in rspwdokmsg$

recvRspWDOK.Guard1 : $msg\_atm(selfMsg) = selfATM$

recvRspWDOK.Guard2 : $msg\_card(selfMsg) = c$

recvRspWDOK.Guard3 : $msg\_bal(selfMsg) = b$

then

rspwdok_SM_leaveState_rspwdokmsg : $rspwdokmsg := rspwdokmsg \setminus \{selfMsg\}$

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

Event *sendRspWDFail* $\widehat{=}$

refines *sendRspWDFail*

any

selfMsg constructed instance of class msg

selfATM

c

b

where

selfMsg.type : $selfMsg \in MSG \setminus msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

sendRspWDFail.Guard1 : $selfMsg \in RSP\_WDFAIL\_MSG$

sendRspWDFail.Guard2 : $selfATM \notin msg\_atm[msg]$

sendRspWDFail.Guard3 : $msg\_atm(selfMsg) = selfATM$

sendRspWDFail.Guard4 : $msg\_card(selfMsg) = c$

sendRspWDFail.Guard5 : $msg\_bal(selfMsg) = b$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

rspwdfail_SM_enterState_rspwdfailmsg : $rspwdfailmsg := rspwdfailmsg \cup \{selfMsg\}$

end

Event *recvRspWDFail* $\widehat{=}$

refines *recvRspWDFail*

any

$selfMsg$ contextual instance of class msg

$selfATM$

c

b

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

rspwdfail_SM_isin_rspwdfailmsg : $selfMsg \in rspwdfailmsg$

recvRspWDFail.Guard1 : $msg\_atm(selfMsg) = selfATM$

recvRspWDFail.Guard2 : $msg\_card(selfMsg) = c$

recvRspWDFail.Guard3 : $msg\_bal(selfMsg) = b$

then

rspwdfail_SM_leaveState_rspwdfailmsg : $rspwdfailmsg := rspwdfailmsg \setminus \{selfMsg\}$

`msg_destructor` : $msg := msg \setminus \{selfMsg\}$

end

Event *sendRspCB* $\widehat{=}$

refines *sendRspCB*

any

selfMsg constructed instance of class msg

selfATM

c

b

where

`selfMsg.type` : $selfMsg \in MSG \setminus msg$

`selfATM.type` : $selfATM \in ATM$

`c.type` : $c \in ValidCard$

`b.type` : $b \in \mathbb{N}$

`sendRspCB.Guard1` : $selfMsg \in RSP\_CB\_MSG$

`sendRspCB.Guard2` : $selfATM \notin msg\_atm[msg]$

`sendRspCB.Guard3` : $msg\_atm(selfMsg) = selfATM$

`sendRspCB.Guard4` : $msg\_card(selfMsg) = c$

`sendRspCB.Guard5` : $msg\_bal(selfMsg) = b$

then

`msg_constructor` : $msg := msg \cup \{selfMsg\}$

`rspcb_SM_enterState_rspcbmsg` : $rspcbmsg := rspcbmsg \cup \{selfMsg\}$

end

Event *recvRspCB* $\widehat{=}$

refines *recvRspCB*

any

selfMsg contextual instance of class msg

selfATM

c

b

where

`selfMsg.type` : $selfMsg \in msg$

`selfATM.type` : $selfATM \in ATM$

`c.type` : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

rspcb_SM_isin_rspcbmsg : $selfMsg \in rspcbmsg$

recvRspCB.Guard1 : $msg\_atm(selfMsg) = selfATM$

recvRspCB.Guard2 : $msg\_card(selfMsg) = c$

recvRspCB.Guard3 : $msg\_bal(selfMsg) = b$

then

rspcb_SM_leaveState_rspcbmsg : $rspcbmsg := rspcbmsg \setminus \{selfMsg\}$

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

END

B.14 Generated Event-B of the Machine *mMW_R2*

MACHINE mMW_R2

REFINES mMW_R1

SEES mMW_R2_implicitContext

VARIABLES

`msg` refined class instances

EVENTS

Initialisation

begin

`msg.init` : $msg := \varnothing$

end

Event *sendReqWD* $\widehat{=}$

refines *sendReqWD*

any

selfMsg constructed instance of class msg

selfATM

c

am

where

`selfMsg.type` : $selfMsg \in MSG \setminus msg$

`selfATM.type` : $selfATM \in ATM$

`c.type` : $c \in ValidCard$

`am.type` : $am \in \mathbb{N}$

`sendReqWD.Guard1` : $selfMsg \in REQ\_WD\_MSG$

`sendReqWD.Guard2` : $selfATM \notin msg\_atm[msg]$

`sendReqWD.Guard3` : $msg\_atm(selfMsg) = selfATM$

`sendReqWD.Guard4` : $msg\_card(selfMsg) = c$

`sendReqWD.Guard5` : $msg\_wdAmount(selfMsg) = am$

then

`msg_constructor` : $msg := msg \cup \{selfMsg\}$

end

Event *sentReqCB* $\widehat{=}$

refines *sendReqCB*

any

selfMsg constructed instance of class msg

selfATM

c

where

selfMsg.type : $selfMsg \in MSG \setminus msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

sentReqCB.Guard1 : $selfMsg \in REQ\_CB\_MSG$

sentReqCB.Guard2 : $selfATM \notin msg\_atm[msg]$

sentReqCB.Guard3 : $msg\_atm(selfMsg) = selfATM$

sentReqCB.Guard4 : $msg\_card(selfMsg) = c$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

end

Event *sentRspWDOK* $\widehat{=}$

refines *sendRspWDOK*

any

selfMsg constructed instance of class msg

selfATM

c

b

where

selfMsg.type : $selfMsg \in MSG \setminus msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

sentRspWDOK.Guard1 : $selfMsg \in RSP\_WDOK\_MSG$

sentRspWDOK.Guard2 : $selfATM \notin msg\_atm[msg]$

sentRspWDOK.Guard3 : $msg\_atm(selfMsg) = selfATM$

sentRspWDOK.Guard4 : $msg\_card(selfMsg) = c$

sentRspWDOK.Guard5 : $msg\_bal(selfMsg) = b$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

end

Event *sentRspWDFail* $\widehat{=}$

refines *sendRspWDFail*

any

selfMsg constructed instance of class msg

selfATM

c

b

where

selfMsg.type : $selfMsg \in MSG \setminus msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

sentRspWDFail.Guard1 : $selfMsg \in RSP\_WDFAIL\_MSG$

sentRspWDFail.Guard2 : $selfATM \notin msg\_atm[msg]$

sentRspWDFail.Guard3 : $msg\_atm(selfMsg) = selfATM$

sentRspWDFail.Guard4 : $msg\_card(selfMsg) = c$

sentRspWDFail.Guard5 : $msg\_bal(selfMsg) = b$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

end

Event *sentRspCB* $\widehat{=}$

refines *sendRspCB*

any

selfMsg constructed instance of class msg

selfATM

c

b

where

selfMsg.type : $selfMsg \in MSG \setminus msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

sentRspCB.Guard1 : $selfMsg \in RSP\_CB\_MSG$

sentRspCB.Guard2 : $selfATM \notin msg\_atm[msg]$

sentRspCB.Guard3 : $msg\_atm(selfMsg) = selfATM$

sentRspCB.Guard4 : $msg\_card(selfMsg) = c$

sentRspCB.Guard5 : $msg\_bal(selfMsg) = b$

then

msg_constructor : $msg := msg \cup \{selfMsg\}$

end

Event *recvReqWD* $\widehat{=}$

refines *recvReqWD*

any

selfMsg contextual instance of refined class msg

selfATM

c

am

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

am.type : $am \in \mathbb{N}$

recvReqWD.Guard1 : $selfMsg \in REQ\_WD\_MSG$

recvReqWD.Guard2 : $msg\_atm(selfMsg) = selfATM$

recvReqWD.Guard3 : $msg\_card(selfMsg) = c$

recvReqWD.Guard4 : $msg\_wdAmount(selfMsg) = am$

then

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

Event *recvReqCB* $\widehat{=}$

refines *recvReqCB*

any

selfMsg contextual instance of refined class msg

selfATM

c

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

recvReqCB.Guard1 : $selfMsg \in REQ\_CB\_MSG$

recvReqCB.Guard2 : $msg\_atm(selfMsg) = selfATM$

recvReqCB.Guard3 : $msg\_card(selfMsg) = c$

then

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

Event $recvRspWDOK \mathrel{\widehat{=}}$

refines $recvRspWDOK$

any

$selfMsg$ contextual instance of refined class msg

$selfATM$

c

b

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

recvRspWDOK.Guard1 : $selfMsg \in RSP\_WDOK\_MSG$

recvRspWDOK.Guard2 : $msg\_atm(selfMsg) = selfATM$

recvRspWDOK.Guard3 : $msg\_card(selfMsg) = c$

recvRspWDOK.Guard4 : $msg\_bal(selfMsg) = b$

then

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

Event $recvRspWDFail \mathrel{\widehat{=}}$

refines $recvRspWDFail$

any

$selfMsg$ contextual instance of refined class msg

$selfATM$

c

b

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

recvRspWDFail.Guard1 : $selfMsg \in RSP\_WDFAIL\_MSG$

recvRspWDFail.Guard2 : $msg\_atm(selfMsg) = selfATM$

recvRspWDFail.Guard3 : $msg\_card(selfMsg) = c$

recvRspWDFail.Guard4 : $msg\_bal(selfMsg) = b$

then

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

Event *recvRspCB* $\widehat{=}$

refines *recvRspCB*

any

selfMsg contextual instance of refined class msg

selfATM

c

b

where

selfMsg.type : $selfMsg \in msg$

selfATM.type : $selfATM \in ATM$

c.type : $c \in ValidCard$

b.type : $b \in \mathbb{N}$

recvRspCB.Guard1 : $selfMsg \in RSP\_CB\_MSG$

recvRspCB.Guard2 : $msg\_atm(selfMsg) = selfATM$

recvRspCB.Guard3 : $msg\_card(selfMsg) = c$

recvRspCB.Guard4 : $msg\_bal(selfMsg) = b$

then

msg_destructor : $msg := msg \setminus \{selfMsg\}$

end

END

Bibliography

[1] OMG: UML 2.1.2 Superstructure Specification. (2007). http://www.omg.org/cgi-bin/docs/formal/2007-11-02.pdf. Date Last Accessed: 23/2/2010.

[2] Object Management Group (2007). *Introduction to OMG's Unified Modelling Language (UML)* . Online. http://www.omg.org/gettingstarted/what_is_uml.htm. Date Last Accessed: 23/2/2010.

[3] Mammar A. and Laleau R. A Formal Approach Based on UML and B for the Specification and Development of Database Application. *Autom Softw Eng*, 13(4):497–528, 2006.

[4] Butler M. Hallerstede S. Abrial, J. R. and L Voisin. An Open Extensible Tool Environment for Event-B. In *Proceedings of ICFEM*, pages 473–480, 2006.

[5] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[6] J. R. Abrial. *Modelling in Event-B: System and Software Engineering*. To be published by Cambridge University Press, 2009.

[7] J. R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Journal Fundamentae Informatica. volume 77, pp. 1-28., IOS Press.*, 2007.

[8] Jean-Raymond Abrial. Formal Methods in Industry: Achievements, Problems, Future. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 761–768, New York, NY, USA, 2006. ACM.

[9] Alexander Knapp, Stephan Merz and Martin Wirsing. Refining Mobile UML State Machines. In *AMAST*, pages 274–288, 2004.

[10] Kalliopi Androutsopoulos. The Reactive System Development Support Tool. Technical report, 1999.

[11] Axel van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE*, page 249, 2001.

[12] R. Back and R. Kurki-Suonio. Decentralization of Process Nets With Centralized Control. In *Proceedings of ACM SIGACT-SIGOPS Symp on Distributed Computing*, pages 131–142, 1989.

[13] R. J. R. Back and K. Sere. Superposition Refinement of Reactive Systems. *Formal Aspects of Computing*, 3:324–346, 1995.

[14] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK*, pages 334–354, 2005.

[15] E. Ball. *An Incremental Development Process for the Development of Multi-agent Systems in Event-B.* PhD Thesis, University of Southampton, August 2008. http://eprints.ecs.soton.ac.uk/16575/.

[16] E. Ball and M. Butler. Patterns for Effective Modelling, Refinement and Verification of Client-Server Models in Event-B, 2007 (Private Communication with E. Ball).

[17] Wim Bast. The Essence of Model Driven Architecture. Online. http://www.jaxmagazine.com/itr/online_artikel/psecom,id,548,nodeid,147.html. Date Last Accessed: 23/2/2010.

[18] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Structuring and Refinement of Class Diagrams. *Hawaii International Conference on System Sciences*, 6:6018, 1999.

[19] Jonathan P. Bowen. Comp.specification.z and Z FORUM Frequently Asked Questions. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, volume 967 of *LNCS*, pages 561–569. Springer-Verlag, September 1995.

[20] Jonathan P. Bowen. *Formal Specification and Documentation Using Z: A Case Study Approach.* International Thomson Publishing, 1996.

[21] Alan Brown. An Introduction to Model Driven Architecture. Online. http://www.ibm.com/developerworks/rational/library/3100.html., 17 February 2004. Date Last Accessed: 23/2/2010.

[22] M. Butler and S. Hallerstede. The Rodin Formal Modelling Tool. In *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London*, 2007.

[23] M. J. Butler. An Approach to the Design of Distributed Systems with B AMN. *In: Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, 1997.

[24] M. J. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods IFM2009, Springer, LNCS 5423*, 2009.

[25] M. J. Butler. Incremental Design of Distributed Systems with Event-B. In: Engineering Methods and Tools for Software Safety and Security - Marktoberdorf Summer School 2008, pp. 131-160, IOS Press., 2009.

[26] Michael Butler and Divakar Yadav. An Incremental Development of the Mondex System in Event-B. *Formal Aspects of Computing*, 20(1):61–77, January 2008.

[27] Michael J. Butler. On the Verified-by-Construction Approach. Facs Facts Newsletter Issue 2006-1, March 2006. pg 6-10.

[28] Dominique Cansell and Dominique Méry. Tutorial on the Event-based B Method, 2006. IFIP FORTE 2006. Paris.

[29] Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk Interface Specification Language. *ACM Transaction Software Engineering Methodology*, 3(3):221–153, 1994.

[30] Claudia Pons. Heuristics on the Definition of UML Refinement Patterns. In *SOFSEM*, pages 461–470, 2006.

[31] Jim Conallen. Modelling Web Application Architectures with UML. *Commun. ACM*, 42(10):63–70, 1999.

[32] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, 2nd edn.* Cambridge University Press, Cambridge, 2002.

[33] E. Roberts, J. Villani, M. Osman, D. Godso, B. King, M. Ricciardi. Aviation System Analysis Capability Executive Assistant Design. Technical Report NASA/CR-1998-207679, May 1998.

[34] N. Evans and M. Butler. A Proposal For Records in Event-B. *In T. Nipkow an J. Misra, editors Formal Methods 2006*, 4085:221–235, 2006.

[35] Clemens Fischer. CSP-OZ: A Combination of Object-Z and CSP. Technical Report. University of Oldenburg, Germany, April, 30 1997.

[36] The Eclipse Foundation. Eclipse Modelling Framework. Online.http://www.eclipse.org/emf/.Date Last Accessed: 26/4/2010.

[37] The Eclipse Foundation. Graphical Modelling Framework. Online. http://www.eclipse.org/gmf/. Date Last Accessed: 26/4/2010.

[38] Peter Frey. Combining UML Use Cases and VDM-SL. Paper for the Seminar in Software Technology at the Institute for Software Technology (IST), Graz University of Technology, Austria, 2000.

[39] G Booch, J Rumbaugh and I Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[40] Albert L. Baker Gary T. Leavens and Clyde Ruby. JML: A Notation for Detailed Design. *In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors)*, Behavioral Specifications of Businesses and Systems:175–188, 1999.

[41] Zhiming Liu Gary T. Leavens and He Jifeng (eds). JML's Rich, Inherited Specifications for Behavioral Subtypes. *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, Volume 4260 of Lecture Notes in Computer Science:2–34, 2006.

[42] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Mller, and Joseph Kiniry. JML Reference Manual (DRAFT), February 2007.

[43] J.V. Guttag, K.D. Jones A. Modet J.J. Horning, with S.J. Garland, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. January 19, 1993.

[44] J. V. Guttage and J. J. Horning. Introduction to LCL, A Larch/C Interface Language. Digital Equipment Corporation, July 24, 1991.

[45] Anthony Hall. Seven Myths of Formal Methods. In *IEEE Software,7(5):*, pages 11–19, September 1990.

[46] Eckhardt Holz. Application of UML Within the Scope of New Telecommunication Architectures. In Martin Schader and Axel Korthaus, editors, *The Unified Modelling Language – Technical Aspects and Applications*, pages 207–219. Physica-Verlag, Heidelberg, 1998.

[47] Hongjiang Gao, Zheng Qin, Liping Shao and Xingchen Heng. Specifying and Verifying Cases Retrieval System Combining Event B and Spin. In *ICSC '07: Proceedings of the International Conference on Semantic Computing*, pages 53–60, Washington, DC, USA, 2007. IEEE Computer Society.

[48] J. J. Horning. Combining Algebraic and Predicative Specifications in Larch. Springer Berlin/Heidelberg, 1985.

[49] Akram Idani, Yves Ledru, and Didier Bert. Derivation of UML Class Diagrams as Static Views of Formal B Developments. In *ICFEM*, pages 37–51, 2005.

[50] Information Technology Programming Languages VDM-SL. First Committee Draft Standard: CD 13817-1, Nov 1993. ISO/IEC JTC1/SC22/WG19 N-20.

[51] Jens Schönborn and Marcel Kyas. Refinement Patterns for Hierarchical UML State Machines. In *FSEN*, pages 371–386, 2009.

[52] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[53] Kevin D. Jones. LM3: A Larch Interface Lanugage for Modula-3: A Definition and Introduction, Version 1.0. Technical Report 72. DEC SRC, Jun 1991.

[54] D. Clark K. Lano and K. Androutsopoulos. UML to B: Formal Verification of Object Oriented Models. In *Integrated Formal Methods: Proceeding of the 4th international conference*, pages 761–768. Springer, 2004.

[55] Hans Kristian Agerlund Lintrup Kenneth Lausdahl and Peter Gorm Larsen. Connecting UML and VDM++ with Open Tool Support. In *FM*, pages 563–578, 2009.

[56] R. Laleau and A. Mammar. An Automatic Generation of B Specifications from Well-defined UML Notations for Database Applications. In *International Symposium on Programming Systems*, 2001.

[57] W. Längst, A. Lapp, K. Knorr, H. P. Schneider, J. Schirmer, D. Kraft, and W. Kiencke. CARTRONIC-UML Models: Basis for Partially Automated Risk Analysis in Early Development Phases. In Jan Jürjens, María Victoria Cengarle, Eduardo B. Fernandez, Bernhard Rumpe, and Robert Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 3–18. Technische Universität München, Institut für Informatik, 2002.

[58] Larsen P. G., Battle N., Ferreira M., Fitzgerald J., Lausdahl, K. and Verhoef M. The Overture Initiative Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, 2010.

[59] Gary T. Leavens. An Overview of Larch/C++: Behavioral Specifications for C++ Modules. Technical Report 96-01b. Iowa State University, February 1996.

[60] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for Enhanced Languages and Methods to Aid Verification. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236, New York, NY, USA, 2006. ACM.

[61] H. Ledang and J. Souquiéres. Contributions for modelling uml state-charts in b. In *In: Proc. IFM, LNCS 2335*, pages 109–127, 2002.

[62] H. Ledang and J. Souquieres. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. In *Proceedings of APSEC 2002*, Gold Coast, Queensland, Australia, December 4-6 2002.

[63] H. Ledang and J. Souquieres. Integration of UML Views using B Notation. In *Proceedings of WITUML02*, Malaga, Spain, June 11 2002.

[64] D. Lightfoot. *Formal Specification Using Z: Second Edition.* Palgrave, 2001.

[65] Jackson M. *System Development.* Prentice Hall, 1983.

[66] M. Moller, E. Olderog, H. Rasch and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In *Integrated Formal Methods: Proceeding of the 4th international conference*, pages 267–286. Springer, 2004.

[67] M. Plaska, M. Walden and C. Snook. Documenting the Progress of the System Development. In *Methods, Models and Tools for Fault Tolerance*, pages 251–274, 2009.

[68] Nenad Medvidovic. *Introduction to Formal Methods.* Lecture Notes of Course CS 599: Formal Methods in Software Architectures. Computer Science Department, Viterbi School of Engineering, University of Southern California, Nov 2000. http://sunset.usc.edu/classes/cs599_2000/. Date Last Accessed: 23/2/2010.

[69] Abrial J.-R. Voisin L Metayer, C. Event-B Language. Technical Report Deliverable 3.2. EU Project IST-511599 - RODIN, May 2005.

[70] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. OMG, Jun 2003.

[71] Michael Möller. Specifying and Checking Java using CSP. In Workshop on Formal Techniques for Java-like Programs - FTfJP'2002, Jun 2002. citeseer.ist.psu.edu/530828.html.

[72] Anamaria Martins Moreira and David Dharbe. Tutorial: Software Engineering with the B Method. 8th Brazilian Symposium on Formal Methods, 2005.

[73] Mara Nikolaidou and Dimosthenis Anagnostopoulos. Enterprise Information Systems Configuration: Emphasizing the Symbiotic Relationship between Applications and the Underlying Network. *iceccs*, 00:47–56, 2004.

[74] Nuno Amálio, Fiona Polack and Susan Stepney. UML + Z: Augmenting UML with Z. In Marc Frappier, Henri Habrias, eds. Software Specification Methods : an Overview Using a Case Study, new edn, Hermes Science Publishing, 2006.

[75] OMG. *Object Constraint Language Specification, version 2.0.* Object Modelling Group, May 2006. http://www.omg.org/cgi-bin/doc?formal/2006-05-01. Date Last Accessed: 26/4/2010.

[76] Object Management Group (OMG). Common Warehouse Metamodel. Online. http://www.omg.org/technology/documents/formal/cwm.htm. Date Last Accessed: 26/4/2010.

[77] Object Management Group (OMG). Meta-Object Facility. Online. http://www.omg.org/mof/. Date Last Accessed: 26/4/2010.

[78] Patrice Chalin, Perry R. James, and George Karabotsos. The Architecture of JML4, a Proposed Integrated Verification Environment for JML. Dependable Software Research Group, Concordia University, ENCS-CSE-TR 2007-006, May 2007.

[79] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor: A Successful Application of B in a Large Project. . In *In Proceedings of FM99, Toulouse, France*, page 712, 1999.

[80] D. Pilone and N. Pitman. *UML 2.0 In A Nutshell*. O' Reilly Media, Inc, Sebastopol, United States of America, 2005.

[81] R. Razali, C. Snook and M. Poppleton. Comprehensibility of UML-based Formal Model: A Series of Controlled Experiments. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 25–30, New York, NY, USA, 2007. ACM.

[82] N. Ramsey. Developing Formally Verified Ada Programs. *SIGSOFT Softw. Eng. Notes*, 14(3):257–265, 1989.

[83] Said, M. Y., Butler, M. and Snook, C. Language and Tool Support for Class and State Machine Refinement in UML-B. In *In: FM2009 - 16th International Symposium on Formal Methods, 2-6th November 2009, Eindhoven*, pages 579–595, 2009.

[84] S. Schneider. *The B-method an Introduction*. Palgrave Macmillan, 2001.

[85] E. Sekerinski. Graphical Design of Reactive System. LNCS 1393, Springer-Verlag, pp. 182-197. In *In: D. Bert, Editor, B'98: REcent Advances in the Development and Use of the B Method*, 1998.

[86] Bran Selic and J. Rumbaugh. Using UML for Modeling Complex Real-time Systems. In *Technical Report, Objectime Limited and Rational Software Corporation*, 1998.

[87] D. Sheppard. *An Introduction to Formal Specification With Z and VDM*. McGraw-Hill, Berkshire, England, 1995.

[88] J. Siegel and the OMG Staff Strategy Group. Developing in OMG's Model Driven Architechture. Object Management Group White Paper, November 2001. Revision 2.6.

[89] R. Silva and M. Butler. Supporting Reuse Mechanisms for Developments in Event-B: Composition. In: http://eprints.soton.ac.uk/69662/, 2009.

[90] Silva, R. and Butler, M. Parallel Composition Using Event-B. http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B, July 2009.

[91] Anthony J. H. Simons. A Theory of Regression Testing for Behaviourally Compatible Object Ypes: Research Articles. *Softw. Test. Verif. Reliab.*, 16(3):133–156, 2006.

[92] Smith G. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[93] C. Snook and Walden. M. Refinement of Statemachines Using Event B Semantics, B2007. Formal Semantic and Development in B. LNCS volume 4355/2006. pp. 171-185, 2006.

[94] Colin Snook and Michael Butler. U2B - A Tool For Translating UML-B Models Into B. *UML-B Specification for Proven Embedded Systems Design*, April 2004.

[95] Colin Snook and Michael Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, January 2006.

[96] Colin Snook and Michael Butler. UML-B and Event-B: an integration of languages and tools. In *The IASTED International Conference on Software Engineering - SE2008*, February 2008.

[97] Colin Snook, Michael Butler, and Ian Oliver. The UML-B Profile for formal systems modelling in UML. *UML-B Specification for Proven Embedded Systems Design*, April 2004.

[98] J. M. Spivey. The Z notation: A Reference Manual. Prentice Hall, 1989.

[99] Stephan Merz, Martin Wirsing and Júlia Zappe. A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems. In *FASE*, pages 87–101, 2003.

[100] Thai Son Hoang, Hironobu Kuruma, David A. Basin and Jean-Raymond Abrial. Developing Topology Discovery in Event-B. *Sci. Comput. Program.*, 74(11-12):879–899, 2009.

[101] F. Truyen. The Fast Guide to Model Driven Architecture: The Basics of Model Driven Architecture. Cephas Consulting Corp, January 2006.

[102] Jeanette M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8, 10–23, 1990.

[103] Jeannette M. Wing, Eugene J. Rollins, and Amy Moormann Zaremski. Thoughts on a Larch/ML and a New Application for LP, 1992.

[104] X. Leroy, D. Doligez, J. Garrigue, D. Remy and J. Vouillon. The Objective Caml System, Documentation and User's Manual. INRIA, France, 2008.