ROYAUME DU MAROC
Université Mohammed V - RABAT -
جامعة محمد الخامس – الرباط –
Faculté des sciences
كلية العلوم
CENTRE D'ETUDES DOCTORALES - SCIENCES ET TECHNOLOGIES

N° d'ordre : 3138

THESE

En vue de l'obtention du : ***DOCTORAT***

Structure de Recherche : Laboratoire de Matière Condensée et Sciences Interdisciplinaires (LaMCScI)
Discipline : Informatique
Spécialité : Génie Logiciel

Présentée et soutenue le 13/10/2018 par :

Sanae EL MIMOUNI

Towards a Formal Approach of Communication Protocols through Refinement and Proofs

JURY

Saïd SLAOUI	PES, Faculté des Sciences, Rabat	Président
Mohamed BOUHDADI	PES, Faculté des Sciences, Rabat	Directeur de Thèse
Faissal OUARDI	PH, Faculté des Sciences, Rabat	Rapporteur
Jalal LAASSIRI	PH, Faculté des Sciences, Kénitra	Rapporteur
Mohammed RZIZA	PES, Faculté des Sciences, Rabat	Examinateur

Année Universitaire : 2018/2019

Avant propos

Les travaux presént́és dans cette thèse,ont été effectués au sein au Laboratoire de Matière Condensée et Sciences Interdisciplinaires (LaMCScI) à la Faculté des Sciences de Rabat. Ce travail a été effectué sous la direction du Professeur BOUHDADI Mohamed.

J'exprime mes profonds remerciements à mon directeur de thèse, le professeur BOUHDADI Mohamed pour toute son aide. Je suis ravie d'avoir travaillé en sa compagnie car outre son appui scientifique, il a toujours été là pour me soutenir et me conseiller au cours de l'élaboration de cette thèse. Enfin, j'ai été extrêmement sensible à ses qualités humaines d'écoute et de compréhension tout au long de ce travail doctoral.

Je suis très honorée à remercier de la présence à mon jury de thèse et je tiens à remercier :

Monsieur SLAOUI Saïd Professeur d'enseignement supérieur à la faculté des sciences de Rabat, pour l'honneur qu'il m'a fait en acceptant d'être le président du jury de ma thèse. Je tiens à l'assurer de ma profonde reconnaissance pour l'intérêt qu'il porte à ce travail.

Je remercie Monsieur RZIZA Mohammed Professeur d'enseignement supérieur à la faculté des sciences de Rabat,je suis très ravie de votre présence au sein du jury. Je vous remercie pour les commentaires que vous tenterez d'avancer et qui me seront bénéfiques. Je vous exprime ma profonde reconnaissance.

Monsieur OUARDI Faissal Professeur habilité à la faculté des sciences de Rabat, pour avoir accepté d'être rapporteur de mes travaux et pour ses observations qui m'ont permis d'améliorer la qualité de ce mémoire. Je tiens à lui exprimer mes remerciements pour l'honneur qu'il me fait en participant à ce jury.

Je tiens également à remercier Monsieur LAASSIRI Jalal, Professeur habilité à la Faculté des Sciences de Kenitra, pour l'honneur qu'il m'a fait pour sa participation à mon jury de thèse en qualité de rapporteur de mon travail, pour le temps consacré à la lecture de cette thèse, et pour les suggestions et les remarques judicieuses qu'il m'a indiquées.

En premier lieu, je remercie mes parents qui ont su croire en moi et qui m'ont apporté toute leur aide quand j'en ai eu besoin. Ce mémoire leur est dédié à 200%. Je tiens à remercier trés chaleureusement mon mari qui m'a soutenue tout au long de ces années. Je remercie également toute ma famille qui a contribué de près ou de loin à ce que je suis devenue.

Acknowledgements

After an intensive period, today is the day: writing this note of thanks is the finishing touch on my dissertation. It has been a period of intense learning for me, not only in the scientific arena, but also on a personal level. Writing this dissertation has had a big impact on me. I would like to reflect on the people who have supported and helped me so much throughout this period.

Foremost, I would like to express my sincere gratitude to my advisor Prof. BOUHDADI Mohamed for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

I would like to extend my gratitude to the Professor SLAOUI Saïd a professor at faculty of sciences of Rabat. For accepting to be my Jury President, it's such an honor to have a professor like you in my committee. I feel that your knowledge and insights would be extremely valuable and would greatly enrich my work.

I would like to express my special appreciation and thanks to Professor RZIZA Mohammed a professor at faculty of sciences of Rabat. For serving as a member of my committee even at hardship.

In the same way, I am particularly grateful to Professor OUARDI Faissal a professor at faculty of sciences of Rabat, for the time he spent reading my thesis, and for his interesting comments. His participation to the jury of this thesis and his review of the text are highly appreciated.

I would also like to warmly thank Professor LAASSIRI Jalal a professor at faculty of sciences of Kenitra for the help, the insights and the comments he gave me without restraint. I am glad he accepted to be a member of the jury of this thesis. I appreciate particularly the precise and efficient reviewing of the complete thesis he performed for the private defense.

This dissertation is dedicated to my mom and dad, for their relentless efforts through the years and continents. They have dedicated their life to provide the best of everything to my brother and me. I owe them my education and my love for what I do today. I would also like to deeply thank my husband, Adil, for the great help, support, and encouragement he gave me and for always standing by me.

My friends that helped and made my days so much better, for their friendship and for being the way they are: Sarah Kabbou, Meryem Ouache, Meryem El Bouzidi and Safae Chafi you all are stars! Also I would like to thank my dear friend Hichame Mouline for the support despite the distance.

Not least of all, I owe so much to my whole family for their undying support, their unwavering belief that I can achieve so much. Unfortunately, I cannot thank everyone by name because it would take a lifetime but, I just want you all to know that you count so much. Had it not been for all your prayers and benedictions; were it not for your sincere love and help, I would never have completed this thesis. So thank you all.

Sanae. El Mimouni

Résumé

Différentes approches ont été utilisées pour la spécification formelle et la vérification des protocoles de communication. L'approche considérée dans ce travail se concentre sur la vérification déductive dans laquelle la correction d'un modèle par déduction se ramène à la preuve de formules mathématiques appelées obligations de preuve. Dans notre approche, nous avons choisi la méthode Event-B. Event-B est un langage de spécification formel, elle offre un potentiel élevé en termes de correction grâce à son mécanisme de raffinement bien connu, de ses obligations de preuve bien définies et de la plate-forme RODIN. Notre objectif été de construire un modèle avec une formulation claire et précise des propriétés liées aux protocoles de communication et de décharger toutes les obligations de preuve. Pour cela, nous partons d'exigences abstraites, nous les raffinons progressivement pour obtenir une description concrète et détaillée du modèle et nous vérifions chaque niveau de raffinement par rapport à la spécification construite dans le raffinement précédent. Ce qui va nous permettre de vérifier la correction, c'est-à-dire la préservation des propriétés de l'étape précédente, par rapport aux raffinements successifs. Ceci est atteint par la création et la démonstration de logique.

Mots clefs : Méthodes formelles, Spécification formelle, Event-B, Raffinement, protocol de communication, Rodin platform

Abstract

Different approaches have been used for the formal specification and verification of communication protocols. The approach considered in this work focuses on the deductive verification in which the correction of a model by deduction is reduced to the proof of mathematical formulas called obligations of proof. In our approach, we chose the Event-B method. Event-B is a formal specification language, offering a high potential for correction through its well-known refinement mechanism, well-defined proof obligations, and the RODIN platform. Our goal has been to build a model with a clear and precise formulation of properties related to communication protocols and to discharge all the proof obligations. For that, we start from abstract requirements, we gradually refine them to obtain a concrete and detailed description of the model and we check each level of refinement with respect to the specification built in the previous refinement. This will allow us to verify the correction, that is to say, the preservation of the properties of the previous step, compared to successive refinements. This is achieved by creating and demonstrating logic.

Key words: Formal Method, Formal Specification, Event-B, Refinement, Communication protocol, RODIN Platform

Résumé Détaillé

Les méthodes formelles sont des techniques mathématiques qui fournissent une base rigoureuse pour le développement de logiciels : l'application des méthodes formelles permet d'obtenir une correction et une fiabilité prouvables dans les différentes étapes de la conception et de la mise en œuvre du système. Les méthodes formelles se sont révélées prometteuses pour le développement de méthodes de vérification et de test automatisées et génériques des protocoles.
Appliquer une méthode formelle revient d'abord à construire un modèle mathématique correspondant aux spécifications informelles d'un système. Pour cela, on utilise un formalisme mathématique de haut niveau. Et une structure à la syntaxe et à la sémantique bien définies. Elle doit permettre de décrire le comportement d'un composant en décrivant ses propriétés importantes en basant sur des expressions logiques. Cette thèse propose une approche formelle pour tester les propriétés liées aux protocoles de communication. Les protocoles de communication définissent l'ensemble des règles nécessaires qui précisent les modalités de fonctionnement d'une communication entre des entités communicantes. De nos jours, les systèmes de réseau et les systèmes distribués qui sont construits autour de protocoles de communication, sont largement utilisés. Il devient donc de plus en plus important que les protocoles de communication soient formellement spécifiés et vérifiés.
Nous nous intéressons dans cette thèse à la vérification déductive (preuve de théorèmes), dans laquelle la correction d'un code par déduction se ramène à la preuve de formules mathématiques appelées obligations de preuve. La méthode Event-B et son assistant de preuve Rodin forment l'environnement formel que nous avons choisi pour représenter et analyser nos spécifications. La principale raison de choisir Event-B comme langage de modélisation est le raffinement. La méthode du raffinement en Event-B suit le "paradigme du parachute". Selon ce paradigme, le système est spécifié comme si une personne le décrivait pendant une descente en parachute. Le niveau abstrait du système décrit un système très simple et les détails sont ajoutés au fur et à mesure des raffinements.
Nous nous limitons dans ce travail à une collection de spécifications concernant un domaine d'application à savoir le champ des protocoles de communication. L'objectif vise à la modélisation formelle et l'analyse de certains protocoles de communication avec le langage Event-B. Les protocoles décrits dans notre thèse couvrent un large éventail, tels qu'un protocole de Couche de liaison de données, un protocole de commerce électronique et un protocole cryptographique. Afin de spécifier nos protocoles, nous décrivons ce que le protocole devrait faire, ce que le protocole ne devrait pas faire et comment le protocole devrait réagir.
Sliding window protocole est un protocole classique d'envoi réception utilisé dans le groupe TCP/IP, c'est un protocole Automatic Repeat Request qui assure un transfert de données

fiable sur un canal avec perte. Il existe trois types de ce protocole. Il s'agit de « Envoyer et attendre », « Stop-and-Go », « send-and-wait» « Go-back-n » ou « Fenêtre glissante », « Rejet sélectif ». Les trois varient entre eux en termes d'efficacité, de complexité et d'exigences de tampons. Notre spécification se concentre sur la version Go-Back-N du protocole. Malgré l'importance pratique du protocole de la fenêtre glissante (Sliding Window Protocol), sa vérification formelle a été relativement limitée. Stenning [1] n'a donné qu'une preuve manuelle informelle de ce protocole. Une preuve manuelle semi-formelle est également présentée dans [2], et certaines versions du protocole ont été vérifiées par modèle (Model-checking) pour déterminer les petites valeurs de paramètre dans [3, 4]. Il est nécessaire de présenter un modèle Event-B pour un tel protocole.
Un autre protocole de communication est le protocole de commerce électronique, c'est un protocole de sécurité qui permet aux clients et aux commerçants d'exercer leurs activités de manière électronique via Internet. Nous avons choisi de modéliser le protocole NetBill [5]. Le protocole NetBill est un protocole de commerce électronique optimisé pour la vente et la livraison de produits d'information à bas prix, tels que des logiciels ou des articles de journaux, sur Internet. Il est difficile de vérifier à la main l'exactitude d'une conception basée sur ce protocole. Il est donc très approprié d'appliquer des méthodes et techniques formelles à la modélisation et à la conception d'un tel protocole afin d'obtenir une assurance élevée quant à leur exactitude.
Enfin, nous présentons une approche pour modéliser un protocole cryptographique. Le protocole Tatebayashi, Matsuzaki et Newman (TMN) [6], largement connus sous le nom de «TMN», a été l'un des premiers protocoles d'établissement de clés conçus pour être utilisés dans un environnement mobile. Bien qu'il existe une grande variété de modèles formels qui ont été appliqués pour modéliser et analyser le protocole TMN, tels que les réseaux de Petri colorés [7], mais aucun a été créé par la méthode Event-B.
L'utilisation des méthodes formelles telles que la méthode Event-B permet d'obtenir un très bon niveau de confiance dans le développement. La méthode Event-B est un langage de spécification formel, elle offre un potentiel élevé en termes de correction grâce à son mécanisme de raffinement bien connu, de ses obligations de preuve bien définies et de la plate-forme RODIN. La méthode Event-B est à la fois un langage et une méthode de spécification et de preuve de logiciels informatiques. Elle a été récemment proposée par Jean-Raymond Abrial comme une évolution de la méthode B classique. Le processus de développement en Event-B suppose que chaque spécification est associée à une preuve de correction basée elle même sur le fondement mathématique du langage. Toutefois, la méthode Event-B est directement liée à l'approche de conception de programmes : "correct-par-construction". Cette approche consiste à débuter le développement par un modèle abstrait du système et rajouter progressivement des détails pour générer à la fin un modèle concret très proche de l'implémentation. Formellement, cette construction incrémentale du système est guidée par la technique de raffinement. Le processus de développement engendre plusieurs obligations de preuve qui garantissent sa correction. Ces obligations sont prouvées par l'assistant de preuve Rodin.
Un modèle Event-B est considéré comme correct, lorsque chaque machine, ainsi que le processus de raffinement, sont prouvés par des théorèmes adéquats nommés obligations de preuve (OP) et que chaque événement est réalisable. Des démonstrations mathématiques sont incorporées à Event-B pour vérifier la correction des étapes de raffinement. La gestion des obligations de preuve est une tâche technique soutenue par la plat-forme RODIN, qui fournit un environnement pour le développement correct par les modèles de construction pour les

systèmes logiciels.
L'objectif de cette thèse est de construire un modèle avec une formulation claire et précise des propriétés liées aux protocoles de communication et de décharger toutes les obligations de preuve. Pour cela, nous partons d'exigences abstraites, nous les raffinons progressivement pour obtenir une description concrète et détaillée du modèle et nous vérifions chaque niveau de raffinement par rapport à la spécification construite dans le raffinement précédent. Ce qui va nous permettre de vérifier la correction, c'est-à-dire la préservation des propriétés de l'étape précédente, par rapport aux raffinements successifs. Ceci est atteint par la création et la démonstration de logique. Pour satisfaire nos objectifs, nous avons faix un choix attentif des invariants et des théorèmes de machine qui nous a facilité l'effort de preuve. Un objectif majeur de notre travail a été d'explorer l'utilisation de la méthode Event-B pour spécifier formellement un protocole de communication. Deux approches majeures de la spécification générale ont été identifiées et appliquées à l'ensemble de protocoles de communication que nous avons choisi : (1) un modèle abstrait définissant les opérations qui pouvant être provoquées et (2) un modèle concret où les services sont un processus actif avec échange de messages.
Le but de cette thèse est de fournir une collection de descriptions de protocoles qui montre comment utiliser des techniques de spécification formelles telles que la méthode Event-B dans le domaine des protocoles de communication. De plus, nous souhaitons donner un aperçu de la conception et du fonctionnement des protocoles de communication. Les spécifications de cette thèse ont un niveau d'abstraction approprié pour une compréhension claire des protocoles sans avoir à traiter des détails de l'implémentation.

Contents

Acknowledgements (Français/English) i

Abstract (Français/English) iv

Résumé Détaillé v

List of Figures xi

List of Tables xiv

Acronyms xv

1 Introduction 1
- 1.1 The Need of Formal Approach . . . 2
- 1.2 Motivation . . . 4
- 1.3 Contributions . . . 7
- 1.4 Thesis Organization . . . 8

2 Formal Methods 9
- 2.1 Definition . . . 9
- 2.2 Why Use Formal Methods? . . . 11
  - 2.2.1 Application Domains . . . 13
- 2.3 Classification . . . 14
  - 2.3.1 Property-Based Methods . . . 14
  - 2.3.2 Model-Based Methods . . . 14
- 2.4 Some Formal Methods . . . 15
  - 2.4.1 Z Notation . . . 15
  - 2.4.2 VDM . . . 17
  - 2.4.3 B Method . . . 18
  - 2.4.4 CSP . . . 20
- 2.5 Other Formalisms . . . 21
  - 2.5.1 Action Systems . . . 21
  - 2.5.2 Process Algebra . . . 22
  - 2.5.3 Temporal Logic . . . 22
- 2.6 Refinement . . . 23
- 2.7 Summary . . . 24

3 Event-B: A Modeling Language **25**
3.1 Fundamentals of Event-B . . . 25
3.1.1 Hoare Logic . . . 25
3.1.2 Dijkstra's Weakest Precondition . . . 27
3.1.3 Refinement Calculus . . . 28
3.2 Mathematical Notation . . . 29
3.2.1 Set Theory . . . 29
3.2.2 Sequent Calculus . . . 30
3.2.3 Inference Rules . . . 31
3.3 Event-B Method . . . 34
3.3.1 The Correct-by-Construction Approach . . . 34
3.3.2 Event-B Model . . . 35
3.3.3 Contexts . . . 35
3.3.4 Machines . . . 36
3.3.5 Events . . . 37
3.4 Refinement in Event-B . . . 41
3.5 Decomposition . . . 43
3.5.1 Shared Variable Decomposition . . . 43
3.5.2 Shared Event Decomposition . . . 44
3.5.3 Shared Event Decomposition with Shared Parameters . . . 45
3.6 Generic Instantiation . . . 46
3.7 Proof Obligation Rules . . . 46
3.7.1 Invariant Preservation Rule: INV . . . 47
3.7.2 Feasibility Rule: FIS . . . 47
3.7.3 Guard Strengthening Rule: GRD . . . 48
3.7.4 Simulation Rule: SIM . . . 48
3.7.5 The Numeric Variant Rule: NAT . . . 48
3.7.6 The Variant Rule: VAR . . . 49
3.7.7 Well-Definedness Rule . . . 49
3.8 RODIN Platform . . . 49
3.9 Event-B and Other Formal Methods . . . 52
3.10 Summary . . . 53

4 Communication Protocols Overview **54**
4.1 OSI Reference Model . . . 54
4.1.1 Formal Methods for OSI Layer Protocols . . . 56
4.2 Basics of Communication Protocol . . . 56
4.2.1 Communication Problems . . . 57
4.2.2 Traditional Communication . . . 58
4.2.3 A Simple Message Exchange Protocol . . . 58
4.3 Protocol Development . . . 58
4.3.1 Protocol Engineering . . . 61
4.3.2 Protocol Specification . . . 61
4.3.3 Protocol Verification . . . 62
4.3.4 Protocol Testing . . . 62
4.4 Summary . . . 62

5 The Sliding Window Protocol 63
5.1 Introduction 63
5.2 Informal Description 64
5.2.1 Messages and Channels 64
5.2.2 Sender Sliding Window 65
5.2.3 Receiver Sliding Window 65
5.2.4 Sequence Numbers 66
5.2.5 Timeout and Retransmission 67
5.3 Related Work 68
5.4 Specifying SWP Using Event-B 69
5.4.1 Refinement Strategy 69
5.4.2 Initial Model 69
5.4.3 First Refinement 70
5.4.4 Second Refinement 72
5.4.5 Third Refinement 72
5.4.6 Fourth Refinement 76
5.4.7 Fifth Refinement 81
5.4.8 Sixth Refinement 83
5.5 Summary 85

6 The NetBill Electronic Commerce Protocol 86
6.1 Introduction 86
6.2 NetBill Protocol 87
6.3 Specifying NetBill Protocol Using Event-B 89
6.3.1 Refinement Strategy 89
6.3.2 Initial Model 90
6.3.3 First Refinement 91
6.3.4 Second Refinement 94
6.4 Summary 96

7 The Tatebayashi, Matsuzaki and Newman (TMN) Protocol 98
7.1 Introduction 98
7.2 TMN Protocol 99
7.3 Refinement Strategy 100
7.3.1 Initial Model 100
7.3.2 First Refinement 102
7.4 A Modified Version of TMN Protocol 105
7.5 Summary 106

8 Conclusion and Future Work 107
8.1 Summary 107
8.2 Future Work 109

A Event-B Model of Sliding Window Protocol 111
A.1 Abstract Model 111
A.2 First Refinement 113
A.3 Second Refinement 114

A.4 Third Refinement 115
A.5 Fourth Refinement 116
A.6 Fifth Refinement 119
A.7 Sixth Refinement 121

B Event-B Model of NetBill Protocol **124**
B.1 Abstract Model 124
B.2 First Refinement 126
B.3 Second Refinement 127

C Event-B Model of TMN Protocol **129**
C.1 Abstract Specification 129
C.2 First Refinement 130

Bibliography **144**

List of Figures

2.1 The Model Checking Approach 11
2.2 Proof Process in Theorem-Proving Tools 12
2.3 Birthday Book Example 15
2.4 Init Birthday Book Example 15
2.5 Add Birthday Book Example 16
2.6 Find Birthday Book Example 16
2.7 Remind Birthday Book Example 16
2.8 Birthday Book Example in VDM 17
2.9 Add Birthday Operation in VDM 18
2.10 Find Birthday Operation in VDM 18
2.11 Remind Birthday Operation in VDM 18
2.12 Structure of a B Model 19
2.13 The Birthday Book Example in B 21
2.14 Balzer's Life Cycle 23
2.15 Stepwise Refinement of Specifications 24

3.1 A Proof Tree 31
3.2 Machine and Context 35
3.3 Machine and Context Relationships 36
3.4 Context Structure 37
3.5 Machine Structure 38
3.6 Shared-Variable Decomposition Style 44
3.7 Shared-Event Decomposition Style 44
3.8 An Event with Shared Parameter 45
3.9 Decomposition of an Event with Shared Parameter 45
3.10 Rodin Verification Process 50
3.11 Proving Perspective 51
3.12 Event-B Perspective 52

4.1 ISO Reference Model for Open Systems Interconnection 55
4.2 A Simple Example of Protocol 59
4.3 A Protocol Development Methodology 60
4.4 Conformance Testing, Protocol Description and Implementation 61

5.1 Sender Sliding Window 65
5.2 Receiver Sliding Window 65
5.3 Go-Back-N Normal Operation 66

5.4 Go-Back-N Lost Frame 67
5.5 Window size for Go-Back-N 68
5.6 The Channels 73

6.1 The NetBill protocol 88
6.2 Customer and Merchant View of a Transaction 89

7.1 Message Exchange in TMN Protocol 99

List of Tables

3.1 Binary Relation Operators 30
3.2 Function Operators 30
3.3 Before-After Predicate 39
3.4 Proof Obligations in Event-B 47

5.1 Window Widths of Sliding Window Protocols 64
5.2 Proof Statistics for the Sliding Window Protocol Development 84

6.1 Proof Statistics for the NetBill Protocol Development 96

7.1 Proof Statistics for the TMN Protocol Development 105
7.2 Proof Statistics for the Modified Version of TMN Protocol 106

Acronyms

ACK	Acknowledgement
ACL2	A Computational Logic for Applicative Common Lisp
ACP	Algebra for Communicating Processes
AMN	Abstract Machine Notation
API	Application Programming Interfaces
ARQ	Automatic Repeat Request
ASF	Algebraic Specification Formalism
CCR	Commitment, Concurrency and Recovery
CCS	Calculus of Communicating Systems
CICS	Customer Information Control System
COLD	Common Object-oriented Language for Design
CSP	Communicating Sequential Processes
DTP	Distributed Transaction Processing
EHDM	Enhanced Hierarchical Development Methodology
FIFO	First in, First out
FTP	File Transfer Protocol
GBN	Go-Back-N
HDLC	High-Level Data Link Control
HOL	Higher Order Logic
HTTP	HyperText Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
LOTOS	Language Of Temporal Ordering Specification
OSI	Open System Interconnection
POG	Proof Obligation Generator
POM	Proof Obligation Manager
PPP	Point to Point Protocol
ProCos	Provably Correct Systems
PROTEAN	PROTocol Emulation and ANalysis
PSF	Process Specification Formalism
PVS	Prototype Verification System
RAISE	Rigorous Approach to Industrial Software Engineering
RODIN	Rigorous Open Development Environment for Complex Systems
ROSE	Remote Operations Service Element
SC	Static Checker
SDL	Specification and Description Language

SET	Secure Electronic Transactions
SLIP	Serial Line Internet Protocol
SPX	Sequenced Packet Exchange
SRP	Selective Repeat Protocol
SWP	Sliding Window protocol
TCP	Transmission Control Protocol
TMN	Tatebayashi, Matsuzaki and Newman
TTCN	Testing and Test Control Notation
VDM	Vienna Development Method

1 Introduction

With the development of computer networks and distributed computer systems, the complexity and a variety of communication, protocols, and services have been greatly increased and so have the needs of formal techniques for specifying, validating, implementing and testing protocols. Communication protocols are complex software systems and main components of computer networks. They must be specified entirely and implemented correctly. The development of a protocol usually involves several annoying steps. Experience has proved that the use of informal techniques in these steps of protocol development generally produces systems with errors and unwanted behaviors. Formal methods allow the development of highly reliable and easily maintainable communication protocols.

Formal methods are mathematically-based techniques that provide a rigorous basis for software development: the application of Formal methods makes it possible to accomplish provable correctness and reliability in the various steps of system design and implementation. Formal methods have proven to be promising toward developing automated and generic protocol verification and testing methods. This thesis studies a based formal approach for testing associated properties for communication protocol. Communication protocols define the set of rules needed to exchange messages among communicating entities. Networked and distributed systems, built around communicating protocols, are widely used nowadays. So, it is becoming more significant that communication protocols be formally specified and verified.

The application of the formal methods is becoming increasingly important [8]. The reliable and a quick specification require an easy-to-use methods and tools. The formal methods allow a more systematic approach to protocol validation, implementation and testing. Compared with the traditional use of protocol specifications given in a natural language.

We present a formal modeling and analysis of certain communication protocol with the Event-B [9] language. In this thesis, we have used Event-B as proof-based development method which integrates formal proof techniques for writing specifications and building the model systematically using formal refinement, the key point is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete ones. This strategy eases the proof of the correctness of requirements because only a small number of proof obligations are generated at each step. An Event-B model is considered as correct, when each machine, as well as the process of

refinement, are proved by adequate theorems named proof obligations (PO) and that each event is feasible. Mathematical proofs are incorporated into Event-B to verify the correctness of refinement steps. The management of proof obligations is a technical task supported by Rigorous Open Development Environment for Complex Systems (RODIN), which provides an environment for developing correct by construction models for software-based systems. This incremental approach specifies the transitional rules of the protocol by identifying: the set of all possible states, such as idle, established, and data transfer; the set of all possible valid events and the actions performed by the machine when it encounters each event in each state. The outcome of this procedure was that we achieved a very high degree of automatic proof.

Our aims are constructing a model with a clear and accurate formulation of the communication protocol properties and discharge of all proof obligations. To satisfy these, attentive choice of invariants and machine theorems was important and eased the proof effort. A major focus of our work has been to explore the use of the Event-B method for formally specifying communication protocol. Two major approaches from the general specification were identified and applied to a set of example protocols: (1) an abstract model that defines operations that may be caused and (2) concrete model where the services is an active process with message exchange.

The supported language was sufficiently expressive and all proof obligations could be discharged. We reached a good degree of automatic proof. All interactive proofs involved a small number of steps and were straightforward to reach. The formal specification makes it able to verify, to validate and to implement communications protocols in an efficient way before the software development.

In this chapter, we express the motivation for using formal methods and we express our reasons about why formal methods should become more widely accepted. Our aim is to show the importance of formal methods and the difference that they can make to improve software development.

1.1 The Need of Formal Approach

There is a growing acceptance that formal methods form an essential part of the design of any reliable software system. This is because formal methods have the potential to eliminate ambiguities, and design faults and thereby avoid the associated system failures. In particular, formal model of a system can be used to prove system properties such as performance, reachability, consistency and correctness, mathematically [10]. Moreover, formal models and methods make software designs more tangible by allowing rigorous validation and verification. Validation and verification are important topics in software engineering. Their objective is to "identify and resolve software problems and high-risk issues early in the software lifecycle" to reduce the costs involved in finding and fixing problems later in the lifecycle [11]. In what follows, we will give a brief description of verification and validation.

Verification

Verification has two definitions in Institute of Electrical and Electronics Engineers (IEEE) Standard Glossary of Software Engineering Terminology.

> “ *Firstly, it is contrasted with validation as the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase". The second "is formal proof of program correctness[12]*”

Verifying a system means ensuring that "we are building the product right" [11]. Therefore, verification is used to show that a software product conforms to its specification. We use the term verification to refer to formal verification (See Section 2.1).

Validation

Validation is defined in IEEE Standard Glossary of Software Engineering Terminology as

> “ *The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements[12]*”

Validating a system is undertaken to ensure that "we are building the right product" [11]. Generally speaking validation refers to checking that a developed system meets its stakeholders' expectations. However, validation can happen in different stages of a software development lifecycle. In the initial stage of a lifecycle validation means ensuring that requirements and assumptions represent the stakeholders views of the system, while at the final stage of the development, validation can mean checking the final product against the stakeholders expectations. However, we use the term validation to refer to the process of validating the formal specification or the formal model against its requirements. So, by validation we mean ensuring that "we are building the right model with respect to its requirement", while it is assumed that the requirement represent the stakeholder expectations.

The value of Formal Methods is that they provide the ability to create a system that is “correct-by-construction”: the properties of the entire state space of a system may be deduced, confirming a property for all possible inputs [13]. The process may be applied to any discrete system, whether implemented in hardware or software.

Software Failures

The ever increasing reliance on software in all aspects of modern life makes reliability of software especially salient. Reliability is important when the cost of failure is expensive in monetary terms. It is even more important when human safety is at risk.

Examples of software errors and their associated costs are not lacking, whether the losses

are human or financial. All are rooted in a lack of quality and control over the development process. Some well-known examples are reproduced below:

- Therac-25 (1985-1987): Overdose of radiation on a medical device: 6 deaths and two years before correction [14]
- The Mars Climate Orbiter (1999): Insufficient test time, the robot crashed due to the saturation of its memory space [15]
- Patriot Missile (1991): drifting 6 meters per hour due to a conversion error. 28 soldiers killed in Dhahran [16]
- Ariane 5 (1996): excessive reuse of routines written for Ariane 4, 500 million dollars of pure loss [17]
- AT & T network crash (90's): an update of the routers makes the network unusable for 9 hours, 60M loss of turnover, cause of the fault: a missing break [18]

In a study [19] commissioned by the National Institute of Standards and Technology in 2002, the cost to the United States economy of software bugs was estimated to be almost 60 billion dollars annually. Other recent studies [20–22] highlight the role of software failure in accidents that caused loss of life or that had huge monetary consequences. For instance, the crash of Air France flight 447 in 2009, which killed 228 people, was partly due to discrepancies in the indicated air speed readings. The studies also show that miscalculated radiation doses at the National Oncology Institute in Panama caused the death of at least 18 persons in 2001. In 2003, the power grid in the United States and Canada shut down at an estimated cost of 13 billion dollars. One the contributing causes was determined to be faulty energy management software. The flood of costly software failures has not abated since the above accidents have been analyzed. A bug in an automatic trading program cost its proprietor 440 million dollars in 30 minutes when it was deployed on August 1st 2012 [23]. Also, closer to home, outdated software and a sloppily designed network architecture, allowed hackers from Iran to access Diginotar, a Dutch root certificate authority [24]. This break-in potentially exposed private communication of Iranian citizens to third parties. It also raised doubts about the current security model that is used for secure communication on the Internet.

Recurrent causes of these failures are: a lack of guards against unexpected or improper input and a mismatch between (implicit) specification and actual software. A lot of these faults could have been prevented by a careful adherence to strict coding and design standards, as well as thorough reviews. However, these methods face limitations in both the unexpectedness of some events and the sloppiness of human reviewing and testing.

1.2 Motivation

Increasingly, various and complex communication protocols are being used in distributed systems and computer networks of different types. Protocol verification is the process of analyzing the protocol specifications for the existence of various errors that could lead to

improper operation. Writing formal specifications can help in designing a protocol. Being able to check quickly if a protocol specification satisfies a more abstract specification can speed the design process.

Event-B is a formal method for system-level modeling and analysis. Main features of Event-B are the use of set theory as a modeling notation, the use of refinement to describe systems at various abstraction levels and the use of mathematical proof to verify consistency between refinement levels. Along the refinement, a set of proof obligations is discharged. The purposes of the proof obligations are to verify the consistency of a specification and to preserve the functionality from its abstract specification. It is difficult to manually generate and prove the proof obligations. Thus, the RODIN platform [25] has been built to provide an automated tool to generate and prove the proof obligations automatically or interactively. Besides, The RODIN platform can also support modeling in Event-B. Event-B, together with the RODIN platform, has been successfully applied to several practical safety-critical systems. Some concrete examples are a train controller system [26], hybrid systems [27, 28], an air traffic information system [29], a spacecraft system [30], and a metro system [31]. Event-B can be regarded as a method for correct-by-construction software development.

In this thesis, we apply a formal method such as Event-B to a collection of specifications concerning one application area, the field of communication protocols. The objective aims at formal modeling and analysis of three different kinds of communication protocol such as protocol from Data Link Layer, an e-commerce protocol, and a cryptographic protocol.

Despite the practical significance of the Sliding Window protocol, relatively little has been done on its formal verification. Stenning [1] only gave an informal manual proof for his protocol. A semi-formal manual proof is as well, presented in [2], and some versions of the protocol have been model-checked for small parameter values in [3, 4, 32]. It is needed to present an Event-B model for such protocol.

Another communication protocol is the electronic commerce protocols which are security protocol that allows customers and merchants to conduct their business electronically through the Internet. We choose to model NetBill protocol [5, 33]. It is difficult to verify the correctness of a design based on this protocol by hand. So it makes very appropriate the application of formal methods and techniques to the modeling and design of such protocol to gain high assurance about their correctness.

At last, we present an approach to model a cryptographic protocol. One of the earliest key establishment protocols designed for use in a mobile environment was that of Tatebayashi, Matsuzaki and Newman (TMN) [6], which has widely become known as the TMN protocol. While there exist a great variety of formal models that have been applied to model and analyze TMN protocol, such as Coloured Petri Nets [7] but none are made by Event-B Method.

To specify our protocols we follow the same principle. We start from abstract requirements, we gradually refine them to achieve concrete and fine-grained description of the model and verify each refinement level against the specification constructed in the previous refinement. Event-B allows correctness, i.e. the preservation of the properties of the earlier step, to be verified over successive refinements. This is reached by the creation and proving of logical

hypotheses, or proof obligations.

The correctness of the Event-B models, presented in this thesis, basically depends on proofs. When an automatic proof fails [9], it might be:
1. The statement to prove is true, but the automatic prover is not smart enough, so we must prove it with the interactive procedure.
2. The statement to prove is false, so the model has to be significantly modified.
3. The statement to prove cannot be proved, so the model should be enriched.

Reasons for Selection

A great variety of formal specification techniques exist, some of which are general purpose (such as Z, Vienna Development Method (VDM) or Common Object-oriented Language for Design (COLD)), while others are generally used in a specific domain of application (such as Language Of Temporal Ordering Specification (LOTOS), Specification and Description Language (SDL) and Process Specification Formalism (PSF)). The mathematical theories on which these languages are based range from set theory and temporal logic to lambda-calculus and process algebra.

The reason to choose Event-B, in particular, is motivated by several factors. Event-B is a simplification of Classic-B, it promotes a layered style of formal modeling, where a model is developed as chains of abstract models, level by level concrete details are progressively introduced via provably correct refinement steps. This style of modeling, which called refinement, decomposing machines into small, discrete events explicitly linked to their abstractions. This encourages more incremental refinement and easier verification by the generation of more easily discharged POs, enabling the practical construction of larger systems. However, Event-B's main advantage is its flexibility, both in the notation itself and its supporting tools. Event-B does not strictly define a notation, but a methodology for the construction and analysis of linked logical objects, and the notation seen when using support tools is actually an arbitrary front-end for users familiar with Classic-B and Z notation. Event-B was therefore developed closely alongside a supporting tool-set, RODIN[34]. RODIN has a modular architecture based on the Eclipse [1] framework with clearly defined Application Programming Interfaces (API) exposing the database, and supporting expansion via the addition of Eclipse plug-ins [35]. While building models with RODIN[2] platform, proof obligations are automatically generated. The strongest aspect of this platform is that a semi-automatic theorem prover is built into it, which saves the user lots of time by automatically generating a large majority of the proofs required. The RODIN platform is open source, so it supports a large number of plug-in tools, such as the animation tools AnimB[3], ProB[36] and model decomposition tool [37], The introduction of these plug-in tools makes modeling and proof construction more flexible.

[1]Eclipse platform homepage. http://www.eclipse.org/
[2]http://rodin.cs.ncl.ac.uk
[3]http://www.wiki.event-b.org/index.php/AnimB

1.3 Contributions

Formal methods have been applied in the design of a great number of systems. Many protocols have been specified and verified formally. Some protocol standards are even defined by means of a formal method. Event-B is a famous formal specification language, which provides a refinement mechanism and a set of proof obligations for modeling and verifying the specifications. The purpose of this thesis is to provide with a collection of protocol descriptions which illustrates how to use formal specification techniques such as the Event-B method in the field of communication protocols. Furthermore, we wish to give insight into the design and operation of communication protocols. The specifications in this thesis have a level of abstraction that is appropriate for a clear understanding of the protocols without having to deal with implementation details.

Publications

The following papers have been published or presented, and contain material based on the content of this thesis.

- S. El Mimouni and M. Bouhdadi. *Event Based Formalization of Communication Properties for an E-Commerce Protocol: An Event-B Approach*, International Journal of Engineering Research in Africa, Vol 37, pp3 78-90, 2018
- S. El Mimouni and M. Bouhdadi. *A Mechanized Formal Refinement Proof of Modbus Communication Using Event-B Proof System* , International Journal of Intelligent Engineering and Systems, Vol.11, No.4, pp.97-106, 2018.
- S. El Mimouni and M. Bouhdadi. *An event-B approach of the lightweight directory access protocol*, International Journal of Applied Engineering Research, vol. 11, no. 14, pp. 8229-8233, 2016.
- S. El Mimouni and M. Bouhdadi. *An Incremental Proof-Based Process of the NetBill Electronic Commerce Protocol*, Book Chapter, Networked Systems, Springer International Publishing, Volume 9944 of the series Lecture Notes in Computer Science, pp 209-213,2016
- S. El Mimouni and M. Bouhdadi. *Applying event-b refinement to the sliding window protocol*. In 18th IEEE International Conference on Computational Science and Engineering, CSE 2015, Porto, Portugal, October 21-23, 2015, pp 58-65.
- S. El Mimouni and M. Bouhdadi. *Formal modeling of the Simple Text Oriented Messaging Protocol using Event-B method*, Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2015, Marrakech, Morocco, November 17-20, 2015, pp 1-4.
- S. El Mimouni and M. Bouhdadi. *An incremental refinement approach to a development of TMN protocol*, Proceedings of the 2015 11th International Conference on Information Assurance and Security, IAS 2015, Marrakech, Morocco, December 14-16, 2015, pp 135–139.

- S. El Mimouni, R. Filali, A. Amamou, B. Boulamaat and M. Bouhdadi. *A Mechanically and Incremental Development of the Remote Authentication Dial-In User Service Protocol*. Proceedings of the 1st International Conference on Mathematical Methods & Computational Techniques in Science & Engineering (MMCTSE 2014). Athens, Greece, November 28-30, 2014, pages 199-203
- R. Filali, S. El Mimouni, A. Amamou, B. Boulamaat and M. Bouhdadi. *Modeling of SNMP Protocol in Event-B*. Proceedings of the 1st International Conference on Mathematical Methods & Computational Techniques in Science & Engineering (MMCTSE 2014). Athens, Greece, November 28-30, 2014, pages 208-211

1.4 Thesis Organization

This thesis follows a general pattern of moving from the general and abstract to the specific and implementation. After discussing the introduction and motivation of this work, the rest of the dissertation is organized as follows. First, the definition of formal methods, their advantages and their usage in standards are discussed in Chapter 2. Also, brief descriptions of some formal methods are given. Chapter 3 describes the Event-B formal method, showing the main elements in the method and how they relate. Then in Chapter 4, communication protocols are explained in detail, with the protocol development methodology. Next on Chapter 5, a proposed methodology for construction of formal models of the sliding window protocol in the Event-B formal notation is presented. On Chapter 6, the illustration of how to realize an Event-B model for the Netbill Electronic Commerce Protocol. Case study of a cryptographic protocol named TMN protocol are detailed in Chapter 7. Finally, the dissertation concludes in Chapter 8 where the conclusions and future work are stated.

Appendix A, B, and C represent respectively full Event-B model of the three protocols, which presented in this study: Sliding Window Protocol, Netbill protocol, and TMN protocol.

2 Formal Methods

The background of our thesis starts with the introduction of formal methods, it first asks what formal methods are, and what they are good for and then examples of real systems where formal methods were successfully applied are given. This chapter, explains classifications of formal methods along with overviews of several formal methods relevant to our thesis are introduced in. In last, refinement is briefly covered, including other formalisms.

2.1 Definition

Although formal program analysis techniques have a long history (including work by Hoare [38] and Dijkstra[39]), formal methods were only established in the 1980s. These techniques are used to analyze the behavior of software applications written using a programming language. The correctness of a program is then demonstrated using a program proof based on the calculation of the weakest precondition [40].

In the FM89 report [41], formal methods were defined as:

> " *Methods that add mathematical rigour to the development, analysis, and operation of computer systems and to applications based thereupon.*"

> " *...are nothing but the application of applied mathematics–in this case, formal logic—to the design and analysis of software-intensive systems.*"

Formal methods can be defined as mathematically-based techniques which are used for specifying and reasoning about software and hardware systems [42]. The essence of formal methods comes down to proof: (i) formulating proof obligations in terms of formal specifications and models, (ii) verifying, via algorithmic proof search, that a designed system meets its specifications, and (iii) algorithmically synthesizing all or parts of a system so as to satisfy its specifications. Unlike traditional calculus-based engineering mathematics, formal methods rely primarily on discrete mathematics and computer science formalisms such as finite state machines.

Formal methods can be divided into two basic elements: specification and verification. Each of these is discussed below.

Formal Specification

Formal specification is the specification of a program's properties in a language defined by a mathematical logic. A formal software specification is a statement expressed in a language whose vocabulary, syntax, and semantics are formally defined. So,formal specification is the use of notations derived from formal logic to define (1) the requirements that the system is to achieve, (2) a design to accomplish those requirements, and (3) the assumptions about the world in which a system will operate. The requirements explicitly define the functionality required from the system as well as enumerating any specific behaviors that the system must meet, such as safety properties

Formal Verification

Formal verification is the process of checking whether a design fulfills some requirements (properties) or not. The verification is done by providing a formal proof of an abstract mathematical model of the program, with respect to a certain formal specification or property. After the formal model of a program is built, we can validate a variety of properties over the model. Formal verification methods have recently become applicable by industry and there is a growing demand for professionals able to apply them. There are two well-established approaches to verification:

- **Model checking**:is an automatic verification technique mainly used in hardware controllers and communication protocols. This technique can be used for verifying concurrent systems with a finite state[43]. Usually supported by a computer tool "ModelChecker". It consists of automatically checking the desired properties on a system specification. The system is usually modeled by state machines. The properties to be checked are written in a temporal logic. Figure 2.1 presents the model checking approach. The tool receives as input the system model and the desired/undesired property to be checked. The output is the answer whether the system holds or not the requested property. In the last case, it is common to provide a counter-example showing why the property is not satisfied. In model checking all the valid inputs and possibilities are verified to guarantee the correctness of the system. To accomplish this task, a model checking tool uses a combinatorial amount of states to represent the system. In another words, the number of states required to represent a system increases exponentially with its size, leading to a problem known as state explosion.

  The success of the model checking verification depends on the user's expertise. Building a good model is a trade-off between representing the important points of the system and decreasing the size of the model to avoid the state space problem. In this process, the model designer must be really careful to not remove fundamental system characteristics and, at the same time, reduce the system complexity so the model is feasible to be verified.

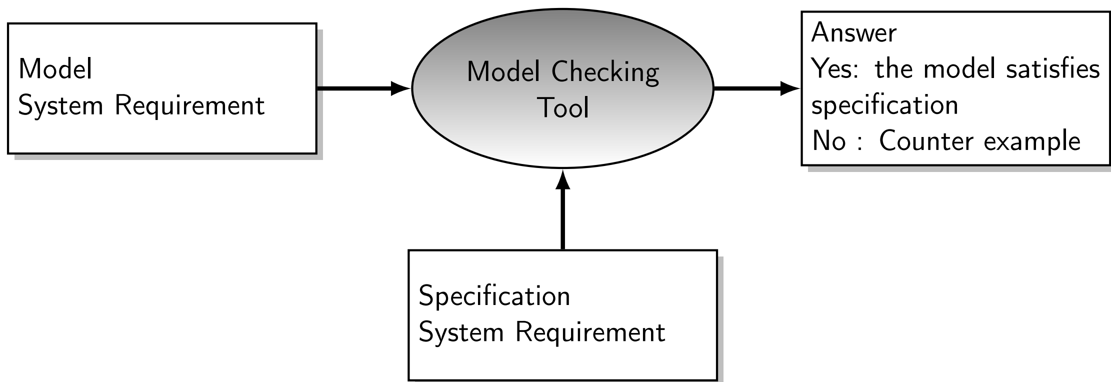


Figure 2.1 – The Model Checking Approach

- **Theorem proving**:An alternative approach to formal verification is the deductive reasoning verification technique [44] or theorem proving. Theorem proving is based on a set of axioms and rules of deduction or inference. The verification in this case consists in demonstrating a proposition from the axioms by application of the inference rules. In the case where the proof of a proposition is not found then we can not affirm that the proposition is a theorem. The specification of the system is done in first order or higher order logic. From this precise formulation of the system the designer can infer relations to prove its correctness. Often the theorem proving tool requires some guidance from the user and the proof itself can be almost obtained by an interactive process. Since the proof may be complex, steps of decomposition and simplification of proof obligations are necessary. This step can be repeated several times in the case of complex axioms. Figure 2.2 shows how the proof environment of a proof assistant works. The proof assistant tries to verify the proof obligations automatically or in an interactive way by using the hypotheses of the theoretical bases of the used logic or the inference rules.

While theorem proving can be used for reasoning in finite state systems, the model checking techniques can be used for systems with finite state space. It is worth to remember that even though these automated formal verification techniques propose automated solutions, neither approach works without some degree of human assistance. For example, theorem proving sometimes requires advice of which properties worth to verify. Model checkers, on the other hand, can quickly get stuck when checking millions of useless states and human guidance can be handy.

2.2 Why Use Formal Methods?

In the last few years, formal methods have emerged as an alternative approach to ensuring the quality and the correctness of hardware designs. Formal methods apply mathematical methods and formalisms to software construction, in different phases of the development process. They can be used in the specification phase, where they help in making the specification precise and in identifying areas where the development team lacks understanding of the problem to be solved. Sometimes they are used in the code generation phase to guarantee the correctness of the code with respect to the specification. Generally, the use of formal methods, however

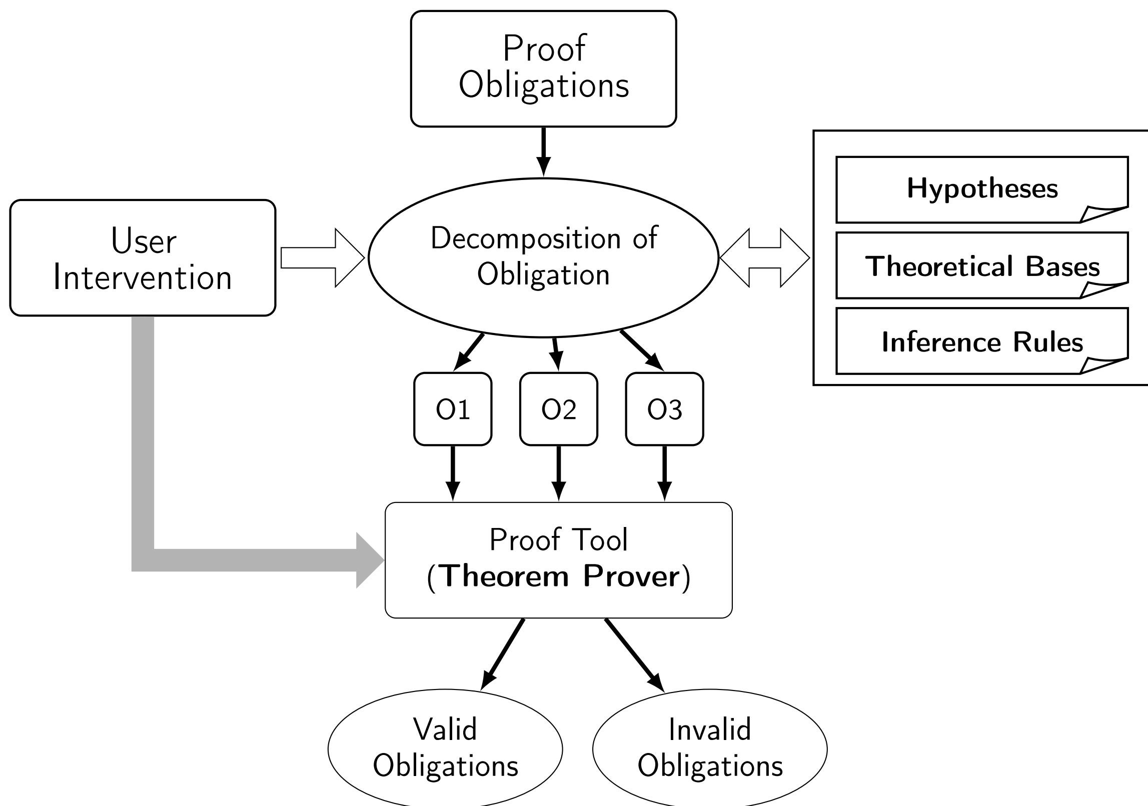


Figure 2.2 – Proof Process in Theorem-Proving Tools

insignificant, does have a positive effect on the reliability of the software in question, if other software engineering practices are not abandoned. Formal methods research has also helped mainstream software engineering to effect many changes for the better (structured programming being one example).

Formal methods are intended to systematize and introduce rigor into all the phases of software development. This helps us to prevent overlooking critical issues, provides a standard means to record various assumptions and decisions, and forms a basis for consistency within many related activities, which provides a method for producing software that is correct by construction.

Formal methods are claimed to allow the development of complex software under a firm mathematical foundation resulting in high quality, more correct software compared to conventional design methods. Besides contributing to a more firm and complete requirements and a better system design, formal methods also help the user to better understand a system.

Formal methods have been applied successfully in critical systems such as: an air traffic control information system [45], railway signaling systems [46], spacecraft systems [47] and medical control systems [48]. They have also been used for software tool specification [49], the specification of part of IBM's Customer Information Control System (CICS) [50] and a real-time system kernel [51].

2.2.1 Application Domains

In [52] the authors consider that formal methods can be categorized based on the application domain and based on the environmental assumptions. These categories can be used when choosing an appropriate formal method. The following application domains are considered:

Protocol Design and Engineering: Protocol engineering is the scientific methodology supporting protocol design. There has been a long-standing common history between formal methods and protocol design [53]. From the beginning, formal methods have been a core part of protocol engineering, and protocols have been a key application target for formal methods. It has become standard practice to use formal description and verification techniques when designing new protocols.

Software Design and Engineering: Formal methods are an important and growing part of software engineering, to which they provide theoretical foundations as well as analysis tools. Certain branches of software engineering, such as software architectures [54], are directly inspired from formal methods.

Hardware Design and Engineering: Formal verification tools are widely used which are aimed at detecting design mistakes. Hardware design, as an application domain, has contributed significantly to the development of formal methods by bringing many challenging problems (e.g., combinational logic, sequential logic, synchronous circuits, asynchronous circuits, system on chip, and network on chip) with all related issues of correctness and efficiency and, more recently, new issues about energy consumption.

Furthermore, different environments are considered, based on how well understood and predictable they are, with appropriate formal methods suggested for each environment as follows:

Nominal environment: which is well-understood and predictable. In such an environment, formal methods focus on correctness and performance issue.

Faulty environment: which is mostly understood and predictable, but abnormal events can affect the system. In such environments, the formal methods focus on dependability and performance issues.

Hostile environment: which is neither totally understood nor predictable and its behavior cannot be trusted. In such environments, the formal methods focus on security issues. Both general formal methods and dedicated formal methods have been successfully applied to security issues. For example, model checking has been used to find unknown attacks in security protocols [55, 56]. Examples of dedicated formal methods are security-oriented formal notations and software tools for automated analysis of security protocols. Furthermore, in order to ensure security one has to ensure both correctness and dependability. At the same time, formal methods cannot address security issues such as computer hacking, tampering or social engineering.

2.3 Classification

Formal methods can be classified into two categories. The first one is property-based methods, which are based on the indirect specification of properties. Property-based methods are, in turn, divided into two subcategories: axiomatic and algebraic methods. The second category is model-based methods, which are based on forming a formal model of the software system.

2.3.1 Property-Based Methods

In the property-based method, the system's behavior is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy. Property-oriented formal methods fall into two different subcategories: axiomatic methods and algebraic methods:

Axiomatic Methods

Axiomatic methods are based on first-order predicate logic, which is used to state pre and post conditions of operations over abstract data types. Examples of such methods are Larch [57], ANNA [58][59], and Enhanced Hierarchical Development Methodology (EHDM) [60].

Algebraic Methods

Algebraic methods are based on multi-sorted algebras, where properties of the behavior of the specified system are related to equations over the entities of the algebra. Clear [61], Algebraic Specification Formalism (ASF) [62] and ACT ONE [63] are methods based on equational logic. Temporal Logic [64], and Provably Correct Systems (ProCos) [65] are based on temporal logic. EVES [66], Higher Order Logic (HOL) [67], and OBJ3 [68] are methods based on higher-order logic. LOTOS [69] [70] is a well-known hybrid method, which is based on a combination of different semantic concepts.

2.3.2 Model-Based Methods

In a model-based method, one defines a system's behavior directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. Model-oriented formal methods can be further categorized as process algebra methods (e.g. Calculus of Communicating Systems (CCS) [71][72], Communicating Sequential Processes (CSP) [73]), state machine methods (e.g. ASLAN [74], PAISLey [75][76], Petri nets [77], and Statecharts [78] [79] [80]), or set theoretic methods (e.g. Rigorous Approach to Industrial Software Engineering (RAISE) [81], VDM [82], and Z [83]).

2.4 Some Formal Methods

2.4.1 Z Notation

Z notation was proposed by Jean-Raymond Abrial in 1977 with the help of Steve Schuman and Bertrand Meyer. Developed further in Programming Research at Oxford University. Z notation became an International Organization for Standardization (ISO) standard in 2002. Z is a formal notation for specifying and designing computer systems and software. Z is typically used in a modeling style [84] in which an abstract state is included, containing enough information to describe changes in state that may be performed by a number of operations on the system. Each of the operations defines a relation between a before and after version of the state. The state may contain invariants which are predicates relating the various components in the abstract state which should always apply regardless of the current state of the system.

Example:The Birthday Book

As an example, consider the specification of a birthday book, adapted from the Z reference manual [83].

The goal is to specify a system to record people's birthday. [NAME; DATE] The types NAME and DATE are given types, representing people's names and dates in the year respectively.

$BirthdayBook$

$known : \mathbb{P}\, NAME$

$birthday : NAME \rightarrow\!\!\!\!\!+ DATE$

$known = \mathrm{dom}\, birthday$

Figure 2.3 – Birthday Book Example

The state has two variables (Figure 2.3): known is the set of names in the birthday book and birthday is a partial function relating people's names to their birthdays. The state invariant known =dom birthday says how the values of the state variables are related.

$BirthdayBook_{Init}$

$BirthdayBook$

$known = \emptyset$

Figure 2.4 – Init Birthday Book Example

Initially (Figure 2.4), the birthday book is empty. (That birthday is empty follows from the state invariant).

$$
\begin{array}{|l}
\textit{AddBirthday} \\
\hline
\Delta BirthdayBook \\
name? : NAME \\
date? : DATE \\
\hline
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\} \\
\hline
\end{array}
$$

Figure 2.5 – Add Birthday Book Example

To add a birthday to the birthday book (Figure 2.5), we require an input name and a date. The name must not already be in the birthday book. The new name and date are simply added to birthday.

$$
\begin{array}{|l}
\textit{FindBirthday} \\
\hline
\Xi BirthdayBook \\
name? : NAME \\
date! : DATE \\
\hline
name? \in known \\
date! = birthday(name?) \\
\hline
\end{array}
$$

Figure 2.6 – Find Birthday Book Example

This read-only operation (Figure 2.6) will give the corresponding birth-date for a name, provided the name is in the birthday book.

$$
\begin{array}{|l}
\textit{Remind} \\
\hline
\Xi BirthdayBook \\
today? : DATE \\
cards! : \mathbb{P}\, NAME \\
\hline
cards! = \{n : known | birthday(n) = today?\} \\
\hline
\end{array}
$$

Figure 2.7 – Remind Birthday Book Example

This read-only operation (Figure 2.7) will find the names of people whose birthday falls on a given date.

2.4.2 VDM

VDM is one of the longest established model-oriented formal methods for the development of computer-based systems and software. It consists of a group of mathematically well-founded languages and tools for expressing and analyzing system models during early design stages, before expensive implementation commitments are made.

VDM is a model-oriented specification technique in which software systems are specified by defining the possible states of (an abstraction of) the desired system, together with a set of initial states and operations for changing from one state to another. The state is usually expressed as a set of state variables, with constraints on the values they can take. Constraints are written in an extension of typed (many-sorted) predicate calculus, which has support for finite sets, maps(finite partial functions), finite sequences, Cartesian products and more.

Example: The Birthday Book

Type Definitions:

NAME and DATE are (type) parameters to the specification, representing peoples' names and dates in the year, respectively.

State Definition:

state *BirthdayBook* of
known : NAME **-set**
birthday : NAME $\xrightarrow{m}$ DATE
inv *mk-BirthdayBook (names,info)* $\underline{\Delta}$ *names* = **dom** info
init σ $\underline{\Delta}$ σ.known = {}
end

Figure 2.8 – Birthday Book Example in VDM

The state has two fields (Figure 2.8): known is the set of names in the birthday book and birthday is a partial function relating peoples' names to their birthdays. The state invariant says how the values of the state variables are related. Initially, the birthday book is empty.

Operation Definitions

To add a birthday to the birthday book (Figure 2.9), we require an input name and a date . The name must not already be in the birthday book. The change is effected by a function overwrite. Note that write permission to "known" is needed because it changes implicitly, being the domain of birthday.

AddBirthday (name : NAME ; date : DATE)
ext wr known : NAME -**set**
 wr birthday : NAME $\xrightarrow{m}$ DATE
pre name $\notin$ known
post birthday = birthday $|\{name \rightarrow$ date$\}$

Figure 2.9 – Add Birthday Operation in VDM

The next operation (Figure 2.10) looks up the birth date corresponding to a given "known's" name.

FindBirthday (name : NAME) date : DATE
ext rd known : NAME -**set**
 rd birthday : NAME $\xrightarrow{m}$ DATE
pre name $\in$ known
post date = birthday(name)

Figure 2.10 – Find Birthday Operation in VDM

The third operation (Figure 2.11) finds the names of all "known" whose birthday falls on a given date

Remind (day: DATE) bornOn: NAME -**set**
ext rd known : NAME -**set**
 rd birthday : NAME $\xrightarrow{m}$ DATE
post bornOn = {n: NAME | n $\in$ known $\wedge$ birthday(n) = day }

Figure 2.11 – Remind Birthday Operation in VDM

2.4.3 B Method

B is a method for specifying, designing and coding software systems [13] invented by Jean-Raymond Abrial based on the first order logic and set theory. B-method defines three basic components: abstract machine, refinement, and implementation. Abstract machines are divided into three levels: the MACHINEs, which describe the highest level of specification, the REFINEMENTs, which include all the intermediary steps between the specification and the code, and the IMPLEMENTATIONs, which define the coding.

The B method defines a unique notation, known as Abstract Machine Notation (AMN) [13], which allows us to describe the above mentioned three levels of abstraction.

The basic structure of B model is presented in figure 2.12.

```
MACHINE n
SETS S
CONSTANTS C
PROPERTIES P
VARIABLES V
INVARIANT I
INITIALISATION T
OPERATIONS
  outputs <-- op (inputs) =
        PRE preconditions
        THEN substitutions
        END ;
  ...
END
```

(a) Structure of a B Machine

```
REFINEMENT n_r
REFINES n
SETS S_r
CONSTANTS C_r
PROPERTIES P_r
VARIABLES V_r
INVARIANT I_r
INITIALISATION T_r
OPERATIONS
  outputs <-- op (inputs) =
        PRE preconditions_r
        THEN substitutions_r
        END ;
  ...
END
```

(b) Structure of a B Refinement

Figure 2.12 – Structure of a B Model

The name of the machine is given in MACHINE clause. The machine n can have a list of parameters p. These parameters are assumed to be independent of each other and their logical properties are specified by CONSTRAINTS clauses. Each item defined in the SETS clause is corresponding to a definition of given or enumerated sets. CONSTANTS clause contains a list of identifiers which are constant within the operations of the machine. The clause PROPERTIES contains predicates and formula conjunction defines logical properties for sets and constants. The variables of the machine are listed in the VARIABLES clause. The constraints on the variables, including the typing of the variables, are specified in the INVARIANT of the machine. DEFINITIONS of mathematical abbreviations can be given in terms of the state variables and constants. The initialization operation of the machine is specified in the INITIALISATION section of the machine. The methods or operations of the machine are listed in the OPERATIONS section. Input parameters are listed after the name of the operation in its definition, and output parameters are listed to the left of an arrow from the operation name. It is possible to link and extend machines via the **USES**, **SEES**, **INCLUDES** and **EXTENDS** structuring mechanisms. These have the following roles:

- **SEES**: to support the read-only sharing of a component between other subsystems of a development, and the independent refinement of this component.
- **USES**: to support the definition of a shared read-only component between machines in a single subsystem of a development, whose state can be linked with the states of the machines which use this component.
- **INCLUDES**: to support the incremental addition of operations and state to a specification of a subsystem.
- **EXTENDS**: as with **INCLUDES**, but with all operations of the included machine becoming operations of the extending machine.

Promoted and supported by RATP [1], the B method, and Atelier B, the tool implementing it, has been selected as a tool by industries in the field of critical systems regarding the risk. A remarkable example of its use is in the railway control system in Paris, the Meteor line 14 driverless metro in Paris: Over 110 000 lines of B models were written, discharging 29,000 proofs, and generating 86 000 lines of Ada. No bugs were detected after the proofs, neither at the functional validation, at the integration validation, at the on-site test, nor since the metro lines operate (October 1998) which has been working since 1998 [85]. Another example is KVB: Alstom an automatic train protection for the French railway company (SNCF), installed on 6,000 trains since 1993. 60,000 lines written in B method, discharging 10,000 proofs and finally 22,000 lines of Ada.

Example: The Birthday Book

Figure 2.13 shows the birthday book example as a B machine. The VARIABLES section defines that known and birthday are the variables making up the state. There is an explicit initialisation and in the OPERATIONS section the operations AddBirthday and RemoveBirthday are described. Both known and birthday are set to ∅. In the definition of the operation AddBirthday both variables are changed explicitly. Generally in B all changes to variables must be stated explicitly via generalized substitutions.

2.4.4 CSP

CSP[73] is a formalism for modelling concurrent systems, devised by Hoare in 1978 [86]. CSP allows description of systems in terms of component processes that operate independently, and interact with each other solely through the passing of messages.

CSP is used to describe a network of communicating processes. A process is composed of two things, namely events and primitive processes. The primitive processes are fundamental behavior, such as the deadlock process STOP and the successful termination process SKIP.

A process interacts with its environment by synchronously engaging in atomic events. A process which my participate in event a then act according to process description P is written: a → P. The environment can decide between two processes using the choice operator $\Box$. P $\Box$ Q represents the process that offers the choice to the environment between behaving as process P or as process Q. There is also a nondeterministic choice operator ⊓: P ⊓ Q represents the process that internally chooses between behaving as P or Q.

Other CSP primitives:

- P; Q (sequential composition)
- $P|[X]|$ Q (synchronous), P||| Q (asynchronous)
- P_1 ▿ e → P_2 (interrupt process)

[1] Régie Autonome des Transports Parisiens: operates bus and metro public transport in Paris

```
MACHINE BirthdayBook
SETS
    NAME; DATE
VARIABLES
   known,birthday
INVARIANT
   known:POW(NAME)
    birthday:NAME+-- >DATE
   known=dom(birthday)
INITIALISATION
    known,birthday := {},{}
OPERATIONS
AddBirthday(name,date) =
    PRE
        name:NAME &
        date:DATE &
        name/:known
    THEN
       birthday(name) := date || known := known \/ name
   END
RemoveBirthday(name,date) =
    PRE
        name:NAME & date:DATE &
        name:dom(birthday)&
        date:ran(birthday)&
        date=birthday(name)
    THEN
       birthday := {name} ⩤birthday
   END
END
```

Figure 2.13 – The Birthday Book Example in B

2.5 Other Formalisms

2.5.1 Action Systems

Action system is a formalism introduced by Back and Kurki-Suonio [87] to reason about behavior of parallel and distributed systems. The basic entities of action systems are actions that can take place in the system. The actions are atomic, i.e. an execution of any action cannot be interfered by the other actions of the considered action system. The requirement of atomicity allows us to claim that a parallel execution of an action system establishes the same result as a sequential execution.

An action system statement has the following form:

$$A = begin\ var\ x := x_0;\ \textbf{do}\ A_1[]...[]A_m\ \textbf{od}\ end\ :\ z \tag{2.1}$$

We denoted the local variables of action system A as x, initialized local variables as x_0, the global variables as z, and $A_1,...,A_m$ are the actions of A.

Each action has the form:

$$A_i = g_i \longrightarrow S_i \tag{2.2}$$

Here g_i is the guard of the action and S_i is the statement (or body) of the action. The guard of action A is denoted by gA and the statement of it by sA. Thus action A can be written in the following way A=gA ⟶ sA. The body of an action can be an arbitrary statement. It may terminate or not. The local and global variables of action system are distinct. They form the state variables y, y=x ∪ z. The set of state variables accessed in action A is denoted vA.

2.5.2 Process Algebra

Process Algebras are mathematically rigorous languages with well defined semantics that allow describing and verifying properties of concurrent communicating systems.

Several process Algebra exist like :CSP proposed by Hoare [73], CCS proposed by Milner [72], and Algebra for Communicating Processes (ACP) introduced by Bergstra and Klop [88]. Besides this process Milner developed π-Calculus [89], which is an evolution of CCS.

The subject of each of these is an abstract concept of a "process", defined as a component which may react to external events, generate events, and carry out internal processing. Each language provides means to specify processes, combine processes, and reason about their properties (although reasoning is usually carried out in an auxiliary proof language, such as a modal logic).

2.5.3 Temporal Logic

Temporal logic was originally developed to be used in philosophy, and it has been suggested to be used in computer science by Brustall [90] and Pnueli[91]. Temporal logics have been widely used for the specification of reactive systems and there exist important works which prove its utility for the modeling of such systems[92]. Temporal Logic is a formal system for specifying and reasoning about concurrent programs

Temporal logic allows to express two kinds of proprieties of a program :

- *Safety properties* : which assert that the program does not do something bas, eg. system should not crash.
- *Liveness properties*: which assert that the program does eventually do something good. eg. every packet sent must be received at its destination.

2.6 Refinement

Refinement is the approach that synthesizes a program from a specification step by step, such that each step increases the level of precision with respect to the initial specification. Each added step represents an implementation choice, such as the choice of algorithm for implementing a given function, or the choice of a concrete data type to implement an abstract type[2] or even the weakening of a precondition of an operation. Refinement ensures that the resulting implementation corresponds to the initial abstract specification.

Different refinement steps must be proved correct, i.e. the effect of the concrete specification must not contradict the effect of the abstract/refined specification, in order for the final program to keep the same properties as the original specification. Every step, thus produces a number of refinement proof obligations that must be discharged. The correctness of each different step is in principle much easier to establish than the overall correctness.

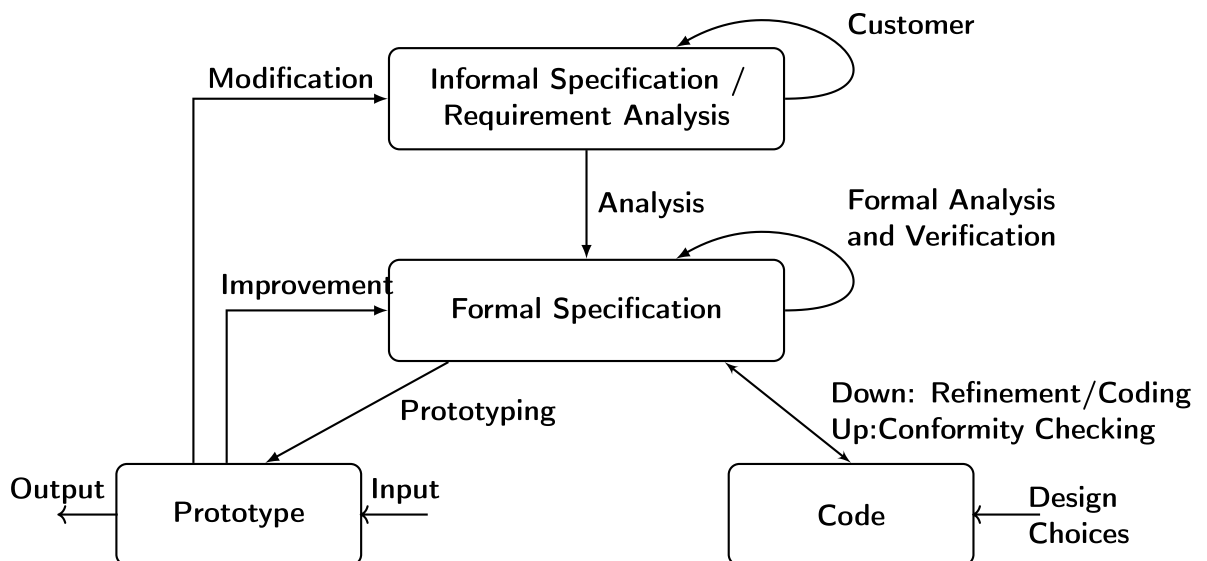


Figure 2.14 – Balzer's Life Cycle

This is the technique followed by approaches like Z, VDM, and B. This very special ability to link a high-level view to the resulting code via a chain of design choices and proofs is particularly suitable and has been used in the context of a vertical application of the Balzer life cycle[3] (Figure 2.14) [93]. Refinement is a very well-known and successful example of the use of formal methods in industry. It is well supported in terms of tools, and what is more, it provides the easiest way to realize Balzer's vision of the software development process. As cited before, software development through a chain of formally verified refinement steps is sometimes referred to as correct-by-construction.

[2] say the implementation of a set as a linked list

[3] The novelty of Balzer's life cycle is that it explicitly advocates the use of formal methods at different stages, and in particular where the relationship between the requirements, the model, and the implementation is concerned

The stepwise refinement process is illustrated in Figure 2.15. Since the refinement process preserves the system's correctness, the software developed by refinement is correct by construction.

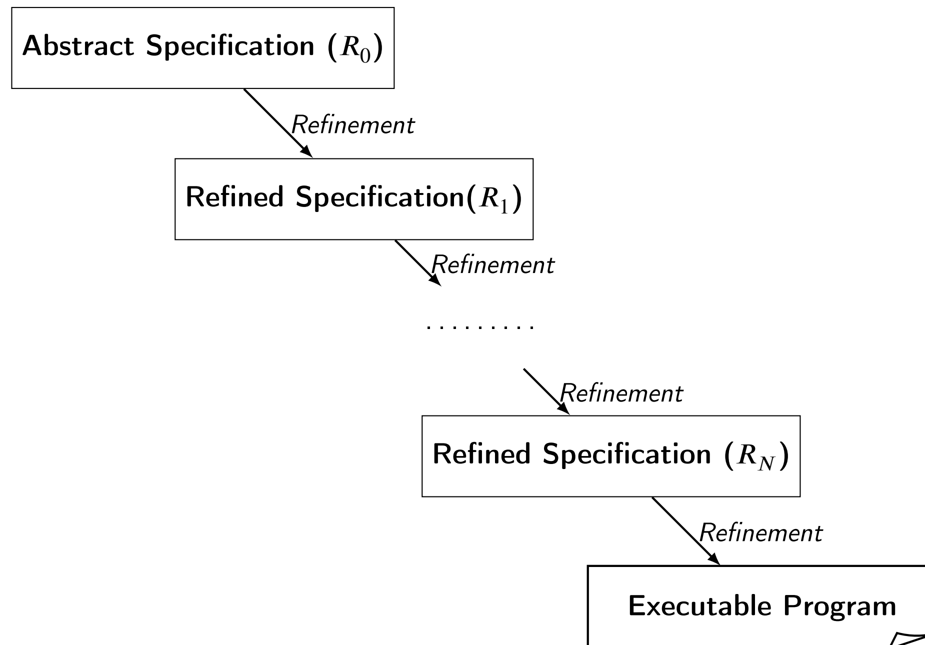


Figure 2.15 – Stepwise Refinement of Specifications

2.7 Summary

This chapter has covered the formal methods background, techniques, and tools used for specification and verification. We began by describing formal methods as mathematically based techniques for describing the properties of a system, then we give the reasons of using formal methods, such as improving system understanding and proving system properties formally, have meant that some certification and standard bodies require the use of formal methods in safety-critical system development. We have also presented some existing formal methods, along with a simple example of each formal method.

3 Event-B: A Modeling Language

Event-B inherited some features from its predecessor formal languages. As a result, in this chapter, we will go through from Hoare logic, Dijksra's weakest precondition to Refinement Calculus these techniques which their modeling and verification approaches affected the existing modeling and verification features of Event-B. As Event-B is both a language and a method. Its concepts are limited and allow the user to manage a simple palette of tools: axioms, theorems, theories, events, machine, context, and refinement. This chapter, presents the fundamental mathematical structures that are needed to formalize the Event-B model. The simple notions identified in this chapter underline the formal reasoning about properties of programs, specifications using the Event-B method.

3.1 Fundamentals of Event-B

3.1.1 Hoare Logic

The Hoare logic [38] is a well known approach to verifying "small" programs. It was developed during the late sixties and allows to formulate properties for the partial correctness of while programs. These properties can then be verified using the Hoare calculus. Originally it was not possible to express and verify termination properties. The original concepts were seeded by the work of Robert W. Floyd, who had published a similar system [94] for flowcharts.

A Hoare formula 3.1 has three parts:
precondition: P (formula in predicate logic)
program: S
postcondition: Q (formula in predicate logic)
notation:

$$\{P\}\ S\ \{Q\}^1 \qquad (3.1)$$

Its interpretation is as follows: if precondition P holds and program S terminates, then postcondition Q holds after S has been executed. Hoare logic was very influential in its

[1] Note: Hoare's original notation was P {S} Q instead of {P} S {Q}, but the latter form is now more widely used.

contribution to the state of the art and influences can be seen in the preconditions and postconditions of Design by Contract approaches of Eiffel [95], SPARKAda [96], and JML [97]. Another influence of Hoare logic was in Dijkstra's wp-calculus [39] which later contributes to the semantic definition of the B-method; which is a predecessor of our formal method of interest, Event-B method.

Example: {X = 1} X:=X+1 {X = 2}. Here P is the condition that the value of X is 1, Q is the condition that the value of X is 2 and S is the assignment command X:=X+1 (i.e. 'X becomes X+1'). {X = 1} X:=X+1 {X = 2} is true.

When verifying programs there are two types of correctness:

Partial and Total Correctness

For a partial correctness specification {P} S {Q}, we'll be able to get the following information: Given a starting state that satisfies P, S may terminate or not. If S terminates, after S's execution, you will reach a final state which satisfies Q. If not, Q is useless since there is no final state.

Example:

```
{x==10}
while (y != 0):
    y = y - 1
x = 0
{x==0}
```

It is a partial correctness specification. If y is initialized with some number equal or greater than 0, S will terminate and after that x is 0. While if y starts with a negative number, S will loop forever and since it does not terminate, you will not reach a state 'after S's execution'.

Indeed, Q can be anything if S is a dead-loop. For example, for any P and Q:

```
{P}
while (true):
    y = y - 1
{Q}
```

is always a partial correctness specification.

If P is not strong enough, we can not guarantee S's termination, let alone reason about the state after S's execution. In this case we can manually add a condition: S terminates. With P and it, the reasoning can continue.

For total correctness specification {P} S {Q}, P is strong enough to guarantee S's termination, so you can conclude that S will terminate and the final state satisfies Q.

Example:

```
{x==10}
while (x != 0):
    x = x - 1
{x==0}
```

is a total correctness specification.

We can note informally :

$$Total\ Correctness = Termination + Partial\ Correctness \tag{3.2}$$

3.1.2 Dijkstra's Weakest Precondition

A few years after Hoare's paper, Dijkstra published his influential book "A Discipline of Programming" [40] in which a framework for specifying semantics based on 'predicate transformers' – rules for transforming predicates on states – is described. In work related to the Hoare-style axiomatic proof, Dijkstra developed the concept of the weakest precondition in "Guarded Commands, Non-Determination and Formal Derivation of Programs" [39], This title contains a lot of confused words, but the concept is actually very simple. As we have seen previously, Hoare's logic gives us rules explaining how the actions of a program behave. But it does not tell us how to apply these rules to establish a complete proof of the program.

Dijkstra reformulates Hoare's logic by explaining how, in Hoare triplet {P} S {Q}, the instruction, or the instruction block, S transforms the predicate P, into Q. This form is called forward-reasoning. We calculate from a pre-condition and one or more instructions, the strongest post-condition we can achieve. Informally, considering what is received as input, we calculate what will be returned in the output. If the desired post-condition is at most as strong, then we have proved that there is no unwanted behavior.

Example: Here are a number of valid Hoare Triples:

- {x = 5} x := x * 2 { true }
- {x = 5} x := x * 2 { x > 0 }
- {x = 5} x := x * 2 { x = 10 } ∨{ x = 5 }
- {x = 5} x := x * 2 { x = 10 }

All triples are true, but the last one is the most useful because it contains the strongest post-condition x=10,because, If {P} S {Q} and for all Q' such that {P} S {Q'}, Q ⇒ Q', then Q is the strongest postcondition of S with respect to P. { x = 10 } is strongest because this post-condition implies all the post-conditions.

The form that interests us, the calculation of the weakest pre-condition, works in the opposite direction, we are talking about "backward reasoning". Starting from the desired post-condition and the instruction we are dealing with, we will find the weakest precondition that ensures this operation. If our actual pre-condition is at least as strong, that is, it implies the weakest pre-condition, then our program is valid.

Djikstra introduces 'weakest preconditions' as a predicate transformer semantics that treats assignment statements in a way equivalent to Hoare's assignment axiom, using the notation wp(S,Q). Let S be a statement in some programming language and let Q be a predicate. wp(S,Q) is the set of initial states (described by a predicate) for which S terminates and Q is true on termination. The weakest precondition is used to identify the set of all initial states such that when the statements S are applied the program terminates and postcondition Q holds. wp is a predicate transformer that relates a precondition to any post-condition Q.

For example, in order to have a variable x equal to 3 after the assignment statement "x := x+1", the program state prior to this statement must have x equal to 2. Therefore, wp(x := x+1; x = 3) is the set of all program states such that x has value 2.

A program S is correct with respect to predicates [P,Q] if P ⇒ wp(S,Q).

3.1.3 Refinement Calculus

The idea of stepwise development was first proposed by Dijkstra [98] and Wirth [99] as an approach to develop correct programs. Later, Back [100] proposed the mathematical foundation of the refinement process, which was further developed into the refinement calculus framework by Back and Von Wright [101]. The refinement based development is formalized using the weakest pre-condition semantics [40].
Let S be a statement and P a post-condition predicate, i.e., a set of states which can be reached after executing S, then wp(S, P) represents weakest pre-condition that guarantees establishing P after executing S. Now suppose that S is refined by T, then the refinement relation ⊑ between S and T can be expressed [101] by the weakest pre-conditions as follows:

$$\text{S} \sqsubseteq \text{T} \ iff \ \text{for all P} : \text{wp(S, P)} \Rightarrow \text{wp(T, P)}$$

Where ⇒ stands for implication between predicates. Informally speaking, a refined specification is said to refine an abstract specification if any postcondition, which abstract specification can establish, is also established by the refined specification. Alternatively, the refinement relation can also be expressed using before-after predicates relating initial and final states of statements. The refinement process reduces non-determinism of the abstract specification and makes it more implementable. The transitivity of the refinement relation guarantees that each intermediate refined specification from R_1 to R_N as well as the Executable program are correct refinements of initial Abstract Specification (R_0) [101].

$$R_0 \sqsubseteq R_1 \sqsubseteq \ldots \sqsubseteq R_N \sqsubseteq \text{Executable Program}$$

The refinement can be categorized [101] into two forms: data and algorithmic refinement. The data refinement replaces an abstract data structure by the one which is more concrete and closer to the implementation, while preserving the global system behavior. On the other hand, algorithmic refinement introduces more concrete programming language structures to work on the data while leaving the structure of the data unchanged.

Refinement of the Specification

The specification is the principal feature of abstract programs. Its precondition (pre) describes the initial states; its postcondition (post) describes the final states; and its frame (w) lists the variables whose values may change. The compact form is w: [pre , post].
Refined specification from a given one might seem something enigmatic. Luckily, there are a large number of refinement rules to help one in doing so. Carroll Morgan [102] list more than 70 such rules in his book "Programming from specifications". We list below just two of this rules.
A specification is improved by strengthening its postcondition, so that the new postcondition implies the old: if a book is available in paperback and hardback, then it is available in paperback at least.

Rule 1 (Strengthen Postcondition:) *If post′ $\Rrightarrow$ post , then*
w: [pre , post] $\sqsubseteq$ w: [pre , post′]

(The statement A $\Rrightarrow$ B is used to indicate that A is stronger than B. $\Rrightarrow$ is the "implied everywhere" symbol and indicates that "everywhere A implies B") .The requirement post′ $\Rrightarrow$ post must hold whenever the rule is used, and it is called the proviso.

A different kind of improvement is gained by weakening a precondition, so that the old precondition implies the new: if at least 4Mb is required, then certainly at least 2Mb is required.

Rule 2 (Weaken Precondition:) *If pre $\Rrightarrow$ pre′ , then*
w: [pre , post] $\sqsubseteq$ w: [pre′ , post]

3.2 Mathematical Notation

Mathematics is used in "formal" software specification and development in two related but distinct ways:

Discrete mathematics (dealing with properties of sets, sequences,etc.) is useful for modeling requirements or designs in a clear, precise and abstract manner.

Systems of proof and reasoning can support verification and validation of specifications and development steps - proving the internal consistency of individual specification modules and the correctness of development steps between related modules comes into the category of verification, while checking that certain expected properties hold by attempting to prove them from a module description comes into the category of validation.

3.2.1 Set Theory

Set theory is branch of mathematics that deals with the properties of well-defined collections of objects, which may or may not be of a mathematical nature, such as numbers or functions. Event-B uses this notation to define the constants, relations and data structures used in models. A key feature of the Event-B set-theoretical notation is the models of relations as sets

of mappings. Different types of relations and functions are also defined as sets of mappings with different additional properties. The notation of Event-B follows the classic set-theoretical notation such as Cartesian product (E ↦ F ∈) S×T, power set (E ∈ ℙ(S)), set comprehension (E ∈ {x.P|F}), set equality (S = T), in which E and F stand for expressions; S and T stand for sets; P stands for a predicate. The set comprehension says that E is the set of elements in F that satisfy the predicate P. Table 3.1 and Table 3.2 detail binary relation operators and functions syntax.

Table 3.1 – Binary Relation Operators [9]

Name	Syntax	Definitions
Set of all binary relations	r ∈ S ↔ T	r ⊆ S × T
Domain	E ∈ dom(r)	∃y.E ↦ y ∈ r
Range	F ∈ ran(r)	∃x.x ↦ F ∈ r
Set of all total relations	r ∈ S T	r ∈ S ↔ T ∧ dom(r)=S
Set of all subjective relations	r ∈ S T	r ∈ S ↔ T ∧ ran(r)=T
Set of all total and subjective relations	r ∈ S T	r ∈ S T ∧ r ∈ S T
Converse	E ↦ F ∈ r^{-1}	F ↦ E ∈ r
Domain subtraction	E ↦ F ∈ S ⩤ r	¬ E ∈ S ∧ E ↦ F ∈ r
Range subtraction	E ↦ F ∈ r ⩥ T	E ↦ F ∈ r ∧ ¬ F ∈ T
Relational image	F ∈ r[U]	∃x.x ∈ U ∧ x ↦ F ∈ r

Table 3.2 – Function Operators [9]

Name	Syntax	Definitions
Identity	E ↦ F ∈ id	E = F
Set of all partial functions	f ∈ S ⇸ T	f ∈ S ↔ T ∧ (f^{-1} ; f) ⊆ id
Set of all total functions	f ∈ S → T	f ∈ S ⇸ T ∧ S=dom(f)
Set of all partial injections	f ∈ S ⤔ T	f ∈ S ⇸ T ∧ f^{-1} ∈ T ⇸ S
Set of all total injections	f ∈ S ↣ T	f ∈ S → T ∧ f^{-1} ∈ T ⇸ S
Set of all partial surjections	f ∈ S ⤀ T	f ∈ S ⇸ T ∧ T=ran(f)
Set of all total surjections	f ∈ S ↠ T	f ∈ S → T ∧ T=ran(f)
Set of all bijections	f ∈ S ⤖ T	f ∈ S ↣ T ∧ f ∈ S ↠ T

3.2.2 Sequent Calculus

In order to present the sequent calculus, we need to define certain notions:

(i) *A sequent* composes of a finite set of predicates (called hypotheses) and a single goal predicate. A sequent with the hypotheses H and the goal G is written as follows:

$$H \vdash G$$

Informally, a sequent stands for some statement that we want to prove: Under the assumption of the hypotheses H, prove the goal G.

(ii) An *inference rule* is made of two elements: the antecedent element and the consequent element. It is used when we construct a sequent based proof. The antecedent denotes a

finite set of sequents, while the consequent denotes a single sequent. An inference rule R1 with antecedent A and consequent C is usually written as follows:

$$\frac{A}{C}\ \textbf{R1}$$

We could say that inference rule R1 yields a proof of sequent C as soon as we have proofs of each sequent of A. If the antecedent A is empty, we say the inference rule R1 yields a proof of sequent C.

(iii) A *theory* is a set of inference rules.

(iv) The proof of a sequent within a theory is simply a finite tree with certain constraints.

As an example for the construction of a proof tree. We give the following theory involving sequents S1 to S6 and rules R1 to R6:

$$\frac{}{S2}\ \textbf{R1} \qquad \frac{}{S4}\ \textbf{R2} \qquad \frac{S2 \quad S3 \quad S4}{S1}\ \textbf{R3} \qquad \frac{}{S5}\ \textbf{R4} \qquad \frac{S5 \quad S6}{S3}\ \textbf{R5} \qquad \frac{}{S6}\ \textbf{R6}$$

The proof of sequent S1 is illustrated in Figure 3.1:

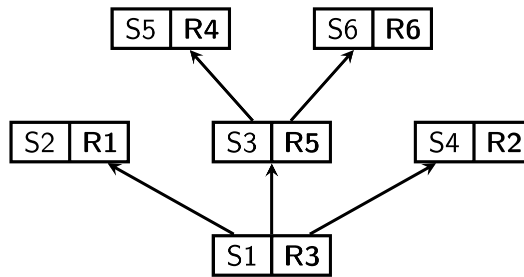


Figure 3.1 – A Proof Tree

This tree can be interpreted as follows: In order to prove S1, we prove S2, S3, and S4, according to rule R3. In order to prove S2, we prove nothing more, according to rule R1. In order to prove S3, we prove S5 and S6, according to R5. And so on.

This tree can be represented also in this form:

$$S_1\ \mathbf{R_3} \begin{cases} S_2\ \mathbf{R_1} \\ S_3\ \mathbf{R_5} \begin{cases} S_5\ \mathbf{R_4} \\ S_6\ \mathbf{R_6} \end{cases} \\ S_4\ \mathbf{R_2} \end{cases}$$

3.2.3 Inference Rules

The proof obligations are discharged using certain inference rules. The theory of underlying Event-B is based on some initial inference rules. This theory is refined or extended by

propositional logic and predicate logic. In this section we summarize the inference rules in Event-B and their definitions.

Initial Theory

Basic inference rules of mathematical reasoning are called **HYP**, **MON**, and **CUT**. Here are their definitions:

- **HYP**: If the goal P of a sequent belongs to the set of hypotheses of this sequent, then it is proved.
- **MON**: In order to prove a sequent, it is sufficient to prove another sequent with the same goal but with fewer hypotheses.
- **CUT**: If you succeed in proving a predicate P under a set of hypotheses H, then P can be added to the set of hypotheses H for proving a goal Q.

$$\frac{}{H, P \vdash P}\ \textbf{HYP} \qquad \frac{H \vdash Q}{H, P \vdash Q}\ \textbf{MON} \qquad \frac{H \vdash P \quad H, P \vdash Q}{H \vdash Q}\ \textbf{CUT}$$

The Propositional language

The propositional logic extends the initial inference rules by adding predicate such as falsity, negation, conjunction, disjunction and implication. Each kind of predicates is given two rules: a left rule, labeled with **_L**, and a right rule, labeled with **_R**. This corresponds to the predicate appearing either in the hypothesis part (left) or in the goal part (right) of the consequent of the rule. Here are their definitions:

- **FALSE_L**: If the hypotheses of a sequent included false assumption, then the sequent is proved.
- **FALSE_R**: If a goal P and its negation ¬ P are both proved under hypotheses H, then H is false.
- **TRUE_L**: This rule can be proved by using **MON** (substitute P by ⊤).
- **TRUE_R**: This rule can be proved by using **HYP_L** and **AND_L**.
- **NOT_L**: To prove a sequent which says the goal Q of the sequent is proved under the hypotheses H and ¬P, we only need to prove the sequent with hypotheses H and ¬ Q, and with goal P.
- **NOT_R**: If we succeed in proving the goal ⊥ under hypotheses H and P, the the the sequent with goal of ¬P can be proved.
- **AND_L**: The relation between hypotheses H and some predicate P is conjunction.

- **AND_R**: To prove a sequent with goal P ∧ Q, we should individually prove the sequent with the same hypotheses H, but with goals P and Q, respectively.
- **OR_L**: To prove a sequent with hypotheses include P ∨ Q, we have to prove the sequent with the same goal R, but with hypotheses P and Q, individually.
- **OR_R**: If we succeed in proving the goal Q under hypotheses H and ¬P and goal Q, then the new sequent with new goal P ∨ Q can be proved.
- **IMP_L**: This rule is a derived rule from **OR_L** and **FALSE_R** (substitute P ⇒ Q by ¬ P ∨ Q).
- **IMP_R**: This rule is the same as rule OR_R (substitute ⇒ Q by ¬ P ∨ Q).
- **CASE**: To prove a sequent with goal P, we should prove the sequent with hypotheses Q and the sequent with hypotheses ¬Q.

The initial theory is enlarged with the following inference rules:

$$\frac{}{H,\bot \vdash P}\ \textsf{FALSE_L} \qquad \frac{H,\neg Q \vdash P}{H,\neg P \vdash Q}\ \textsf{NOT_L} \qquad \frac{H,P,Q \vdash R}{H,P \wedge Q \vdash R}\ \textsf{AND_L}$$

$$\frac{H \vdash P \quad H \vdash \neg P}{H \vdash \bot}\ \textsf{FALSE_R} \qquad \frac{H,P \vdash \bot}{H \vdash \neg P}\ \textsf{NOT_R} \qquad \frac{H \vdash P \quad H \vdash Q}{H \vdash P \wedge Q}\ \textsf{AND_R}$$

$$\frac{H \vdash P}{H,\top \vdash P}\ \textsf{TRUE_L} \qquad \frac{}{H \vdash \top}\ \textsf{TRUE_R}$$

$$\frac{H,P \vdash R \quad H,Q \vdash R}{H,P \vee Q \vdash R}\ \textsf{OR_L} \qquad \frac{H,\neg P \vdash Q \vdash R}{H \vdash P \vee Q}\ \textsf{OR_R}$$

$$\frac{H,P,Q \vdash R}{H,P,P \Rightarrow Q \vdash R}\ \textsf{IMP_L} \qquad \frac{H,P \vdash Q}{H \vdash P \Rightarrow Q}\ \textsf{IMP_R}$$

$$\frac{H,Q \vdash P \quad H,\neg Q \vdash P}{H \vdash P}\ \textsf{CASE}$$

Predicate Logic Extension

The predicate logic extends the previous inference rules by adding variables, expressions, quantified predicates, and equality. Two new syntactic categories are introduced called expression and variable. The quantified predicate is written in the form of ◊x.P, where ◊ stands for the universal quantifier ∀ or the existential quantifier ∃, x stands for a non-empty list of variables, P stands for a predicate. The notation [x := E]P stands for the instantiation of the quantified variable x by expression E in the predicate P.

- **ALL_L**: If we can prove a sequent with a quantified predicate ∀x.P and an assumption [x := E]P in the hypotheses, then the sequent without the assumption [x := E]P is proved.
- **ALL_R**:To prove a sequent with a quantified predicate ∀x.P as its goal, it is enough to prove a similar sequent without quantifier ∀ in the goal
- **XST_L**: If we can prove a sequent with a predicate P in the hypotheses, then we can prove a sequent with an instantiation of variable x in P.
- **XST_R**: If a goal of a sequent is obtained by an instantiation of a quantified variable x in predicate P, then we say the sequent with a goal which says ∃x.P is proved.
- **CUT_XST**: This is a derived rule from **CUT**, **XST_L** and **XST_R**.
- **EQ_LR**: This inference rule applies the equality assumption from left to right in the remaining hypotheses and goal.
- **EQ_RL**: This inference rule applies the equality assumption from right to left in the remaining hypotheses and goal.

$$\frac{H, \forall x.P, [x := E]P \vdash Q}{H, \forall x.P \vdash Q} \textsf{ALL_L} \qquad \frac{H \vdash P}{H \vdash \forall x.P} \textsf{ALL_R}\ (x\ not\ free\ in\ H)$$

$$\frac{H, P \vdash Q}{H, \exists x.P \vdash Q} \textsf{XST_L}(x\ not\ free\ in\ HandQ) \qquad \frac{H \vdash [x := E]P}{H \vdash \exists x.P} \textsf{XST_R}$$

$$\frac{H \vdash \exists x.Q \quad H, Q \vdash P}{H \vdash \exists x.P} \textsf{CUT_XST}\ (x\ not\ free\ in\ H)$$

$$\frac{[x := F]H, E = F \vdash [x := F]P}{[x := E]H, E = F \vdash [x := E]P} \textsf{EQ_LR} \qquad \frac{[x := E]H, E = F \vdash [x := E]P}{[x := F]H, E = F \vdash [x := E]P} \textsf{EQ_RL}$$

3.3 Event-B Method

3.3.1 The Correct-by-Construction Approach

The Event-B method [9] is both a language and a method of specifying and proving computer software. It has recently been proposed by Jean-Raymond Abrial as an evolution of the classical B method [13]. The new method preserved the power and simplicity of the old, and it also brought improvements in several aspects, including the specification of reactive systems. The Event-B development process assumes that each specification is associated with a proof of correction based on the mathematical foundation of the language itself. It should be noted that the method is based on first-order logic and set theory.
However, the Event-B method is directly related to the program design approach: "correct-by-construction". This method ensures that when the formal development of the system is

completed that it is already verified. This approach consists in starting the development with an abstract model of the system and gradually adding details to generate at the end a concrete model very close to the implementation. Formally, this incremental construction of the system is guided by the refinement technique [103] that preserves the properties of the system, including correction and termination properties. The development process generates several proof obligations that guarantee its correction. These obligations are proven by verification tools that often incorporate automatic and interactive proofing procedures.

3.3.2 Event-B Model

Event-B models are organized in terms of two basic components: contexts and machines(Figure3.2).

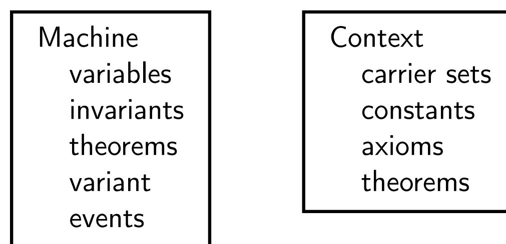


Figure 3.2 – Machine and Context

The Context is composed of carrier sets, constants, axioms, and theorems while the Machine is made up of variables, invariants, theorems, variants, and events. The components both in Contexts and Machines are all called modeling elements, and they correspond to the static and dynamic parts of a model, respectively.

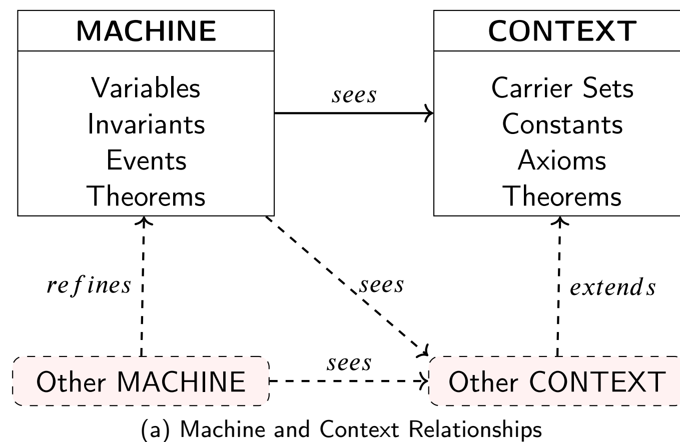
There are three kinds of relationships between components of an Event-B model as shown in Figure 3.3.

- A concrete machine can only "refine" at most one more abstract machine.
- A concrete context can "extend" zero, one, or several more abstract contexts.
- A machine can "see" zero, one, or several contexts.

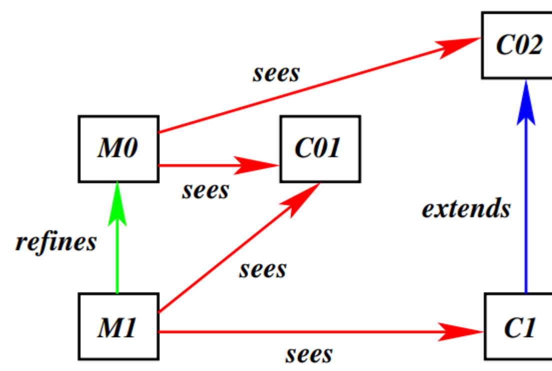
If a machine "sees" a context, then all the components like constants, sets, and axioms defined in the context and extended from other Contexts can be used by the machine.

3.3.3 Contexts

The first structure is called a context (Figure 3.4), and it provides the definition of the sets, constants, axioms for sets and constants, and theorems that can be derived from the axioms of the context Cx1. The context Cx0 is a previous context that has already been defined, and it extends the current context. A context is validated when sets $S_1,\ldots, S_n$, constants $C_1,\ldots,$ C_m and axioms $ax_1,\ldots, ax_p$are well-formed and when all theorems $th_1,\ldots, th_q$ are proved.



(a) Machine and Context Relationships



M0 sees C01 and C02 ***explicitly***

M1 sees C01 and C1 ***explicitly***

M1 sees C02 ***implicitly***

(b) Machine and Context Example [9]

Figure 3.3 – Machine and Context Relationships

3.3.4 Machines

The dynamic part of a model is expressed using the notion of the machine. An Event-b machine (Figure 3.5) is defined by a set of clauses. A machine is either a basic machine or a refinement of an abstract machine. Shortly, the clauses mean:

- **VARIABLES** clause represents the variables of the model of the specification. Refinement may introduce new variables in order to enrich the described system.
- **INVARIANT** clause describes, thanks to first order logic expressions, the properties of the attributes defined in the clause VARIABLES. Typing information and safety properties are described in this clause. These properties shall remain true in the whole

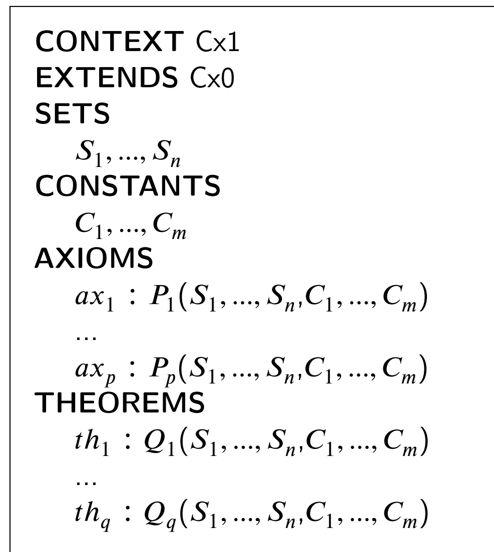


Figure 3.4 – Context Structure

model and in further refinements. Invariants need to be preserved by the initialization and events clauses.

- **THEOREMS** lists the various theorems which have to be proved within the machine. In order to prove a theorem, we assume the axioms and theorems of the seen contexts, the invariants and theorems of the abstract machines, the local invariants, and also the theorems which are written before the present one.
- **INITIALISATION** clause allows to give initial values to the variables of the corresponding clause. They define the initial states of the underlying transition system.
- **EVENTS** clause defines all the events that may occur in a given model. Each event is described by a body thanks to generalized substitutions defined below. Each event is characterized by its guard (i.e. a first order logic expression involving variables). An event is fired when its guard evaluates to true.

3.3.5 Events

Events are operations that update the state variables of a machine. Each event is composed of guard and action statements. An event is allowed to execute an operation whenever all its guard statements return true. Action statements define the behavior of the event operation and are required to update the state variables of the machine. We consider three kinds of events depending on when they are introduced during the development of a model:

- **Convergent**: new events introduced after a refinement. These kind of events refine skip and require a variant to ensure non-divergence.
- **Anticipated**: event declared in anticipation that does not need to decrease a variant

MACHINE M1
SEES Cx1
VARIABLES x
INVARIANTS
  $inv_1 : I_1(\text{x}, S_1, ..., S_n, C_1, ..., C_m)$
  ...
  $inv_r : I_1(\text{x}, S_1, ..., S_n, C_1, ..., C_m)$
THEOREMS
  $th_1 : A_1(\text{x}, S_1, ..., S_n, C_1, ..., C_m)$
  ...
  $th_s : A_s(\text{x}, S_1, ..., S_n, C_1, ..., C_m)$
EVENTS
  EVENT Initialisation
    BEGIN
      $x :\mid (P(x'))$
    END ...
  EVENT e
    ANY t
    WHERE
      G(x,t)
    THEN
      $x :\mid (P(x, x', t))$
    END ... **END**

Figure 3.5 – Machine Structure

(but must not increase it either); it only decreases the variant when it becomes convergent in a further refinement [9].

- **Ordinary**: neither convergent nor anticipated.

An event can be represented by the following form:

evt ≙ **any** p **when** G **with** W **then** S **end**

The short form

evt ≙ **begin** S **end**

where p denotes internal parameters of the event, G is a predicate denoting *guards*, and S denotes the *actions* that update some variables. S can be executed only when G holds. When we refine an abstract event, some variables of that event might be disappeared in its concrete event. W denotes *witnesses* that are additional elements in the concrete event for indicating the disappeared variables and their values. Given that the variables of the machine containing the event are denoted by v, S is composed of several *assignments* of the form:

$$x := E(v)$$
$$x :\in E(v)$$
$$x :| Q(v, x')$$

where x are some variables, and x' represents a state of x which their values are changed just after assigning x to an expression $E(v)$ or a predicate $Q(v, x')$. The former form is deterministic, which x are assigned to precise values. The latter form is non-deterministic assigning x to be elements of carrier sets. The effect of each assignments can also be described by a before-after predicate:

$$BA(x := E(v)) \triangleq x' = E(v)$$
$$BA(x :\in E(v)) \triangleq x' \in E(v)$$
$$BA(x :| Q(v', x')) \triangleq Q(v, x')$$

A before-after predicate describes the relationship between the state just before an assignment has been triggered and the state just after the assignment has been triggered (represented by x and x' respectively).

Table 3.3 – Before-After Predicate

Action	Before-after (BA) predicate	Explanation
$x := F(x, y)$	$x' = F(x, y) \wedge y' = y$	standard assignment
$x :\in Set$	$\exists t.(t \in Set \wedge x' = t) \wedge y' = y$	non-deterministic assignment from set
$x :| P(x, y, x')$	$\exists t.P(x, y, t) \wedge x' = t) \wedge y' = y$	non-deterministic assignment by given post-condition

The action of an event can have one of several forms of assignment, represented in Table 3.3. Where x and y are disjoint lists of state variables, and x', y' represent their values in the after state.

The $F(x, y)$ represents here a mathematical function that defines a new value for x (denoted by x') deterministically. The second part of the BA predicate requires all the remaining variables (y) should not change as a result of the assignment. The *Set* represents any defined set while $P(x, y, x')$ is a post-condition relating initial values of x and y to the final value x'. The :∈ and :| represent non-deterministic assignment operators operating on sets and predicates respectively.

Example: The Birthday Book

What follows is an example of a birthday book which uses both event parameters and functions. It is simple yet illustrative.

CONTEXT BirthdayBook_ctx
SETS PERSON DATE
MACHINE Birthday_Book

This models an agenda. It is not a reactive system, but it is rather the model of a data abstraction

SEES BirthdayBook_ctx

VARIABLES

bdayBook This will hold the actual data for the agenda: the date for every person

resdate The date of the birthday of some person

INVARIANTS

inv1 : bdayBook $\in$ PERSON $\nrightarrow$ DATE

Every person has only a date in the agenda

inv2 : resdate $\in$ DATE

EVENTS

Initialisation

begin

act1 : bdayBook := $\varnothing$

We start with an empty agenda

act2 : resdate :$\in$ DATE

We do not have any specific value to initialie resdate, so we use a nondeterministic assignment to avoid warnings

end

Event AddBirthday $\widehat{=}$

any

person Person we want to add to the agenda

date Date we want to associate to the person

where

grd1 : person $\in$ PERSON

It has to be a person

grd2 : date $\in$ DATE

It has to be a date

grd3 : person $\notin$ dom(bdayBook)

We do not allow repeated people in the agenda

then

act1 : bdayBook(person) := date

We associate person —-> date (overriding existing tuples if necessary; it should not happen in this example)

end

Event Update_Birthday $\widehat{=}$

We change the birth date for an existing one

any

person As in the previous event

date Id

where

grd1 : person $\in$ dom(bdayBook)

The "type" of person can be inferred from the domain of bdayBook

grd2 : date $\in$ DATE

But we have to state the type of the date

then
act1 : bdayBook(person) := date
Equivalent to bdayBook := ({person} ⩤ bdayBook) ∪ person ↦ date
end
Event Check_Birthday ≙
any
person The person whose birthday we want to know
where
grd1 : person ∈ dom(bdayBook)
the person has to be in the agenda
then
act1 : resdate := bdayBook(person)
We return the stored date
end
END

We separately define an Event-B abstract model and an Event-B concrete model to clearly distinguish them in the section 3.7. According to [104], we can define an Event-B abstract model (refined model) as follows:

Definition 1 (*Event-B abstract model*) *: An Event-B abstract model is a tuple (s, c, A, v, I, E), where s and c are the carrier sets and constants respectively; A(s, c) is a collection of axioms; v are the machine abstract variables; I(s, c, v) is the invariants limiting the possible state of v; E is a set of events. Moreover, each abstract event is defined as a tuple (G,BA), where G(s, c, v) is the event guard triggering actions when it holds; BA(s, c, v, v') is a before-after predicate defining a relation between the current and next states, and representing event actions; and v' is a next state of v.*

An Event-B concrete model (refining model) can be defined with respect to an Event-B abstract model as follows:

Definition 2 (*Event-B concrete model*) *: An Event-B concrete model is a tuple (s, c, A, v, w, J,E2), where w are concrete variables; J(s, c, v, w) is the invariants added in the concrete machine; E2 is a set of concrete events. Moreover, each concrete event is defined as a tuple (H,W,BA2), where H(s, c,w) is the concrete event guard; W(s, c, v, w, v',w') are witnesses for disappearing abstract variables; BA2(s, c,w,w') is a before-after predicate defining a relation between the current and next states, and representing event actions; and w' is a next state of w.*

Note that axioms, invariants, guards, witnesses and actions (before-after predicates) are predicates, while carrier sets, constants, and variables are arguments of those predicates.

3.4 Refinement in Event-B

Refinement is a top-down development method and is at the core of Event-B modeling. We start by specifying the system at an abstract level and gradually refine by adding further details

in each refinement step until the concrete model is achieved. A refinement is a developmental step guaranteeing every behavior in the concrete model is one specified in the abstract model. It usually reduces non-determinism and each refinement step must be proved to be the correct refinement of the abstract model by discharging suitable refinement POs.

In Event-B, refinement has various aspects or classification. Refinement of a machine, from Event-B notation perspective [34, 103, 105], consists of:

1. Refining existing events:
   (a) Adding new parameters, guards and actions to the existing abstract event: in this case, the resulting concrete event is labeled as extended. In an extended event, the existing parameters, guards, and actions cannot be modified.
   (b) Modifying parameters, guards, and actions of the existing abstract event: in this case, the resulting concrete event is labeled as not− extended (refine). Adding new parameters, guards, and actions are allowed, as well.

   In both types, the guards of the concrete event must be proved to be stronger than its abstraction (guard strengthening).
2. Adding new events

   The new event refines a dummy event in the abstraction, which does nothing (skip). The new event does not diverge. It means that it should not take control forever. The new event can be marked as:

   - Convergent: Each convergent event requires a variant to ensure non-divergence.
   - Anticipated: Events that will be introduced in a future refinement, but are declared in anticipation.
   - Ordinary: None of the others and the most commonly used.
3. Add new variables and invariants:

   Introducing new variables usually results in case (2) or case (1.a) of refinement. Sometimes, abstract variables can be replaced by new concrete variables. In this situation, the refinement can result in (1.b). Variable replacement is called data refinement.

   A gluing invariant connects the abstract variables to the concrete variables. In other words, it glues the state of the concrete model to that of its abstraction. The invariant of the concrete model, including gluing invariants should be preserved for every event.

Each abstract event should be refined by at least one concrete event. One abstract event can be refined by more than one concrete event. This is called event splitting. Also, one concrete event can refine more than one abstract event. This is called event merging.

Typically, we classify the refinement into two forms, horizontal and vertical refinements: The difference between horizontal and vertical refinement is methodological and it can help with the process of modeling. However, there is no difference in terms of proof obligations of these two styles of refinement.

Horizontal Refinement

Horizontal refinement, also known as superposition refinement [106, 107] , is the term used when we refine a model by adding new variables and events to the existing model. To ensure provable correctness, these newly added variables and events must not contradict any previous, more abstract model.

Vertical Refinement

Vertical refinement, or data refinement [108], is the term used when we replace an abstract representation of data with a more concrete one, and change the events accordingly. In this case, we need to add gluing invariants, which formally define the relationship between the previous, abstract variables and the newly introduced, concrete ones. The gluing invariants are invariants of a refined machine that refer to variables of the abstract one [34, 109]. The gluing invariant is expressed in terms of a predicate *P(v,w)* connecting the state variables of the abstract machine *(v)* and the corresponding state variables of the concrete machine *(w)* [34].

3.5 Decomposition

As long as a model is more and more refined, the number of its state variables and transitions may increase in such a way that it becomes impossible to manage the whole. That's the way it becomes necessary to split the single refined model into several loosely coupled pieces. Decomposition splits the model into separate components that model individual parts of the system. The user can then further develop the components, concentrating on each component separately. An Event-B model can be decomposed either event based on variable based decomposition.

3.5.1 Shared Variable Decomposition

Shared variable decomposition presented in Figure 3.6 is proposed by Abrial, Metayar and Hallerstede [104, 110, 111]. Machine M is decomposed into machine M1 and M2. The shared variable decomposition does not permit events sharing and a variable can be split into different sub-models, this variable is called a shared variable. First, the events of M are partitioned among M1 and M2. Then the variables of M are distributed according to the event partition. v1 and v3 are private variables, since they are accessed by events of only one sub-model, E1 in M1 and E4 in M2 respectively. v2 is a shared variable which is accessed by event E2 in M1 and E3 in M2. so they share variable var2. However, since sub-models have in fact two copies of the shared variable, they need to learn the changes made to the shared variable in the other sub-models. This is implemented adding so-called external events. The external event of E2' is built in M2, since E2 modifies the shared variable v2 in M1. The invariant distribution is done according to variable distribution. An invariant belongs to a sub-model if all variables used in that invariant belong to that sub-model.

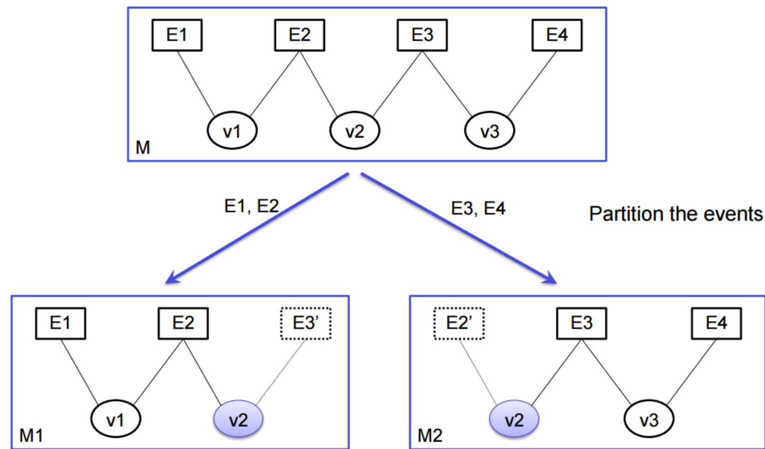


Figure 3.6 – Shared-Variable Decomposition Style

It is worth mentioning that the shared variables decomposition is suitable for the specification and verification of parallel programs [112].

3.5.2 Shared Event Decomposition

Shared event decomposition (Figure 3.7) is introduced by Butler [113], inspired by the synchronous parallel composition of a process. In this decomposition style, the Event-B model is decomposed into various sub-models such that all its events and variables are distributed over the local models.

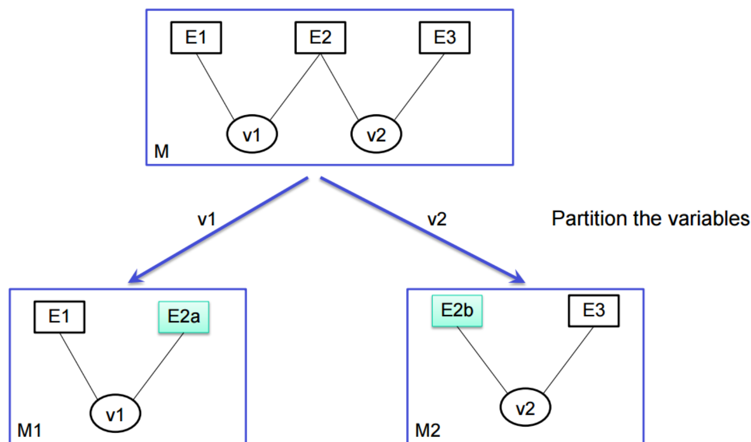


Figure 3.7 – Shared-Event Decomposition Style

As the name suggests, the local sets of events may have common events (shared events). However, the local sets of variables are disjoint, i.e. the partition of the variables will determine the structure of the decomposition. Figure 3.7 presents an example of shared events decomposition. Variables of the machine M are partitioned among the sub-models, M1 and M2. After the variable partition, it is necessary to split the events according to the variable

partition. Events using variables allocated to different sub-models, E2 uses v1 from M1 and v2 from M2, are called shared events and must be split. Part of the shared event, which is corresponding to each variable, E2a and E2b, is used to build sub-model events. Invariant distribution is similar to shared variable decomposition.

3.5.3 Shared Event Decomposition with Shared Parameters

It is possible for guards and actions with disjoint variables to have common parameters. As an example to decompose an event such as M in figure 3.8, two sub-events can be constructed where one refers to the variable v and the other to w. In this case, former event has the action act1 and the latter contains act2. However, both events share the parameter i.

Event **M** $\hat{=}$
any **i**
where
  grd1: $0 \leq i \leq v$
  grd2: $w < \mathbb{N}$
then
  act1: v := v - i
  act2: w := w + i
end

Figure 3.8 – An Event with Shared Parameter[114]

Figure 3.9 shows the decomposition of M to two events which share the parameter i. This parameter is constrained by grd1 in the left sub-event,while it has a loose typing guard in right sub-event. This indicates that i is an output in the event M on the left, whereas it is an input in the other event.

Event **M** $\hat{=}$
any **i**
where
  grd1: $0 \leq i \leq v$
then
  act1: v := v - i
end

Event **M** $\hat{=}$
any **i**
where
  grd1: $i \in \mathbb{Z}$
  grd2: $w < \mathbb{N}$
then
  act1: w := w + i
end

Figure 3.9 – Decomposition of an Event with Shared Parameter[114]

3.6 Generic Instantiation

Generic instantiation in Event-B was introduced in [111] and further elaborated in [115]. The defined model using refinement and decomposition can be defined by using carrier sets (a set containing the elements of a particular structure as opposed to a relation) and constants. This generic model can then be developed with some mathematical theory, such as set or group theory. The interest of this approach of generic instantiation is that it saves us redoing the proofs already done in the abstract development.

3.7 Proof Obligation Rules

When an Event-B model is created or refined, a set of proof obligations must be discharged in order to guarantee certain properties of a model. Proof obligations have two-fold purpose [116]:

- They show that a model is sound with respect to some behavioral semantics.
- They serve to verify the properties of the model.

The proof obligations define what is to be proved for an Event-B model. These proofs concern invariant preservation, Feasibility, Fusion,. . . . They are automatically generated by RODIN platform tool called the proof obligation generator, just to check contexts and machines texts and decide what is to prove in these texts, there are eleven rules for the proof obligation all defined and labeled inside the RODIN platform.

This section defines some of the most common PO rules. In order to define rules dealing with an event, we will use the general definition of an event defined as follows:

```
evt
   any x where
     G(s, c, v, x)
   then
     v :| BA(s, c, v, x, v')
   end
```

Where s denotes the seen sets, c the seen constants, v the variables of the machine, x the abstract parameters, and v' the substituted variables. The notation BA(s, c, v, x, v'), called the before after predicate which describes the state change after the actions occur. The axioms and theorems are denoted by A(s, c), whereas invariants and local theorems are denoted by I(s, c, v), the concrete variables In a refining machines will be denoted by w and the local invariants and theorems by J(s, c, v,w).

Table 3.4 contains a list of important proof obligation in Event-B modeling.

Table 3.4 – Proof Obligations in Event-B

Invariant Preservation	*evt* / *inv* / INV	*evt* is the event name, *inv* is the invariant name
Well definedness	*x* / WD	*x* is the name of axiom,theorem, invariant, guard, action, variant
Feasibility of a non-deterministic event action	*evt* / *act* / FIS	*evt* is the event name, *act* is the action name
Guard Strengthening	*evt* / *grd* / GRD	*evt* is the concrete event name, *grd* is the abstract guard name
Action Simulation	*evt* / *act* / SIM	*evt* is the concrete event name, *act* is the abstract action name
Natural number for a numeric Variant	*evt* / NAT	*evt* is the new event name
Decreasing of Variant	*evt* / VAR	*evt* is the new event name

3.7.1 Invariant Preservation Rule: INV

This proof obligation rule ensures that each invariant in a machine is preserved by each event. The INV rule is written as the following sequent:

$$
\begin{array}{l}
A(s, c) \\
\; I(s, c, v) \\
\; G(s, c, v, x) \\
\; BA(s, c, v, x, v') \\
\vdash \\
\; inv(s, c, v')
\end{array}
$$

where inv(s, c, v´) is the modified specific invariant.

3.7.2 Feasibility Rule: FIS

Feasibility proof obligation ensures that a non-deterministic action is feasible. The FIS rule is written as the following sequent:

$$
\begin{array}{l}
A(s, c) \\
\; I(s, c, v) \\
\; G(s, c, v, x) \\
\vdash \\
\; \exists v'.BA(s, c, v, x, v')
\end{array}
$$

3.7.3 Guard Strengthening Rule: GRD

The guard strengthening rule makes sure that the guards in the refined event are stronger than those in the abstract event. This ensures that when a concrete event is enabled the corresponding abstract one is also enabled.

$$\begin{array}{l} \quad A(s, c) \\ \quad I(s, c, v) \\ \quad J(s, c, v, w) \\ \quad H(y, s, c, w) \\ \quad W(x, s, c, w, y) \\ \vdash \\ \quad g(s, c, v, x) \end{array}$$

where W(x, s, c, w, y) is the witness predicate, in which the abstract parameters x and y are different.

3.7.4 Simulation Rule: SIM

This PO verifies that the actions of a refined event simulate the same behavior than the abstract event it refines. When a concrete event executes, the corresponding abstract event is not contradicted.

$$\begin{array}{l} \quad A(s, c) \\ \quad I(s, c, v) \\ \quad J(s, c, v, w) \\ \quad H(y, s, c, w) \\ \quad W1(x, s, c, w, y, w') \\ \quad W2(v', s, c, w, y, w') \\ \quad BA2(s, c, w, y, w') \\ \vdash \\ \quad BA1(s, c, v, x, v') \end{array}$$

The BA1 and the BA2 stand for the before after predicates of the variables in the abstract event and in the concrete event respectively.

3.7.5 The Numeric Variant Rule: NAT

This rule ensures that under the guards of each convergent or anticipated event, a proposed numeric variant is indeed a natural number.

A(s, c) I(s, c, v) G(s, c, v, x) $\vdash$ $n(s, c, v) \in \mathbb{N}$

3.7.6 The Variant Rule: VAR

This rule ensures that each convergent event decreases the proposed numeric variant or proposed finite set variant. The proof obligation rule is the following if the variant n(s, c, v) is numeric or the variant t(s, c, v) is a finite set:

A(s, c) I(s, c, v) G(s, c, v, x) BA(s, c, v, x, v´) $\vdash$ $n(s, c, v') \leq n(s, c, v)$	A(s, c) I(s, c, v) G(s, c, v, x) BA(s, c, v, x, v´) $\vdash$ $t(s, c, v') \subseteq t(s, c, v)$

3.7.7 Well-Definedness Rule

The well-definedness rule (WD) ensure that all the axioms (axm/ WD), theorems (thm/ WD), invariants (inv/ WD), guards (grd/ WD) and actions (act/ WD) are well-defined.

3.8 RODIN Platform

The RODIN platform [25] is an Eclipse-based [2] integrated development environment for Event-B. It provides specifiers with an effective support for editing the specifications, refining the models, generating and discharging the POs. The platform is an open source and extendable with external plug-ins[3].

The RODIN tool chain is composed of three major components:

- **The Static Checker (SC)**: analyses a model developed in Event-B in order to find syntax and type errors.
- **The Proof Obligation Generator (POG)**: automatically generates the POs that must be verified for a given Event-B model – the different types of POs associated with Event-B was described in Section 3.7. The POG does not perform proofs, it only carries out simple rewritings within a PO sequent.

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

[3] The complete list of past and current plug-in developments can be found at http://wiki.event-b.org/index.php/Rodin_Plug-ins

- **The Proof Obligation Manager (POM):** handles the POs' status as well as the associated proof tree for each PO. It works automatically alongside the automatic RODIN provers or interactively with the user and external provers – since all POs are represented as sequents in predicate calculus, different external provers for predicate calculus can be used through RODIN.

The use of Rodin is extremely simple. The verification process with "Rodin" is illustrated in Figure 3.10: an Event-B model specified by contexts, machines, "refines" refinement relations, "extends" extensions between contexts, etc. The verification principle is carried out by going through two stages: the first consists of verifying the specification of the model introduced. In case of errors (1'), the specification must be corrected (1'.2). If everything is verified (1), Rodin proceeds to the second step of generating the necessary proof obligations (WD, INV, THM, etc.). Some of these obligations can be proved by inference engines introduced to Rodin with various powers. These engines, called automatic provers, make it possible to deduce the implication associated with the obligation of proof. In this case, the proof is called automatic proof (1.1'). However, Rodin's provers are sometimes unable to deduce this implication (1.1). In this case, this proof is called interactive proof which must make interact the skills of the user (1.2). The latter can thus guide the proof process by recalling hypotheses already introduced or by using other previously verified properties (1.3). In some cases, the proof error comes from the specification logic, which requires the user to review his model and modify it if necessary (1.3').

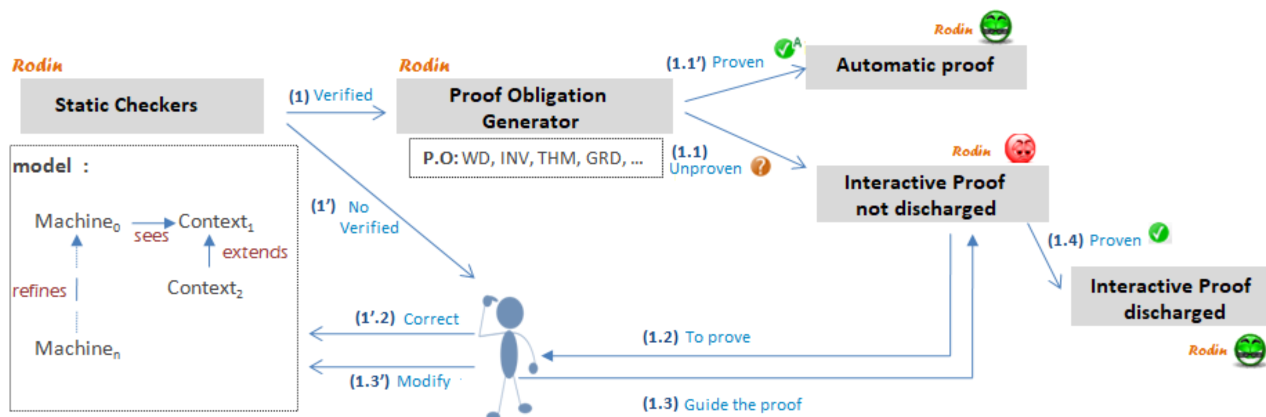


Figure 3.10 – Rodin Verification Process

We present some important windows as follows:

- **Proving Perspective (Figure 3.11):** It provides all proof obligations which are automatically generated for Event-B machines. These proof obligations can be discharged automatically or interactively with hypotheses and goal windows.
- **Event-B Perspective (Figure 3.12):** This perspective includes windows which allow us to edit Event-B machines and contexts. If users encode incorrectly, problem windows will show the error's content.

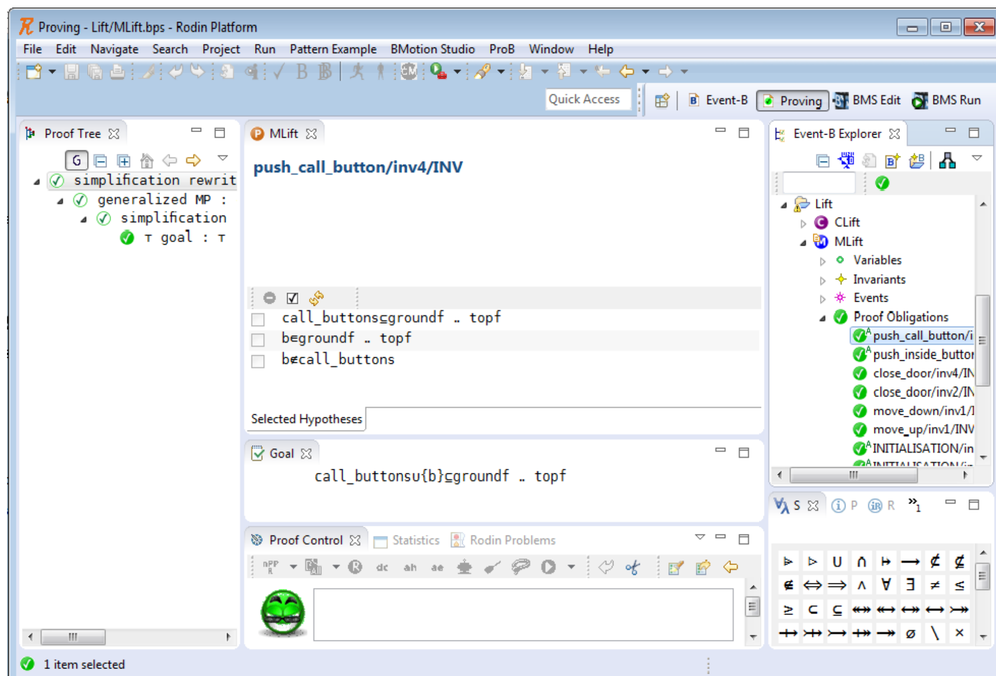


Figure 3.11 – The Proof Obligation Perspective: on the left, it is shown the proof tree of the selected PO, on the middle, on the top window are the hypotheses of the selected PO and just below the respective goal. Below the goal window are the buttons used to interactively discharge a PO, on the right, are the list of generated POs. Having all the POs green, it means that all the POs are discharged

We mention some of the plug-ins available in RODIN which illustrate different aspects of the toolset that have been extended :

- Graphical modeling: The UML-B [117] and iUML-B plug-ins provide a graphical modeling interface la UML on top of the RODIN platform. The graphical models are translated to plain Event-B machines and can be worked with like any Event-B model (proof, animation, etc.). These plug-ins make it particularly easy to model state-machines.
- Decomposition: This plug-in [37] is used to decompose an Event-B model into two communicating models able to be later refined independently. Two distinct approaches are provided: one is organized around some shared variables while the other is organized around shared events.
- ProB [36]: provides animation and model checking capabilities for Event-B models.
- ProR [118, 119]: provides requirement traceability between an Event-B model and the natural language requirements associated to the model.
- Atelier B provers [4]: provide additional capabilities of automated theorem proving (not contained in the RODIN provers).

[4]http://www.atelierb.eu/en/

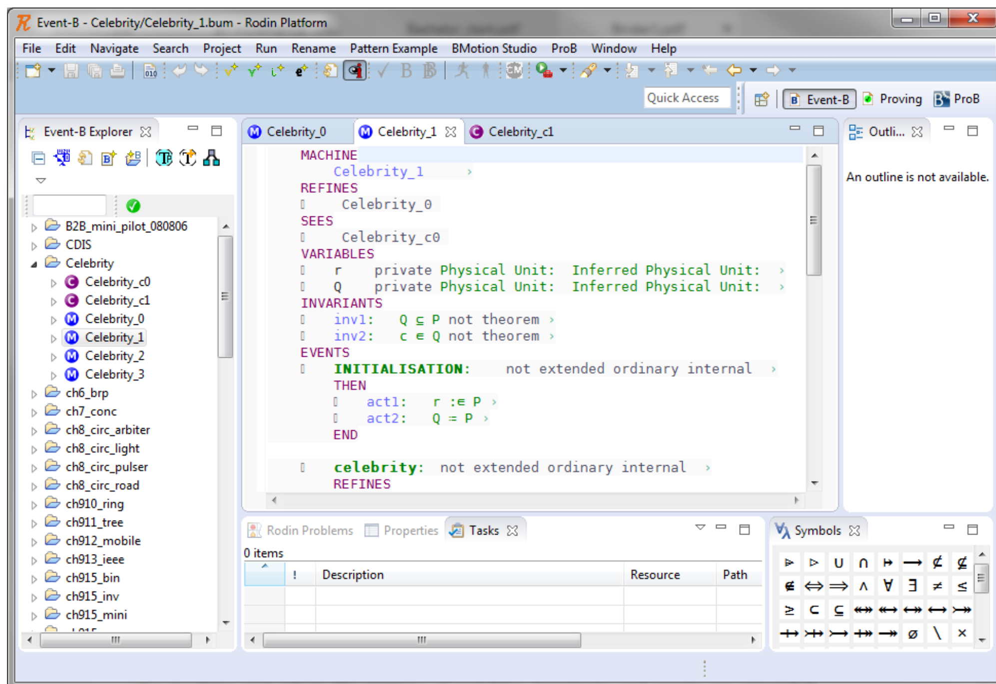


Figure 3.12 – The Event-B Perspective: on the left, the list of projects where the Celebrity project is expanded, showing several machines and a context, in the middle window, a view of a machine Celebrity_1 where the sections of variables, invariants and events can be edited.

- B2Latex[5]: exports Event-B models as Latex documents.

3.9 A Comparison

Z, VDM, and B are state-based formalisms in which a system is modeled by explicitly giving the definition of states and operations. Operations have an effect of transforming the system from a state to another state. While the methods have their own structure of specification, they still have some parts which are alike in purpose. For example, they all have an operation part and state variables.

A main difference between Event-B and the B method is that Event-B allows a richer notion of refinement in which new observables may be introduced in refinement steps. This means that complex interactions between subcomponents may be abstracted away in modeling at an early stage and then incrementally introduced through refinement

In [113], it is mentioned that Event-B refinement is more general than classical B and other related languages including Z and VDM. The ability to introduce new events in a refinement step is a major feature of Event-B. Event-B refinement supports the decomposition of an atomic event and also the decomposition of a machine. Compared with other state-base approaches discussed in Chapter 2, Event-B also uses generalized substitutions as a mechanism

[5] http://wiki.event-b.org/index.php/B2Latex

to transform a system state to another state like B. Event-B supports both concurrent and communication systems. Proof is a methodology used for verification of Event-B models similar to classical B.

3.10 Summary

Event-B is a formal method for modeling and analysis. It uses set theory as a notation, refinement for incremental specification development and technique of theorem proving to verify the model. In this chapter, we have discussed Event-B in details and described its structuring mechanism i.e. context and machines, have been explained. In addition, the elements of an event have been described in detail. Its process of refinement, its proof system, as well as the shared-variable and the shared-event decomposition techniques in Event-B, and finally its tool support, RODIN along with a number of plugins which facilitate the process of modeling.

4 Communication Protocols Overview

Communication protocols specify the set of rules required to exchange messages between communicating entities. Networked and distributed systems, built around communicating protocols, are widely used nowadays. Since such systems are often deployed in safety-critical applications, confidence in protocol correctness is highly required.

4.1 OSI Reference Model

A good starting point for discussing communication protocols is the Open Systems Interconnection.

Open System Interconnection (OSI) Model or also known as the OSI model, is divided into seven layers, as shown in Figure 4.1, is simply a way of sub-dividing the communications system into small parts called layers, to reduce the design complexity of a network system.

These layers in details – starting from the lowest – are the following:

1. **Physical layer:** This layer realizes the transfer of bit sequences using the physical communication medium. It defines the type of communication medium (such as a copper wire, a coaxial cable, an optical fiber, or a radio channel) and the physical and electrical specifications of this medium (such as voltages, cable specifications and pin layout.).
2. **Data Link layer:** The objective of this layer is to assure reliable data transfer between network entities by restricting the limits of the physical connection. It provides error detection and correction for the errors thatmay occur in the physical layer. This layer fragments the incoming data into data frames, transmits them in sequence, processes the acknowledgement frames sent by the receiver and realizes flow control.
3. **Network layer:** This layer performs typical network functions, such as routing and addressing. It selects the path between the source and the destination points and routes the data through this path. It provides transparent flow control for the transport layer.

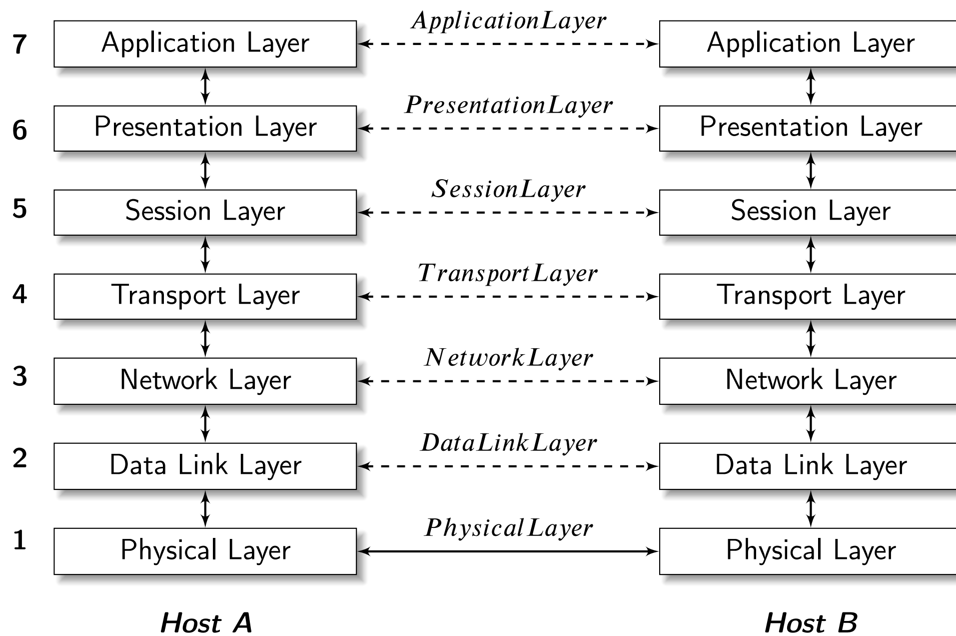


Figure 4.1 – ISO Reference Model for Open Systems Interconnection

4. **Transport layer:** This layer receives data from the session layer, splits them, and transmits this split data to the network layer. It ensures a transparent connection between the end users. It optimizes the usage of resources by the segmentation and de-segmentation of network connections while it hides the details from the users.

5. **Session Layer:** This layer controls the communication between the application processes. It supports their connections and data exchange.

6. **Presentation Layer:** This layer deals with the syntax of the transferred data, i.e., it transforms the different local syntaxes that are used in a heterogeneous system into a standard form that can be interpreted by any participant of the system.

7. **Application Layer:** This layer is the closest one to the end users: it provides an interface to them for reaching the OSI environment.

The structure of the OSI architecture is presented in the figure above, which indicates the protocols used to exchange data between two hosts A and B. The figure displays the bidirectional information flow; information in either direction passes through all seven layers at the end points. When the communication is via a network of intermediate systems, only the lower three layers of the OSI protocols are used in the intermediate systems.

The communications engineer is concerned mainly with the protocols operating at the bottom four layers (physical, data link, network, and transport) in the OSI reference model. These layers provide the basic communications service. The layers above are primarily the concern of computer scientists who wish to build distributed applications programs using the services provided by the network.

4.1.1 Formal Methods for OSI Layer Protocols

This section describes the application of formal methods in OSI layer protocols:

In 1983 [120]: Three formal specifications, using extended Petri nets, of the OSI transport layer protocol, and formal verification of these specifications using the PROTocol Emulation and ANalysis (PROTEAN) [121, 122] and OGIVE/OVIDE [123] analysis tools for Petri nets. Various general and specific properties have been checked, and no harmful error was found.

In 1986 [124]: Formal analysis of — a slightly simplified version of — the OSI session layer protocol, which was described using finite state machines communicating by bounded the First in, First out (FIFO) queues and verified using automated protocol validation techniques based on state-space exploration [125–127]. Various errors have been found, which were reported to standardization bodies and corrected in subsequent versions of the session layer.

In 1988 [128–130] : In the context of the OSI standardization initiative, formal methods (at that time called "formal description techniques") have been promoted as a means to define communication standards in a concise, unambiguous, implementationneutral way [131, 132]. In particular, the LOTOS language [133] has been used intensively to specify the service and protocol of the session layer, the service and protocol of the transport layer, the service and protocol of the network layer, and, at the application layer, the Remote Operations Service Element (ROSE) service, the Commitment, Concurrency and Recovery (CCR) service and protocol, and the Distributed Transaction Processing (DTP) protocol — thus demonstrating that formal techniques such as algebraic data types and process calculi could handle large, complex specifications.

4.2 Basics of Communication Protocol

When computers communicate with each other, there needs to be a common set of rules and instructions that each computer follows. A specific set of communication rules is called a protocol.

- **Protocol:** A set of rules and regulations is called a protocol.
- **Communication:** Exchange of information from one system to another system with a medium is called a communication.
- **Communication Protocol**: A set of rules and regulations that allow two electronic devices to connect to exchange the data with one and another.

Because of the many ways computers can communicate with each other, there are many different protocols – too many for the average person to remember. Some examples of these different protocols include Point to Point Protocol (PPP), TCP/IP, Serial Line Internet Protocol (SLIP), HyperText Transfer Protocol (HTTP), and File Transfer Protocol (FTP).

The essential function of protocols is to provide communication services among the users. Protocols are the basic elements of network operation, providing message exchanges among the

network nodes. Every protocol is used for communication purposes. During the transmission, messages may be damaged, or other problems may arise. The protocols often provide mechanisms to detect and correct the data transfer errors, like:

- A frame is lost
- A frame is doubled
- A frame arrives at a node with bad address
- The order of frames changes.

4.2.1 Communication Problems

In a fully automated system, the model of communication must be unambiguous and consistent. Some problems may occur when these conditions are not satisfied: these are infinite loops, deadlocks, and livelocks [134].

An infinite loop is a situation when a process is waiting for a condition that will never be achieved. A real time example of an infinite loop is: A daily calendar. There is a single page written PTO (please turn over) on both the sides. Note that if we followed the directive word-for-word we will never stop turning over, since there is no instruction to stop.

The deadlock is an interoperability problem between the participants of the communication. To have a clear idea of the deadlock, consider the following example: Alice and Bob happen to want to make a sandwich at the same time. Both need a slice of bread, so they both go to get the loaf of bread and a knife. Alice gets the knife first, while Bob gets the loaf of bread first. Now Alice tries to find the loaf of bread and Bob tries to find the knife, but both find that what they need to finish the task is already in use. If they both decide to wait until what they need is no longer in use, they will wait for each other forever. Deadlock. Consequently, the deadlock is a situation, where two or more participants are waiting for each other. A familiar example in communication systems is when the receiver asks the sender to wait for the next transmission and some time later, it notifies the sender to continue the transmission, but this notification is lost. If no reliability functions are implemented to recover this lost notification, then the receiver waits for the sender and the sender waits for the receiver forever.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. A real-world example of livelock occurs when two friends (Bob & Jack) are walking together to a coffee shop. When they reach the door Bob stops, opens the door and offers Jack to walk in. Rather of walking in, Jack tells Bob, "No you." Bob then says, "No you." This is a classic livelock example, where one signal effects the other and no progress is made because neither Bob nor Jack takes initiative. The risk of livelock occurs when we would like to resolve a deadlock and force more than one participant to change. To prevent this, we must force only one participant of a deadlock to take an action.

4.2.2 Traditional Communication

The client/server model is one of the network architectures to connect computers or applications, where the server provides resources and the client requests for the resources. Usually, the client/server model is designed as a request/response model, where the server does not actively communicate with clients. The client could be a computer, or an application, such as a web browser. The server is, usually, a (powerful) computer that keeps some applications running as constantly as possible. Those applications could be a database, a web server (application), or a printer server (application) that links to a physical printer.

The protocols presented in this thesis are request/response protocols, where a client communicates with a server through a request and a response message(s). In other words, a client retrieves resources from a server in a request message, while the server hands over the contents in a response message. This is a general model that a client retrieves data from a server.

4.2.3 A Simple Message Exchange Protocol

Consider a simple message exchange protocol; in Figure 4.2; where the two users exchange messages through a server. This protocol is based on a master-slave relationship between the source and destination and the exchange of messages takes place in a half-duplex mode.

In any protocol, we have to consider two parts. The first one is concerned with the synchronization between the two entities at the "process level". This covers the exchange of messages which has to happen before they are able to send data. This exchange is often known as the "control phase". The second part is concerned with the synchronization between the two entities at "data level". This covers the message exchange which happens when data are transferred and which is sometimes named the "data phase".

For the protocol of Figure 4.2, the pair request to send and ready to receive is concerned with the synchronization at the process level. The pair data sent and data consumed is related to the synchronization at the data level.

From the flowchart of the source, it is clear that the two events directly related to the protocol are the arrival of the two messages ready to receive and data consumed.

4.3 Protocol Development

The basic approach of the methodology [135] is to form the system requirements into an initial protocol specification, then refine it into successively more implementation oriented specifications. Ultimately, it is refined into an implementation, and there is an iterative feedback after each refinement. The initial protocol specification can be used as a reference for testing the final implementation. There have been many variations of the software engineering "waterfall model". put forward for protocol engineering. A methodology for OSI protocol development (Figure 4.3) is described below.

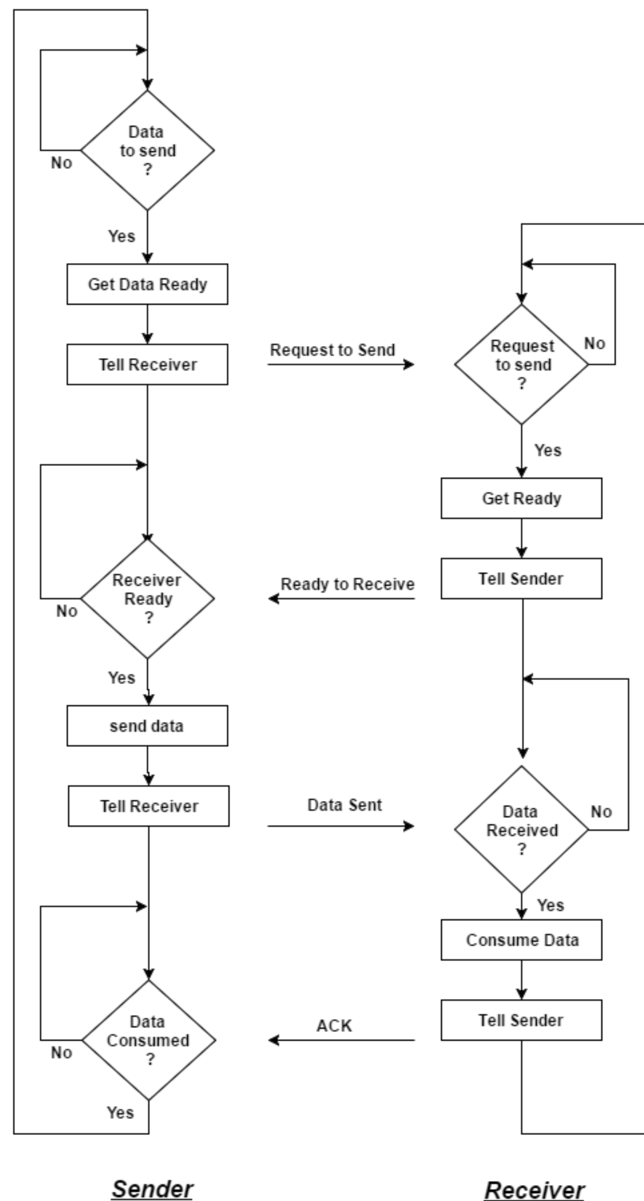


Figure 4.2 – A Simple Example of Protocol

- **Informal Specification:** The requirements of the users are studied. The desired communication services are then specified in a plain language. Some of the details include protocol model, services provided, reliability, and handling of errors.
- **Formal Specification:** It is the specification using a formal language, for example, mathematics. This enables the specifications to be analyzed
- **Protocol Verification:** Based on the formal specification, it is to identify errors early in the development process. Should the error be detected, the specification is iterated.
- **Implementation development:** It is the development of the communication software

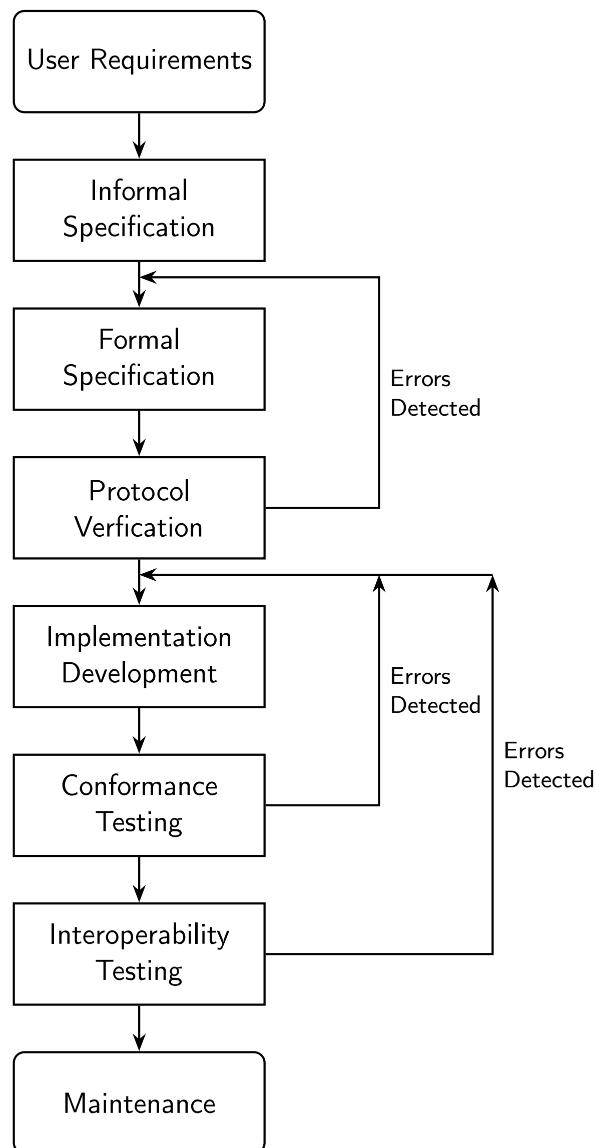


Figure 4.3 – A Protocol Development Methodology

according to specifications. The development of software can be manual or automatic.

- **Conformance testing:** It is the validation of the protocol implementation by testing. The protocol is fed with a selected set of test input, and the output generated are observed and checked against specifications.
- **Interoperability Testing:** It is the validation of the protocol implementation using implementations from different manufacturers. This ensures the protocol achieves the purpose of open communication.

4.3.1 Protocol Engineering

Protocol engineering is the scientific methodology supporting protocol design. There has been a long-standing common history between formal methods and protocol design [53]. From the beginning, formal methods have been a core part of protocol engineering, and protocols have been a key application target for formal methods. This convergence of interests enabled major advances in theory and practice, a cross-fertilization that appears in several books, e.g. [136, 137], in which protocol and formal aspects are intertwined. It has become standard practice to use formal description and verification techniques when designing new protocols; the same holds for cryptographic protocols too.

4.3.2 Protocol Specification

The design of any systems should be started by specifying the basics of the system's operation. The specification should then be elaborated step-by-step, in an increasingly more detailed way(see figure 4.4) [138]

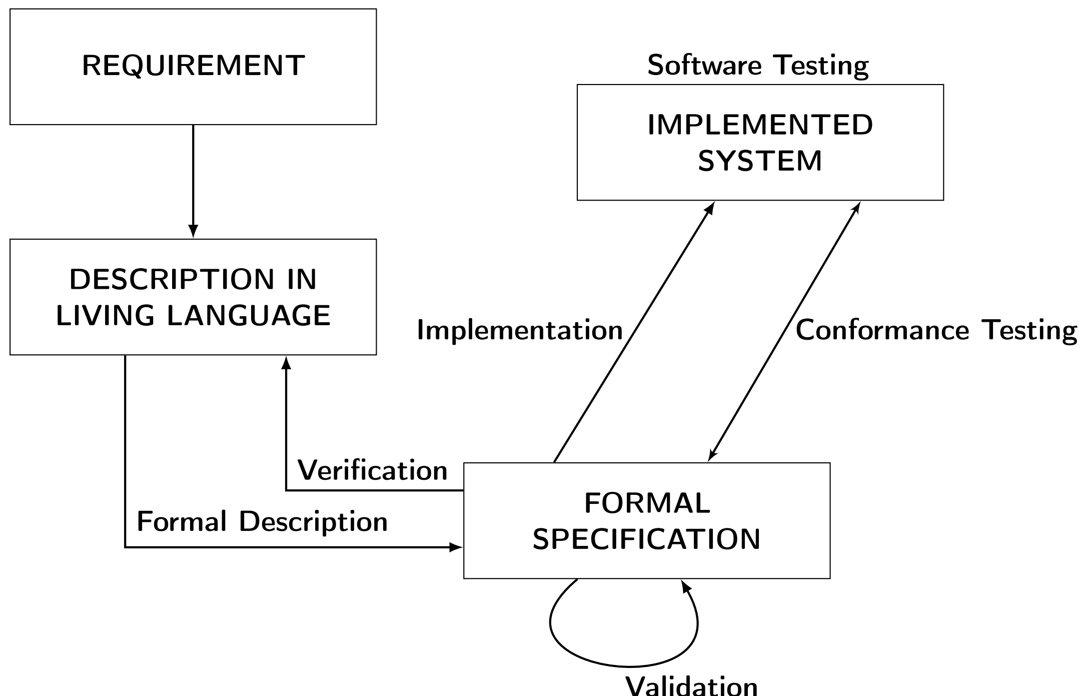


Figure 4.4 – Conformance Testing, Protocol Description and Implementation

The dynamic operation rules of the computer networks are described by the protocols, therefore it must be specified. Specifying the protocols means to determine the protocol definition built upon its formal model [139],[140].

The specification of a protocol includes the definition of message identifiers and message structures. The message name can be mapped to one of four primitives that identify the phase of the interaction:

- A sends a message to B – this is a request message
- The request of A arrives at B – this is an indication message
- B acknowledges the receipt of the message – this is a response message
- The response of B arrives at A – this is a confirmation message

4.3.3 Protocol Verification

Verification is based on the system specification and involves logical reasoning. Therefore it may be used during the design phase before any system implementation exists, in order to avoid possible design errors. While testing and simulation only validate the system for certain test situations, verification allows, in principle, the consideration of all possible situations the system may encounter during actual operation.

Protocol verification is a process of checking whether the interactions of protocol entities, according to the protocol specification, do indeed satisfy certain properties or conditions which may be either general (e.g., the absence of deadlock) or specific to the particular protocol system directly derived from the specification. The verification is less general and more tightly connected to protocol design than the specification.

4.3.4 Protocol Testing

Protocol testing means the control of the protocol that is already in operation. It has two types:

- Conformance testing. This tests whether the characteristics of the implemented protocol conform with those given in protocol specification
- Performance measurement. This determines how fast and how reliable can information transfer be provided by the protocol in operation. in the case of different loads [141, 142].

Testing and Test Control Notation (TTCN) is one of the standardized test languages in the field of telecommunication, and the latest version of TTCN is the third[143, 144]. An important test type is conformance test [145]. It checks if the protocol confirms a given standard.

4.4 Summary

The increasing significance of fast and reliable communication has resulted in the creation of certain communication rules, which nowadays we call the communication protocols. In this chapter, a communication protocol overview is presented, we have discussed the OSI reference model and described briefly the seven layers. Along with, the basics of communication protocol where we cited some communication problems and we give an example of a simple message exchange protocol. In the next sections, we summarize the steps of the protocol engineering.

5 The Sliding Window Protocol

Protocols are the basic elements of network operation providing message exchanges among the network nodes. Every protocol is used for communication purposes. Because of their fundamental importance, protocols must be designed with care so that they work correctly. Sliding Window Protocols are situated in the Data Link Layer of the ISO OSI layer model. They are used to provide reliable data communication between two computers in a network environment. In Tanenbaum's book ([146]) three Sliding Window Protocols are presented. In this chapter, we discuss a general and informal description of a Sliding Window Protocol. In particular, we concentrate on the Go-Back-N (GBN) protocol.

5.1 Introduction

Reliable transmission of data over unreliable channels is an old and well-studied problem in computer science. One of the most efficient protocols for reliable transmission is the Sliding Window protocol (SWP) [1]. The SWP has been widely used in many popular communication protocols such as Transmission Control Protocol (TCP), High-Level Data Link Control (HDLC) and Sequenced Packet Exchange (SPX). The protocol can ensure a correct data transfer over very poor quality communication channels where the frames may be duplicated, lost, or re-ordered [1, 147–149]. There are three sorts of Sliding Window protocols. The three vary among themselves in terms of efficiency, complexity, and buffer requirements. Our specification focuses on the Go-Back-N version of the protocol.

Even with the importance of the Sliding Window protocol, relatively few papers are done on formal methods, some versions of the protocol have been model-checked for small parameter values in [3, 4, 150]. The Sliding Window Protocol has been constructed by stepwise refinement 25 years ago [151], which was influenced by Dijkstra's work on the derivation of programs by using weakest preconditions [40]. For this approach, we developed our model specification in Event-B [9], based on the idea of refinement, the Event-B's systematic approach allows the user to construct models gradually and to facilitate a systematic reasoning method by means of proofs[9]. We liberally used refinements, both of the machines and of contexts. We give a great deal of attention to proofs. Consequently, we now have a specification of sliding window protocol where all proof-obligations have been discharged.

5.2 Informal Description[1]

SWP is a classical receive-send protocol, which is used in TCP/IP protocol group. It is an Automatic Repeat Request (ARQ) protocol used to provide for reliable data transfer over a lossy channel. The term sliding window refers to the management of the memory buffers that are used by the sender to store the outstanding (i.e., already sent but not yet acknowledged) frames, and by the receiver to store the frames that have already been received but are not yet ready to be passed to the user.

The basic principle of the sliding window protocol is the usage of a sending and receiving buffer. For the sender, it is possible to transmit more than one message while awaiting an acknowledgment for messages which have been transmitted before. In hardware description, an equivalent property is called pipelining.

Traditionally, three basic types of the sliding window protocol have been used and also taught in the communication protocols courses. These are the stop-and-wait, GBN, and selective-repeat protocols. We will only focus on the GBN protocol. The three mentioned protocols are treated as the members of the sliding window protocol family [146], deferring in their respective sender and receive window widths, as shown in Table 5.1.

Table 5.1 – Window Widths of Sliding Window Protocols

Protocol	Sender Window Width	Receive Window Width
Stop-and-Wait	1	1
Go-Back-N	W	1
Selective-Repeat	W	W

The GBN protocol has the sender window width greater than 1, which allows the sender to send several (up to the number equal to the sender window width) frames without having to wait for acknowledgments; however, if the sender window width is full the sender must wait for a new acknowledgment. In other words, when the receive window width is 1 the receiver has no room to arrange frames that might come out of order and can accept only the frames that are received in order; therefore, if a frame is lost the sender must resend that frame and all those that were already transmitted after the presumably lost frame.

5.2.1 Messages and Channels

For transmitting a frame, the sender places it into a frame message along with some additional information and sends it to the frame channel. After the receiver eventually receives the message frame from this channel, it sends an acknowledgment message for the corresponding frame back to the sender. This acknowledgment message is transmitted via the acknowledgment channel. After receiving an acknowledgment message, the sender knows that the corresponding frame has been received by the receiver. Therefore, the communication between the sender and the receiver is bidirectional; the sender sends frames to the receiver via the frame channel, and the receiver sends acknowledgments for these frames to the sender via the acknowledgment

[1]Section 5.2 is derived from the narrative and programs in Stenning (1976)[1]

channel.

A sliding window protocol is a protocol that uses the sliding window principle on both side sender and receiver:

5.2.2 Sender Sliding Window

The sender window is an imaginary box covering the sequence numbers of the data frames which can be in transit. In each window position, some of these sequence numbers define the frames that have been sent, others define those that can be sent. The maximum size of the window is 2^N - 1. Inside this sliding window, there are multiple copies of the transmission frames. When the correct Acknowledgement (ACK) arrived, sliding window will slide forward.

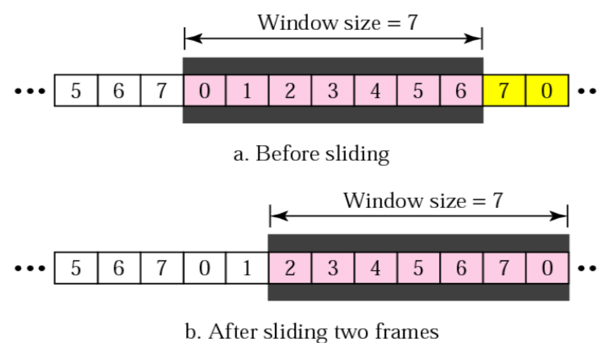


Figure 5.1 – Sender Sliding Window

5.2.3 Receiver Sliding Window

The receiver window makes sure that the correct data frames are received and that the correct acknowledgments are sent. Receiver sliding window in GBN is always 1. It's waiting for the correct frame comes in correct order, then sends back the ACK and slide forward. If the frame is lost or damaged, the receiver will wait for the resend. Even the rest of the frame is correct, the receiver will discard them automatically.

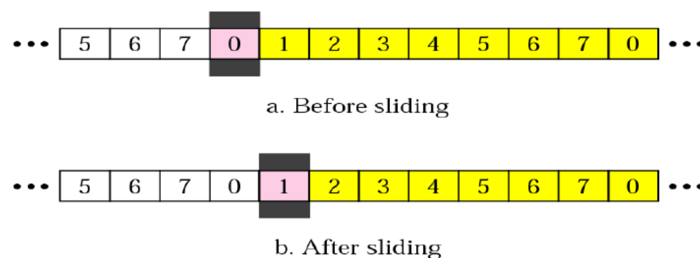


Figure 5.2 – Receiver Sliding Window

GBN is a family of sliding window protocol that is simpler to implement than Selective

Repeat Protocol (SRP), and possibly requires less buffering at the receiver. The principle of GBN protocl is as follows: if frame numbered n is lost, then all frames starting from n are retransmitted; frames out of order need not be accepted by the receiver.

5.2.4 Sequence Numbers

The ordering of the packets is defined using the sequence numbers which are attached to every packet. Packets from the sending window are numbered sequentially. However, because we need to include the sequence number of each frame in the header, we need to set a limit. If the header of the frame allows m bits for the sequence number, the sequence numbers range from 0 to 2^N - 1. For example, if n is 4, the only sequence numbers are 0 through 15 inclusive. However, we can repeat the sequence. So the sequence numbers are

0, 1,2,3,4,5,6, 7,8,9, 10, 11, 12, 13, 14, 15,0, 1,2,3,4,5,6,7,8,9,10, 11,...

In other words, the sequence numbers are modulo 2^N

The sender sends a sequence number with each message. A sequence number is unbounded and is incremented for each new message. The first message transmitted has sequence number 1. The receiver sends an acknowledgment when it receives a message. The acknowledgment carries a sequence number that refers to the last message successfully transferred to the receiving user. If an acknowledgment has to be sent before a successful reception (e.g. the first message was corrupted), it gives sequence number 0.

The Figures 5.3 and 5.4 show some examples of typical event sequences in the GBN protocol.

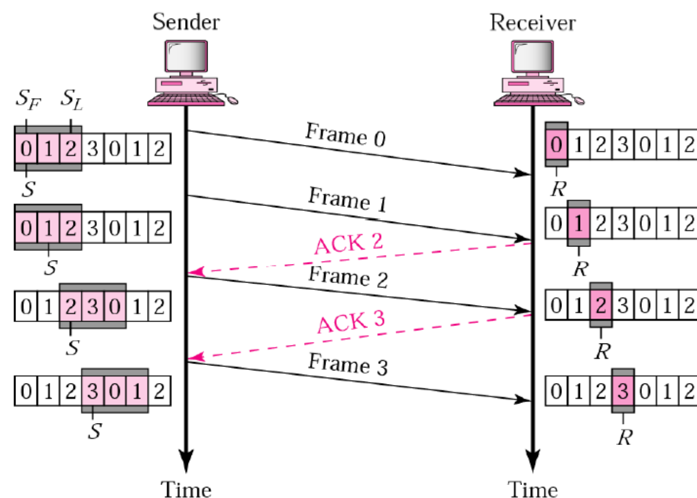


Figure 5.3 – Go-Back-N Normal Operation [152]

Figure 5.3 :Frame 0 & 1 send, ACK 1 & 2 back to sender. Frame 2 send, ACK 3 send back. Then frame 3 send to receiver.

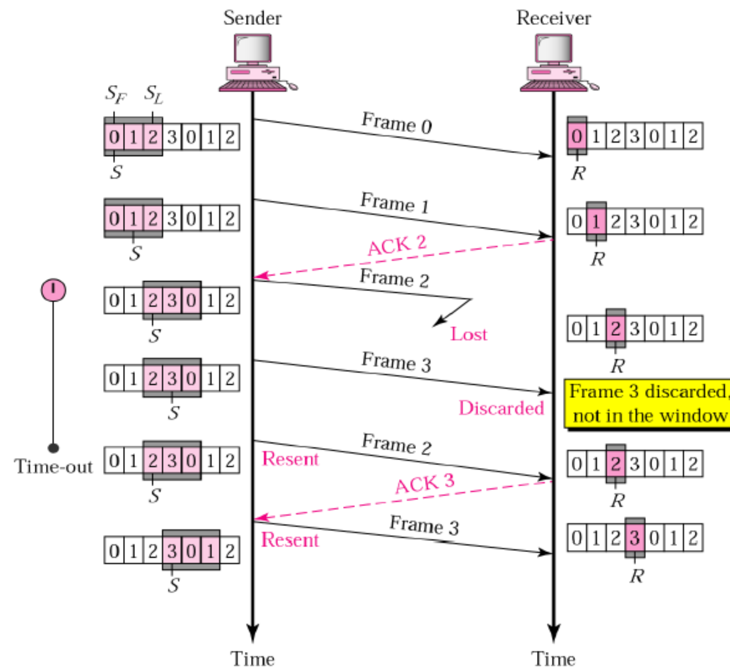


Figure 5.4 – Go-Back-N Lost Frame [152]

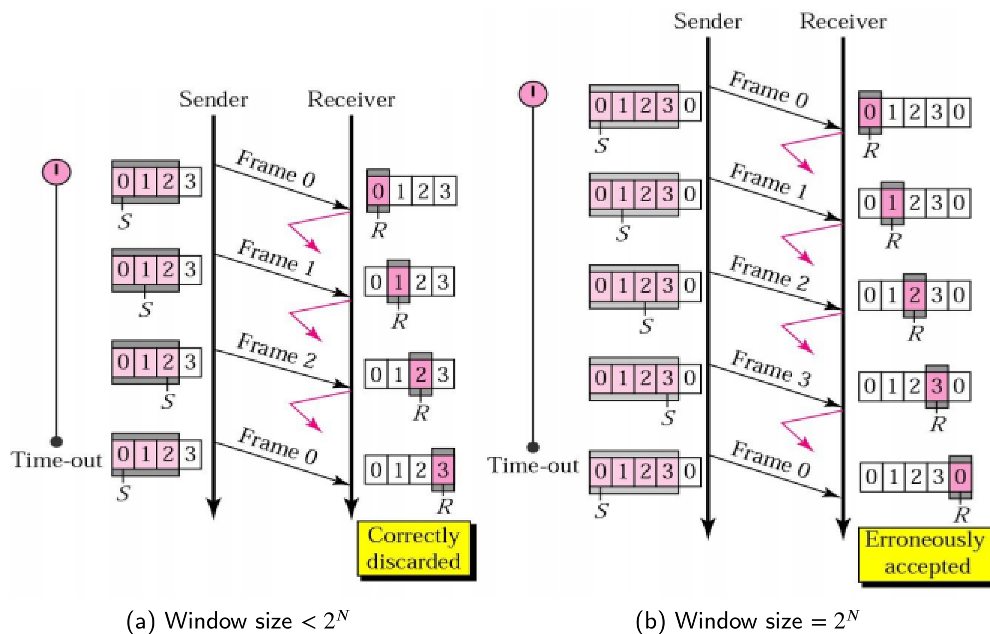
Figure 5.4 : Frame 0 & 1 send, ACK 1 & 2 back to sender. Frame 2 & 3 send, but frame 2 lost in the transmission. When frame 3 received out of order, this frame 3 will be discarded by receiver. After time out, frame 2 resent, then receiver send ACK 3 back and then frame 3 resent.

5.2.5 Timeout and Retransmission

One of the most widespread problems of computer networks is packet loss. The SWP uses positive acknowledgment with retransmission to detect and correct this problem. The transmitter sets a timer for each outgoing packet. If the timer expires before an acknowledgment is received, then the transmitter infers lost packets and retransmits the packet stored in the buffer. The receiver sends a positive acknowledgment if a frame has arrived safe and sound in order. If a frame is damaged or is received out of order, the receiver is inactive and will discard all subsequent frames until it receives the one it is expecting. The silence of the receiver causes the timer of the unacknowledged frame at the sender site to expire.This, in turn, causes the sender to go back and resend all frames, beginning with the one with the expired timer. The receiver does not have to acknowledge each frame received. It can send one cumulative acknowledgment for several frames.

As for the reason why the size of the sender window must be less than 2^N. We take an example from [152], we set N =2, which means the size of the window can be 2^N-1, which is 3. Figure 5.5 compares a window size of 3 against a window size of 4. If the size of the window is 3 (less than 2^2) (figure 5.5a) and all three acknowledgments are lost, the frame

timer expires and all three frames are resent. The receiver is now expecting frame 3, not frame 0, so the duplicate frame is correctly discarded. On the other hand(figure 5.5b), if the size of the window is 4 (equal to 2^2) and all acknowledgments are lost, the sender will send a duplicate of frame 0. However, this time the window of the receiver expects to receive frame 0, so it accepts frame 0, not as a duplicate, but as the first frame in the next cycle. This is an error.



(a) Window size $< 2^N$ (b) Window size $= 2^N$

Figure 5.5 – Window size for Go-Back-N [152]

5.3 Related Work

A Sliding Window Protocol (SWP) in general manages the communication on a point-to-point connection between two computers in a network at the Data Link Layer level in the OSI terminology. A SWP is a full-duplex protocol. This means that data can be transmitted simultaneously from station A to station B and vice versa. This is not the first attempt on Sliding Window Protocols in the context of formal specification. Sliding window protocols have attracted considerable interest from the formal verification community. In this section we present an overview of some work. In [153] Richier et al. specified a SWP in a process algebra based language Estelle/R, and verified safety properties for window size up to eight using the model checker Xesar. Madelaine and Vergamini [154] specified a SWP in Lotos, with the help of the simulation environment Lite, and proved some safety properties for window size six. Stenning [1] only gave an informal manual proof for his protocol. A semi-formal manual proof is also presented in [2]. A more formal, but not fully automated proof for the window size of one is given in [BG94], and for the arbitrary window size in [155]. The combination of abstraction techniques and model-checking in [156] allowed to verify the SW protocol for a relatively large window size of 16 (which is still a few orders less than a possible window size

in TCP). Smith and Klarlund [148] specified a SWP in the high-level language IOA, and used the theorem prover MONA to verify a safety property for unbounded sequence numbers with window size up to 256.

One way in which our work here differs from the other papers on formal verification of the sliding window protocol is that our model of the protocol is different from the others in way they all using a classic formal methods. Our approach is to apply the Event-B method which provide us with the refinement technique. Modeling with Event-B is very attractive thanks to the progressive reasoning that can lighten the complexity of the development. Event-B, based on a verification by Theorem Proving, provides a formal framework for modeling and proofing distributed systems. It also has a possibility of translating the formal specifications of the models into executable code.

5.4 Specifying SWP Using Event-B

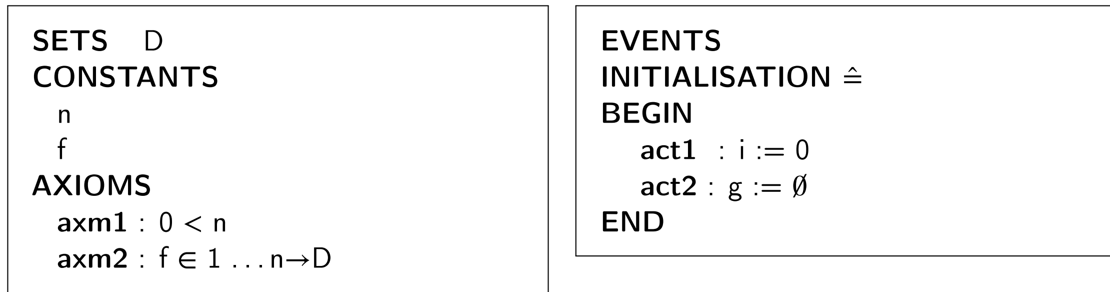
5.4.1 Refinement Strategy

In this short section, we present our strategy for constructing the sliding window protocol. This will be done by means of an initial model followed by six refinements.

- The initial model essentially presents the final result of the protocol where the file is totally in the receiver side which was sent in one shot. At this stage, the receiver and the sender are on different sites.
- In the first refinement, we introduce the file and the status of both the sender and the receiver.
- In the second refinement, we introduce the receiver window in our model
- In the third refinement, we introduce the data and the acknowledgment channels.
- In the fourth refinement, we introduce the unreliability of the channel.
- In the fifth refinement, we introduce timer.
- In the last refinement, we introduce the sequence number of the frames.

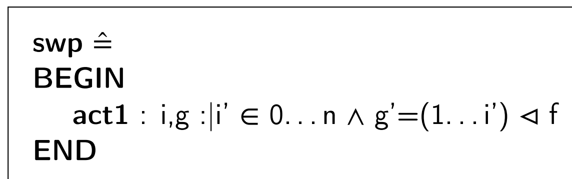
5.4.2 Initial Model

The initial model of SW protocol is presented as follows. At the end of this protocol execution, we want the file f to be copied at the receiver side on a file called g which is empty initially. The context is made of a set D. This set represents the data that are stored in the file.
We need two constants, constant n, which is a positive natural number(axm1), and constant f, which is a total function from the interval 1. . . n to the set D(axm2).



SETS D
CONSTANTS
n
f
AXIOMS
axm1 : $0 < n$
axm2 : $f \in 1 \ldots n \rightarrow D$

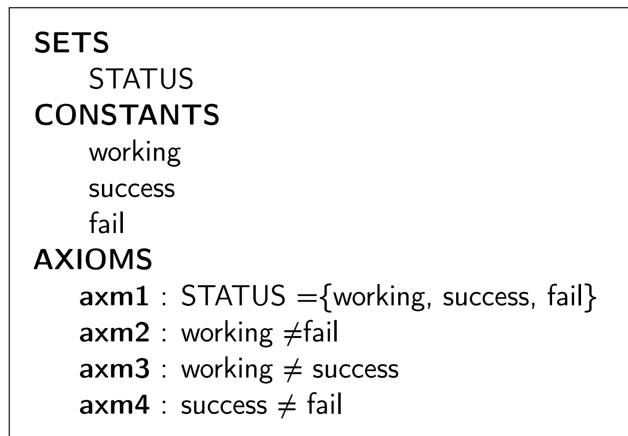
EVENTS
INITIALISATION $\triangleq$
BEGIN
act1 : $i := 0$
act2 : $g := \emptyset$
END



swp $\triangleq$
BEGIN
act1 : $i,g :| i' \in 0 \ldots n \land g' = (1 \ldots i') \lhd f$
END

5.4.3 First Refinement

We are going to refine our abstract model to a more concrete one, by adding more events and more variables to our model. For this, we introduce the receiver and the sender status. Also, we introduce the concept of status. For this, we define a carrier set named STATUS. It contains three constants (working, success, and fail) defined by axioms (axm1, axm2, axm3 and axm4).



SETS
STATUS
CONSTANTS
working
success
fail
AXIOMS
axm1 : STATUS ={working, success, fail}
axm2 : working $\neq$fail
axm3 : working $\neq$ success
axm4 : success $\neq$ fail

Then we can use four variables s_st and r_st defining the status of the two participants, sender and the receiver, and h which is the transmitted file of the length k.

INVARIANTS
- **inv1** : $k \in 0 \ldots n$
- **inv2** : $h = (1 \ldots k) \lhd f$
- **inv3** : $s_st=success \Rightarrow r_st=success$
- **inv4** : $r_st \in STATUS$
- **inv5** : $s_st \in STATUS$
- **inv6** : $h \subseteq f$
- **inv7** : $dom(h) \subseteq 1 \ldots n$

The event swp is fired when both of participants are not working.

swp $\hat{=}$
REFINES
- swp

WHEN
- **grd1** : $s_st \neq working$
- **grd2** : $r_st \neq working$

WITH
- $i' : i'=k \wedge g' : g'=h$

THEN
- skip

END

In this refinement, we have the sender and the receiver in both situations fail and success, they are expressed in these events sender_success, sender_fail, receiver_success, receiver_fail.

sender_success $\hat{=}$
WHEN
- **grd1** : $s_st = working$
- **grd2** : $r_st = success$

THEN
- **act1** : $s_st := success$

END

sender_fail $\hat{=}$
WHEN
- **grd1** : $s_st = working$

THEN
- **act1** : $s_st := fail$

END

receiver_success $\hat{=}$
WHEN
- **grd1** : $r_st = working$

THEN
- **act1** : $r_st := working$

END

receiver_fail $\hat{=}$
WHEN
- **grd1** : $r_st = working$

THEN
- **act1** : $r_st := fail$

END

5.4.4 Second Refinement

In the second refinement, we consider the receiver window. Only receiv_success event will be changer by adding more guards and actions. Our invariants are:

INVARIANTS
inv1 : $k \in 0 \ldots n$
inv2 : $k<n \Rightarrow k+1 \notin dom(h)$

receiver_success $\hat{=}$
WHEN
grd1 : r_st = working
grd2 : $k+1 = n$
grd3 : $k+1 \notin dom(h)$
THEN
act1 : r_st := working
act2 : $k := n$
act3 : $h := h \cup \{n \mapsto f(n)\}$
END

The new event receiver_current_data: correspond to the receiver receiving data from the sender which is not the last one.

receiver_current_data $\hat{=}$
WHEN
grd1 : r_st = working
grd2 : $k+1 < n$
THEN
act1 : $k := k+1$
act2 : $h := h \cup \{k+1 \mapsto f(k+1)\}$
END

5.4.5 Third Refinement

The sender and the receiver are connected by means of two channels as indicated in Figure 5.6: the data channel and the acknowledgment channel.

In this refinement, the window of sender will enter into the scene by cooperating with the receiver in order to transmit the file. Also, we introduce data channel and acknowledgment channel, which are situated between the sender and receiver. Moreover, we create a new carrier set called frame_status, which define the status of the frames, which are acknowledged (acked), not acknowledge (nacked) and not sent (nsent):

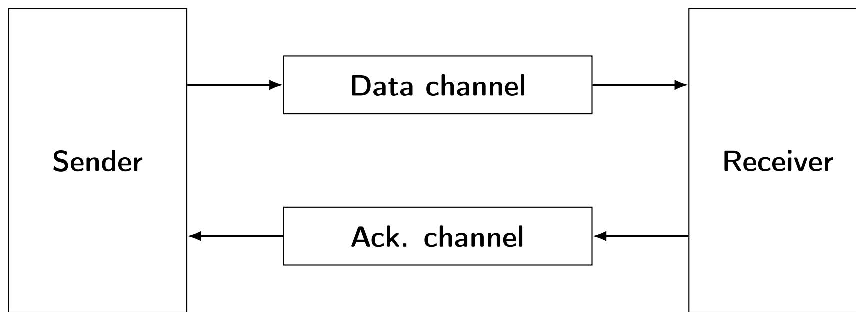


Figure 5.6 – The Channels

AXIOMS
- **axm1** : frame_status=acked, nacked, nsent
- **axm2** : acked $\neq$ nacked
- **axm3** : nacked $\neq$ nsent
- **axm4** : acked $\neq$ nsent

Moreover, we add more contexts about the size of both the receiver window RWS, which is equal to 1, and the sender window SWS which is a natural number and upper or equal to RWS.

CONSTANTS
RWS
AXIOMS
- **axm1** : RWS =1

CONSTANTS
SWS
AXIOMS
- **axm1** : SWS $\in \mathbb{N}$
- **axm2** : SWS $\geq$RWS

INVARIANTS
- **inv1** : buffer_s $\in 0 \ldots n$
- **inv2** : data_chan $\in 1 \ldots n \nrightarrow D$
- **inv3** : ack_chan $\subseteq 1 \ldots n$
- **inv4** : w $\in 1 \ldots n \rightarrow$ frame_status
- **inv5** : data_chan $\subseteq$ f

Events swp, sender_fail, receiver_fail are not modified in this step. The events sender_success and receiver_success are modified as bellow, the italics guards and actions are from the previous refinement.

sender_success $\widehat{=}$
REFINES
sender_success
WHEN
grd1 : *s_st = working*
grd2 : *r_st = success*
grd3 : buffer_s+1 ∈ ack_chan
THEN
act1 : *s_st := success*
act2 : w(n):=acked
act3 : ack_chan := ack_chan \ {buffer_s+1}
act4 : buffer_s := buffer_s+1
END

receiver_success $\widehat{=}$
REFINES
receiver_success
WHEN
grd1 : *r_st = working*
grd2 : *k+1 = n*
grd3 : *k+1 ∉ dom(h)*
grd4 : k+1 ∈ dom(data_chan)
THEN
act1 : *r_st := working*
act2 : *k:= n*
act3 : *h := h ∪ {n ↦ f(n) }*
act4 : ack_chan := ack_chan ∪ k+1
act5 : data_chan := k+1 ⩤ data_chan
END

We define now our new events of this refinement:

- ***receiver_sn_ack*** : correspond to the receiver sends an acknowledgment.
- ***sender_rcv_ack_advanc_win**:* the sender receives an acknowledgment then advances the window.
- ***rcv_ignor_data_out_win**:* which ignore all data out of the window.
- ***sender_rcv_ack** :* correspond to the sender receives an acknowledgment.

receiver_sn_ack $\widehat{=}$
ANY
frame
WHERE
grd1 : r_st = working
grd2 : frame $\leq$ k+ RWS
grd3 : frame $\in$ dom(h)
grd4 : w(frame)=nacked
THEN
act1 : ack_chan := ack_chan $\cup$ frame
act2 : data_chan := frame $\lhd$ data_chan
END

rcv_ignor_data_out_win $\widehat{=}$
ANY
frame
WHERE
grd1 : frame $\in$ dom(data_chan)
grd2 : r_st = working
grd3 : frame > k +RWS
THEN
act1 : data_chan := frame $\lhd$ data_chan
END

sender_rcv_ack_advanc_win $\widehat{=}$
WHEN
grd1 : s_st = working
grd2 : buffer_s+1<n
grd3 : buffer_s $\in$ ack_chan
THEN
act1 : ack_chan := ack_chan $\setminus$ buffer_s+1
act2 : w(buffer_s):= acked
act3 : buffer_s := buffer_s +1
END

sender_rcv_ack $\widehat{=}$
ANY
frame
WHERE
grd1 : frame $\in$ ack_chan
grd2 : s_st=working
grd3 : frame $\neq$ buffer_s+1
THEN
act1 : w(frame) := acked
act2 : ack_chan := ack_chan $\setminus$ frame
END

5.4.6 Fourth Refinement

In this refinement, we present the unreliability of the channel, now the Receiver picks up data from an unreliable data channel. The new contexts of this refinement are about the existence of the frame in the channels which can be either exist, never sent or lost, which are defined by the axioms.

AXIOMS
- **axm1** : $life = \{exist, never_sent, lost\}$
- **axm2** : $exist \neq never_sent$
- **axm3** : $exist \neq lost$
- **axm4** : $never_sent \neq lost$

For the maximum time that a frame can be transmitted we define it by constant max_retransmit, which is a natural number and strictly positive.

AXIOMS
- **axm1** : $max_retransmit \in \mathbb{N}$
- **axm2** : $max_retransmit > 0$

We enlarge our state by adding to it variables unreliable_data and unreliable_ack which are sets. These variables are initialized to empty. In additional we have frame_life which indicates the existence of the frame in the channel, and the variable retries which denote the number of retries of retransmission.

- **inv1** : $unreliable_data \in 1 \ldots n \nrightarrow D$
- **inv2** : $unreliable_ack \subseteq 1 \ldots n$
- **inv3** : $frame_life \in 1 \ldots n \rightarrow life$
- **inv4** : $retries \in 1 \ldots n \rightarrow \mathbb{N}$
- **inv5** : $unreliable_ack \subseteq ack_chan$

All these variables are initialized like this:

- **act9** : $unreliable_data := \emptyset$
- **act10** : $unreliable_ack := \emptyset$
- **act11** : $frame_life := 1 \ldots n \times never_sent$
- **act12** : $retries := 1 \ldots n \times 0$

The following events are modified as indicated below. The information is taken from the unreliable channels.

sender_success ≙
REFINES
    sender_success
WHEN
    grd1 : *s_st = working*
    grd2 : *r_st = success*
    grd3 : *buffer_s+1 ∈ ack_chan*
    grd4 : buffer_s+1 ∈ unreliable_ack
THEN
    act1 : *s_st := success*
    act2 : w(n):=acked
    act3 : ack_chan := ack_chan \ {buffer_s+1}
    act4 : *buffer_s := buffer_s+1*
    act5 : unreliable_ack := unreliable_ack \ buffer_s+1
END

sender_fail ≙
REFINES
    sender_fail
ANY
    frame
WHERE
    grd1 : *s_st = working*
    grd2 : frame ∈ 1 ...n
    grd3 : retries(frame) = max_retransmit
    grd4 : frame_life(frame) = lost
THEN
    act1 : *s_st := fail*
    act2 : retries(frame) := retries(frame)+1
END

receiver_success $\widehat{=}$
REFINES
receiver_success
WHEN
grd1 : *r_st = working*
grd2 : *k+1 = n*
grd3 : $k+1 \notin dom(h)$
grd4 : $k+1 \in dom(data_chan)$
grd5 : $k+1 \in$ dom(unreliable_data)
THEN
act1 : *r_st := working*
act2 : *k:= n*
act3 : $h := h \cup \{n \mapsto f(n)\}$
act4 : $ack_chan := ack_chan \cup k+1$
act5 : $data_chan := k+1 \lhd data_chan$
act6 : unreliable_ack := unreliable_ack $\cup$ k+1
act7 : unreliable_data := k+1 $\lhd$ unreliable_data
END

receiver_fail $\widehat{=}$
REFINES
receiver_fail
ANY
frame
WHERE
grd1 : *r_st = working*
grd2 : frame $\in$ 1... n
grd3 : retries(frame) > max_retransmit
THEN
act1 : r_st := fail
END

receiver_sn_ack $\hat{=}$
REFINES
receiver_sn_ack
ANY
frame
WHERE
grd1 : *r_st = working*
grd2 : *frame ≤ k+ RWS*
grd3 : *frame ∈ dom(h)*
grd4 : *w(frame)=nacked*
grd5 : frame ∈ dom(unreliable_data)
THEN
act1 : *ack_chan := ack_chan ∪ frame*
act2 : *data_chan := frame ⩤ data_chan*
act3 : unreliable_ack := unreliable_ack ∪ {frame}
act4 : unreliable_data := {frame} ⩤ unreliable_data
END

rcv_ignor_data_out_win $\hat{=}$
REFINES
rcv_ignor_data_out_win
ANY
frame
WHERE
grd1 : *frame ∈ dom(data_chan)*
grd2 : *r_st = working*
grd3 : *frame > k +RWS*
grd4 : frame ∈ dom(unreliable_data)
THEN
act1 : *data_chan := frame ⩤ data_chan*
act2 : unreliable_data := {frame} ⩤ unreliable_data
END

sender_rcv_ack_advanc_win ≙
REFINES
  sender_rcv_ack_advanc_win
WHEN
  grd1 : *s_st = working*
  grd2 : *buffer_s+1<n*
  grd3 : *buffer_s ∈ ack_chan*
  grd4 : buffer_s+1 ∈ unreliable_ack
THEN
  act1 : *ack_chan := ack_chan \ buffer_s+1*
  act2 : *w(buffer_s):= acked*
  act3 : *buffer_s := buffer_s +1*
  act4 : unreliable_ack := unreliable_ack \ {buffer_s+1}
END

sender_rcv_ack ≙
REFINES
  sender_rcv_ack
ANY
  frame
WHERE
  grd1 : *frame ∈ ack_chan*
  grd2 : *s_st=working*
  grd3 : *frame ≠ buffer_s+1*
  grd4 : frame ∈ unreliable_ack
THEN
  act1 : *w(frame) := acked*
  act2 : *ack_chan := ack_chan \ frame*
  act3 : unreliable_ack := unreliable_ack \ {frame}
END

The new event SND_resnd_data corresponds to the sender resend data where the frame is lost in the channel and it retries of the sending is less than maximum.

SND_resnd_data $\hat{=}$
ANY
  frame
WHERE
  grd1 : frame $\in$ 1...n
  grd2 : s_st = working
  grd3 : frame_life (frame) = lost
  grd4 : retries(frame) < max_retransmit
THEN
  act1 : unreliable_data := unreliable_data $\cup$ {frame $\mapsto$ f(frame)}
  act2 : frame_life(frame) := exist
  act3 : retries(frame) := retries(frame) +1
END

The next two events correspond to the daemons breaking the channels and causes lost frames.

DMN_data_channel $\hat{=}$
ANY
  frame
WHERE
  grd1 : frame $\in$ dom(unreliable_data)
THEN
  act1 : unreliable_data := {frame} $\lhd$ unreliable_data
  act2 : frame_life(frame) := lost
END

DMN_ack_channel $\hat{=}$
ANY
  frame
WHERE
  grd1 : frame $\in$ unreliable_ack
THEN
  act1 : unreliable_ack := unreliable_ack \ {frame}
  act2 : frame_life(frame):= lost
END

5.4.7 Fifth Refinement

Timers can help to avoid infinite loops and deadlocks. If none of the required activities has started within a predefined duration, the timer indicates the problem.

The time pattern presented in [157] provides a way to include timing properties. The pattern

adds an event tic tock simulating the progression of time. Timing properties are derived from the document. For this step, we need two constants: "propagation" is the propagation time needed for the message to transit from sender to receiver and "sleep_time" is the sleeping time used in the timer. As an adaptation we need two new variables: send_time_msg is the send time message and sleep for the time when sender will stop sleeping at the end of the timer. Variable at presents the active time. Hereby the invariants of this machine. In inv8, if time is beyond send_time_msg+propagation and if the time constraint send_time_msg+propagation has already been processed then we are sure of the succeed of the sender.

INVARIANTS

- **inv1** : $tm \in \mathbb{N}$
- **inv2** : $at \subseteq \mathbb{N}$
- **inv3** : $at \neq \emptyset \Rightarrow time \leq min(at)$
- **inv4** : $sleep \in \mathbb{N}$
- **inv5** : $tsend_time_msg \in \mathbb{N}$
- **inv6** : $propagation \in \mathbb{N}$
- **inv7** : $sleep_time \in \mathbb{N}$
- **inv8** : $s_st = success \land send_time_msg + propagation \notin at \land time \geq send_time_msg + propagation \Rightarrow r_st = success$
- **inv9** : $s_st = working \land at = sleep \Rightarrow time \geq send_time_msg + propagation$

The time event can be used as in the pattern:

Time $\hat{=}$
ANY
  tm
WHERE
  grd1 : $tm \in \mathbb{N}$
  grd2 : $tm > time$
  grd2 : $at \neq \emptyset \Rightarrow tm \leq min(at)$
THEN
  act1 : $time := tm$
END

We will now modify some events, other will not be changed:

action 6: $at := at \cup time + propagation \cup time + sleep_time$ will be added to send_success, sender_send_data and, SND_resnd_data event.

As we can see the two new active times time + propagation and time + sleep_time which are the future arrival time of messages and the awake time ending the timer.

guard 6 : time = send_time_msg+ propagation and **action 8**: at:= at \ {time } in receiv_success and receiv_current_data event.

The receiver events are triggered by send_time_msg+ propagation which is equal to the posted value time + propagation, this active time is deleted from the active time(at).

Therefore, we can conclude that the main purposes of timers in a communication protocol are the following:

- To ensure reliable transmission over an unreliable medium
- To delay the sending of a message to prevent overload
- To check the performance of the protocol

5.4.8 Sixth Refinement

For the last refinement, we add the sequence number of the frames. We need to add another context which we extend from context C5 (because we use the constant SWS). Our constants sequence_num represent the number of the sequence of each frame, this constant is a natural number and greater than sender window size plus 1. We do not forget that in the GO-Back-N protocol the sequence number are presented in modulo.

inv1: ∀frame1, frame2· frame1 ∈ dom(unreliable_data) ∧ frame2 ∈ dom(unreliable_data) ∧ (frame1 $\pmod{sequence_num} = frame2 \pmod{sequence_num}$) ⇒ frame1≠frame2

EXTENDS
  C5
CONSTANTS
  sequence_num
AXIOMS
  axm1 : sequence_num $\in \mathbb{N}$
  axm1 : sequence_num > SWS +1

Here we cite the changes that will be made in our events:

$$\text{send_success} \begin{cases} grd6 : seq_num \in unreliable_ack \\ grd7 : seq_num = buffer_s + 1 \pmod{sequence_num} \end{cases}$$

$$\text{receiv_sn_ack \& SND_resnd_data} \begin{cases} grd6 : seq_num \in dom(unreliable_data) \\ grd7 : seq_num = frame \pmod{sequence_num} \end{cases}$$

$$\text{receiv_success} \begin{cases} grd7 : sq_num \in dom(data_chan) \\ grd8 : sq_num = k + 1 \pmod{sequence_num} \end{cases}$$

rcv_ignor_data_out_win $\begin{cases} grd5 : seq_num \in dom(data_chan) \\ grd6 : seq_num = frame \pmod{sequence_num} \end{cases}$

sender_rcv_ack_advanc_win $\begin{cases} grd5 : seq_num \in unreliable_ack \\ grd6 : seq_num = buffer_s + 1 \pmod{sequence_num} \end{cases}$

sender_rcv_ack & DMN_ack_channel $\begin{cases} grd5 : seq_num \in ack_chan \\ grd6 : seq_num = frame \pmod{sequence_num} \end{cases}$

sender_send_data $\begin{cases} grd5 : seq_num \in dom(data_chan) \\ grd6 : seq_num = k \pmod{sequence_num} \end{cases}$

DMN_data_channel $\begin{cases} grd5 : seq_num \in dom(data_chan) \\ grd6 : seq_num = frame \pmod{sequence_num} \end{cases}$

Proofs Statistics

Table 5.2 expressing the proof statistics of the development in the RODIN tool. These statistics measure the size of the model, the proof obligations are generated and discharged by the RODIN platform, and those are interactively proved. For this specification, most of the proof obligations are automatically discharged by RODIN.
The complete development of the Sliding Window protocol results in 151 POs, within which 133 are proved automatically by the RODIN tool, the remaining 18 POs are proved interactively using the RODIN tool.

Table 5.2 – Proof Statistics for the Sliding Window Protocol Development

Model	Total POs	Automatic Proof	Interactive proof
Abstract model	5	4	1
First refinement	10	10	0
Second refinement	13	8	5
Third refinement	28	25	3
Fourth refinement	40	39	1
Fifth refinement	38	33	5
Sixth refinement	17	14	3
Total(%)	151(100%)	133(88%)	18(12%)

5.5 Summary

In this chapter, we have presented formal specification of the Sliding Window protocol using Event-B. In this approach, the modeling process starts with an abstraction of the protocol which specifies the goals of the protocol. In our case study, a sliding window protocol, sending and receiving windows of different size are the main protocol goals. The abstraction level of our Event-B model shows these goals in a very general way, and then during refinement levels, features of the protocol are modeled and the goals are achieved in a detailed way.

6 The NetBill Electronic Commerce Protocol

Despite that NetBill protocol being a well-known protocol, it is surprisingly difficult to verify the correctness of a design based on this protocol by hand. So it makes very desirable the application of formal methods and techniques to the modeling and design of electronic commerce protocols to gain high assurance about their correctness. Hence, formal methods are needed in order to ensure their correctness and structure their development from specification to implementation.

6.1 Introduction

Among those areas where computers are yielding a higher impact during the last years, we may find the Electronic Commerce (E-Commerce) systems, which facilitate the conduct of business over the Internet with the assistance of Web techniques, have been adopted by more and more companies. E-commerce has become an essential feature in the modern global economic environment. It represents the use of a computer network, mainly the Internet, to purchase and sell products, services, information, and communication. In fact, commerce actions under electronic, computer, and telecommunication support have various advantages with respect to the classic commerce. Electronic commerce protocols are security protocol that allows customers and merchants to conduct their business electronically through the Internet. Among them are Secure Electronic Transactions (SET) [158] and NetBill [5] ,[33].

NetBill is an electronic commerce protocol, which allows customers to purchase information goods from merchants over the Internet.

E-commerce applications are usually very complex. Thus, it is difficult to verify reliability and correctness. It is hard to ensure the correctness of a design based on this protocol by hand. Therefore, it makes very appropriate the application of formal methods and techniques to the modeling and design of such protocol to gain high assurance about their correctness. However, guarantying the correctness of complex e-commerce protocols is a hard task and informal methods are inadequate. Formal methods [159] are needed for the construction of unambiguous and precise models that can be analyzed to identify errors and verify correctness before implementation. The use of formal methods will lead to more reliable and trustworthy e-commerce protocols [160].

Different formal approaches have been applied to model and analyze several protocols, such as modeling SET in model checking [161] or Mondex in Event-B [162]. This paper presents our approach to model communication properties of NetBill protocol and its formal specification in Event-B [9].

6.2 NetBill Protocol

The NetBill protocol is an electronic commerce protocol optimized for the selling and delivery of low-priced information goods, such as software or journal articles, across the Internet. It was developed by Carnegie Mellon University in conjunction with Visa and Mellon Bank to deal with micropayments of the online order.

The NetBill transaction model includes three participants: the consumer (C), the merchant (M) and the NetBill server (S).

A transaction involves three phases: price negotiation, product Request, and payment. In a NetBill transaction, the customer and merchant interact with each other in the first two phases; the NetBill server is not involved until the payment phase. The NetBill protocol consists of eight messages (Figure 6.1), Here is an outline of the NetBill protocol (We use the notation X ⇒ Y to indicate that X sends the specified message to Y) :

Price Request Phase

The first phase of the NetBill protocol is the price request phase. It makes up the first two message of the protocol and consists of a request and a response. It allows a customer to demand the price and a detailed goods description which he found on the web page of a merchant.

1- C ⇒ M : Price Request
2- M ⇒ C : Price Response

Product Request Phase

In case the customer is interested in buying the goods, he is free to demand them from the merchant, who in turn provides them encrypted with a key K.

3- C ⇒ M : Product Request
4- M ⇒ C : Product, encrypted with a key K

After the Product Request Phase, the customer has got the products in encrypted form, but can not use them.

Payment Phase

After the encrypted goods are delivered, the customer submits payment to the merchant

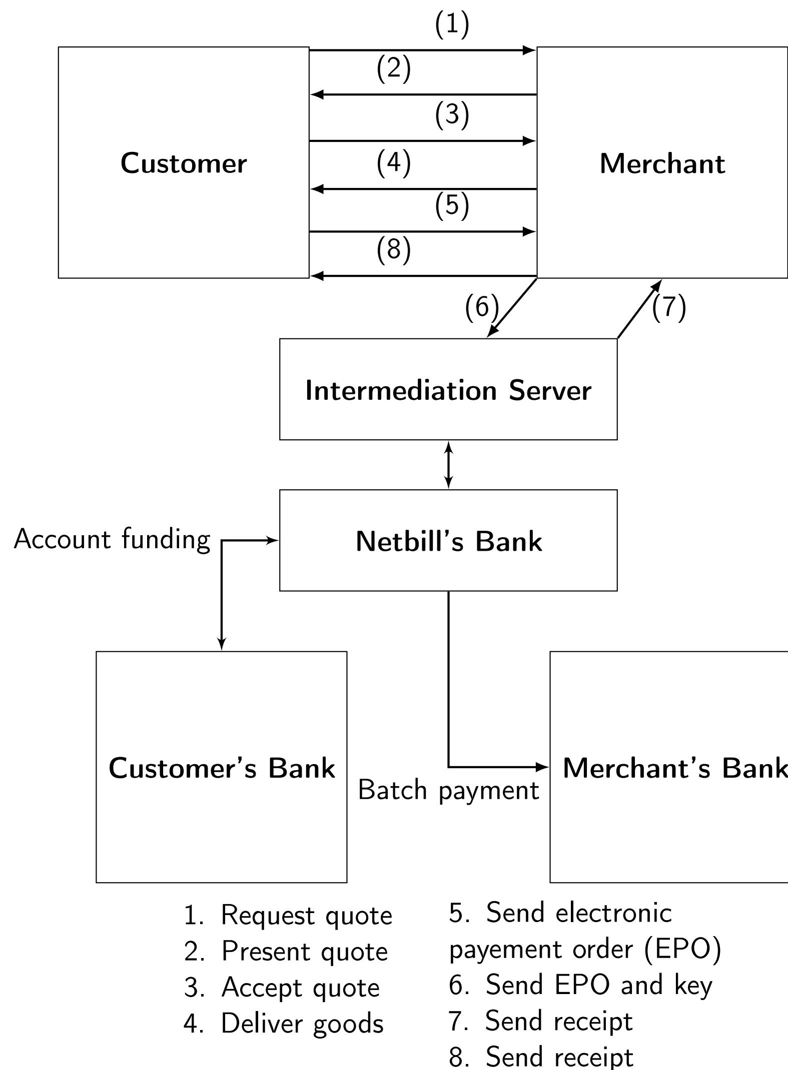


Figure 6.1 – The NetBill protocol

in the form of a signed Electronic Payment Order. If the merchant accepts this electronic check, he forwards it together with the secret product encryption key K to the NetBill server. If everything seems all right, the server carries out the transaction and saves the product decryption key into his database. Finally he responds to the merchant by sending a signed receipt. The merchant forwards the receipt, which contains the key K, to the customer.

5- C ⇒ M : Signed Electronic Payment Order (EPO)
6- M ⇒ S : Endorsed EPO (with K)
7- S ⇒ M : Signed result (with k)
8- M ⇒ C : Signed result (with K)

It is important that the product decryption (or encryption) key K makes part of the EPO

forward to the NetBill server, who stores it in its database. This is necessary in order to prevent the merchant to defraud the customer: if the merchant does not send the key K to the customer, he can recover it directly from the NetBill server.

6.3 Specifying NetBill Protocol Using Event-B

6.3.1 Refinement Strategy

In this short section, we present our strategy for constructing the NetBill protocol. This will be done by means of an initial model followed by two refinements.

1. The initial model is a high level of abstraction, showing that the customer orders a product and that the transaction terminated.
2. The first refinement introduces delivery goods operation along with accepting the encrypted goods by the customer.
3. The second refinement contains the decryption goods along with the operations made in the customer and merchant accounts.

For our model we consider the transactions from the point of view of the customer and the merchant[163], which is shown in the figure below.

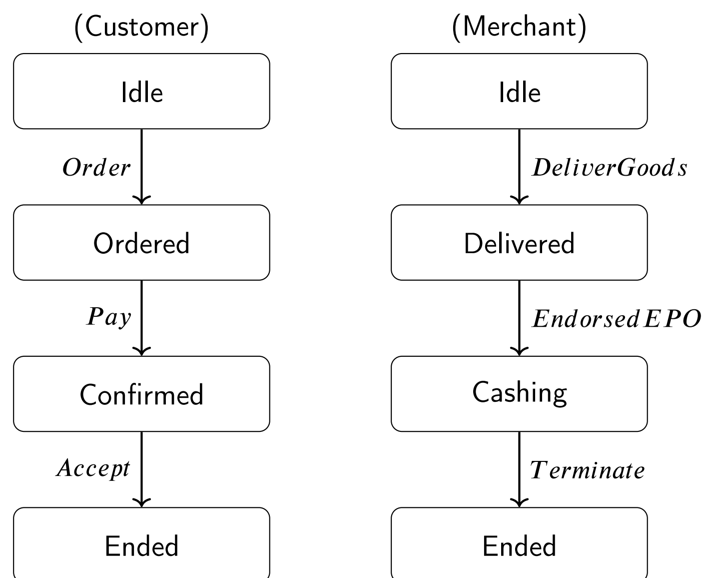


Figure 6.2 – Customer and Merchant View of a Transaction

The Event-B model of the NetBill protocol consist of three machines which model a state and the events representing behavior that could occur, the conditions that must apply if an event occurs and the effect of the event has on the state. As such, a machine gives a representation of possible behaviors of the protocol. In addition to the previous machines, we have three

contexts which are used in Event-B to define constant values such as abstract sets, relations, functions, properties of those constants, called axioms and theorems expressing properties of the constants that can be deduced from the axioms. The abstract sets are sometimes named carrier sets.

6.3.2 Initial Model

In this initial model, we just formalize what the customer can eventually do: order a product. First, we define a carrier set Goods: it describes the goods information(PRD, PRICE and KEY). Then we define another carrier set named Transaction. It is made of six distinct elements: idle, ordered, confirmed, delivered, cashing and ended which present the transaction status.

SETS
Transaction
CONSTANTS
idle
ordered
confirmed
delivered
cashing
ended

SETS
Goods
CONSTANTS
PRD
PRICE
KEY

After that, we define four variables: goods, trans, status and agreed denoting respectively the set of ordered product, a set of the transaction, and a variable Boolean to indicate if there is an agreement or not.

The invariants specify the properties that the variables (the state) must satisfy before and after every event, excepting the initialization where the invariants must be satisfied after the initialization.

INVARIANTS
inv1 : $goods \subseteq$ **Goods**
inv2 : $goods \neq \emptyset \Rightarrow (\exists\ g.goods = \{g\})$
inv3 : $trans \subseteq Transaction$
inv4 : $status \in trans \rightarrow Transaction$
inv5 : $agreed \in BOOL$
inv6 : $\forall\ t.t \in trans \wedge status(t) = ended \Rightarrow agreed = TRUE$
inv7 : $agreed = TRUE \Rightarrow (\forall t.t \in trans \wedge status(t) = ended)$

Finally, we define the dynamics of the system by means of three events: The INIT event makes variable goods empty, agreed false and the status of the transaction is idle, the Order event and the Terminate event which correspond to terminate a transaction :

INIT $\hat{=}$
BEGIN
  act1 : goods := $\emptyset$
  act2 : trans := $\emptyset$
  act3 : status := $\emptyset \times \{idle\}$
  act4 : agreed := FALSE
END

Initialization (INIT) is a distinguished event that occurs only once, before any other event. This event initializes the machine's variables to a set of values that establishes the invariant. Note that the variables do not have any value before initialization.

Order $\hat{=}$
ANY
  g
WHERE
  grd1 : $g \in goods$
  grd2 : $goods = \emptyset$
THEN
  act1 : goods := $\{g\}$
END

Terminate $\hat{=}$
ANY
  t
WHERE
  grd1 : $t \in trans$
  grd2 : agreed = TRUE
THEN
  act1 : status(t) := ended
END

At this point, our abstract machine simply describes the transaction of ordering a product. We will now model behavior that describes the delivery of goods operation and accepting the encryption goods by the customer. This process consists to refine our abstract machine.

6.3.3 First Refinement

We are going to refine our abstract model to a more concrete one, by adding more events and more variables to our model.

For this, we define carrier set AGENT, which we will use as the source of unique identifiers for agents. The set is not given a specific size, but it is declared to be finite, meaning that it does have a size that is a natural number. Sets are potentially infinite unless declared otherwise.

SETS
  AGENT
AXIOMS
  axm1: finite(AGENT)
END

Then we introduce these new variables: t_Id indicates a transaction ID, value corresponds to the price of the goods, encry indicate the encryption of the goods and variable EPO present the signed EPO of customer and merchant.

INVARIANTS
  inv1 : $\forall$ t.t$\in$trans $\Rightarrow$ status(t)=idle
  inv2 : status $\in$ Transaction $\twoheadrightarrow$ Transaction
  inv3 : t_Id $\in$ 0...n
  inv4 : value $\in$ Goods $\rightarrow \mathbb{N}$ each good has a value
  inv5 : $\forall$ t.t $\in$trans $\wedge$ status(t)= idle $\Rightarrow$ status(t)= delivered
  inv6 : encry $\in$ Goods $\rightarrow \mathbb{N}$ each good has an encryption
  inv7 : $\forall$ t.t$\in$trans $\wedge$ status(t)= ordered $\Rightarrow$ status(t)= confirmed
  inv8 : EPO $\in$ AGENTS $\rightarrow \mathbb{N}$ each agent has an EPO
  inv9 : customer $\subseteq$ AGENTS   each customer has a unique Id
  inv10 : Merchant $\subseteq$ AGENTS   each merchant has a unique Id
  inv11 : price $\in \mathbb{N}1$ any price other than free

This refinement aimed at adding new functionality, rather than refining the current functionality. For that reason, all events will be displayed in extended mode, a mode supported by Rodin. In extended mode, only the new parameters, guards and actions are displayed, that is, only the parts of an event that extend the event being refined. The initializations of our new variables. The initializations of our new variables.

INIT extended $\hat{=}$
BEGIN
  act8 : t_Id := 0
  act9 : value := Goods $\times$ {0}
  act10 : encry := Goods $\times$ {0}
  act11 : EPO := AGENTS $\times$ {0}
  act12 : price :$\in$ **N**1
END

For this refinement, event Order will be refine by adding new guards and actions, the customer orders goods and receives the price of the goods from the merchant(act4), then the status of the transaction will be in ordered(act2) then the id of the transaction will increment by 1(act5).

Order $\widehat{=}$
REFINES
  Order
ANY
  g, m, t
WHERE
  grd1 : $g \in goods$
  grd2 : $goods = \emptyset$
  grd3 : $m \notin Merchant$
  grd4 : $t \in trans$
THEN
  act1 : goods := {g}
  act2 : status(t):=ordered
  act3 : Merchant := Merchant $\cup$ {m}
  act4 : value(g) := price
  act5 : t_Id := t_Id +1
END

Now, we define our new events of this refinement:

- FixPrice: event to set a price to a good.
- Start: corresponding to the start of a transaction where initially all transactions are idle.
- Goods_delivery: correspond to sending the encrypted goods to the customer and change the status of the transaction from idle to deliver.
- Pay: correspond to accepting the encrypted goods, generating an EPO and changing the status of the transaction from ordering to confirm.

Start $\widehat{=}$
ANY
  t
WHERE
  grd1 : $t \in Transaction \setminus trans$
  grd2 : $t \in dom(status)$
THEN
  act1 : trans := trans $\cup$ {t}
  act2 : status(t) :=idle
END

FixPrice $\widehat{=}$
ANY
  amount
WHERE
  grd1 : $amount \in \mathbb{N}1$
THEN
  act1 : price := amount
END

Goods_delivery $\widehat{=}$
ANY
  t, g, k
WHERE
  grd1 : $t \in trans$
  grd2 : $g \in goods$
  grd3 : $status(t) = idle$
  grd4 : $k \in KEY$
THEN
  act1 : $goods := \{g\}$
  act2 : $status(t) := delivered$
  act3 : $value(g) := price$
  act4 : $encry(g) := k$
END

Pay $\widehat{=}$
ANY
  t, g, k, c, ID
WHERE
  grd1 : $t \in trans$
  grd2 : $g \in goods$
  grd3 : $status(t) = ordered$
  grd4 : $k \in KEY$
  grd5 : $encry(g)=k$
  grd6 : $value(g) \leq p$
  grd7 : $c \in customer$
  grd8 : $ID \in KEY$
THEN
  act1 : $status(t) := confirmed$
  act2 : $value(g) := price$
  act3 : $EPO(c) := ID$
END

6.3.4 Second Refinement

In this refinement, we enlarge our state by adding to it variables decry and account which define respectively the decryption of goods and the account of the customer and the merchant.

INVARIANTS
  inv1 : $\forall\, t.t \in trans \land status(t)= delivered \Rightarrow status(t)= cashing$
  inv2 : $decry \in Goods \rightarrow \mathbb{N}$
  inv3 : $\forall\, t.t \in trans \land status(t)= confirmed \Rightarrow status(t)= ended$
  inv4 : $account \in AGENTS \rightarrow \mathbb{N}$

We describe our new events of this refinement:

- Endored_EPO: Correspond to checking the EPO sent by the customer, then, the status of the transaction will change from delivered to cashing.
- Accept: Getting the decryption key, then, the status of the transaction will change from confirmed to ended.
- Terminate1: NetBill server debits the account of the customer.

Terminate event will be refined in a form that the NetBill server credits the account of the merchant and sends a receipt to the merchant. Finally, the merchant forwards this receipt containing the decrypting key to the customer(Event accept).

```
Endored_EPO ≙
ANY
    t, m, c, ID
WHERE
    grd1 : t ∈ trans
    grd2 : m ∈ Merchant
    grd3 : c ∈ customer
    grd4 : status(t)= delivered
    grd5 : ID ∈ KEY
    grd6 : EPO(c) =ID
    grd7 : EPO(m) =ID
THEN
    act1 : status(t):=cashing
    act2 : EPO(c) := EPO(m)
END
```

```
Accept ≙
ANY
    g, k, t
WHERE
    grd1 : g ∈ goods
    grd2 : k ∈ KEY
    grd3 : t ∈ trans
    grd4 : status(t) =confirmed
THEN
    act1 : decry(g):=k
    act2 : status(t):=ended
END
```

```
Terminate1 ≙
ANY
    g, m, c
WHERE
    grd1 : g ∈ goods
    grd2 : c ∈ customer
    grd3 : account(c) ≥ value(g)
THEN
    act1 : account(c):= account(c) − value(g)
END
```

Terminate $\hat{=}$
REFINES
Terminate
ANY t, g, c, m
WHERE
grd1 : $t \in trans$
grd2 : $agreed = TRUE$
grd3 : $g \in goods$
grd4 : $c \in customer$
grd5 : $m \in Merchant$
grd6 : $account(c) = account(c) - value(g)$
grd7 : $price = value(g)$
grd8 : $account(c) \neq account(m)$
THEN
act1 : $status(t) := ended$
act2 : $account(m) := account(m) + price$
END

Proofs Statistics: The proof statistics for the development of the Netbill protocol is in Table 6.1. For this specification, most of the proof obligations are automatically discharged by Rodin.
The complete development of the Netbill protocol results in 98 POs, within 85 are proved automatically by the Rodin tool, the remaining 13 POs are proved interactively using the Rodin tool.

Table 6.1 – Proof Statistics for the NetBill Protocol Development

Model	Total POs	Automatic Proof	Interactive proof
Abstract model	10	8	2
First refinement	51	42	9
Second refinement	37	35	2
Total(%)	98(100%)	85(86%)	13(14%)

6.4 Summary

With the growth of the Internet community and the endless possibilities the Internet offers to the person, it didn't take long before someone realized that the web is a really good place for the commercial business. So, very quickly electronic commerce was born, offering almost all kinds of goods to be purchased and delivered, simply over the Internet.

NetBill is an e-commerce protocol that is used for shopping over information products using the Internet. Since it is used extensively by e-commerce web services, its correctness is of

critical importance. The protocol is complex to perform hand-proofs. So, we use a tool based refinement methodology for verifying the NetBill algorithm. In this chapter, we have presented an incremental formal modeling of the NetBill protocol using Event-B. We choose Event-B because it promotes a layered style of formal modeling, where a model is developed as a chain of abstract models; step by step concrete details are progressively introduced via provably correct refinement steps, which allows us to achieve a very high degree of automatic proof. The Rodin platform automatically generates the required proof obligations and Rodin provers discharge the proof obligations automatically for ensuring correctness of the system.

7 The Tatebayashi, Matsuzaki and Newman (TMN) Protocol

Formal methods have shown their interest when developing critical systems, where safety or security is important. This is particularly true in the field of security protocols. Such protocols aim at securing communications over a public network. Small flaws in the development of such systems may cause important economic damages.

7.1 Introduction

Cryptographic protocols play a crucial role in achieving security in today's communication systems. They are widely used for various purposes between the agents: like Exchange secret information, achieve a transaction (Electronic Commerce), authenticate one, some or all the agents, vote and protect copyright on digital content.

Often cryptographic algorithms and protocols are necessary to keep a system secure, particularly when communicating through an untrusted network such as the Internet. Generally, it involves two communicating agents who exchange a few messages, with the help of a trusted server. The exchanged messages are composed of components such as keys, random numbers, timestamps, and signatures [164]. At the end of the protocol, the agents involved may deduce certain properties such as the secrecy and authenticity of an exchanged message[165].

One of the main advantages of formal methods is to provide a clear, well-defined mathematical model that allows to reason about the capacity of an attacker and to precisely state the security guarantees achieved by a protocol in the given model. A large variety of models have been proposed so far, proposing different trade-offs between expressibility and the possibility to automate proofs.

In this chapter, we present an approach to model TMN protocol[6]. While there exist a great variety of formal models that have been applied to model and analyze TMN protocol, such as Coloured Petri Nets [7] but none are made by Event-B Method.

7.2 TMN Protocol

The Tatebayashi, Matsuzaki and Newman (TMN) Protocol [6] is a key distribution protocol for digital mobile communication systems, such as cellular networks. The TMN protocol concerns three players :an initiator A, a responder B and and a server S, to facilitate the distribution of a session key, K_{ab}. The TMN protocol for establishing a session key involves the exchange of four messages ; it is illustrated below:

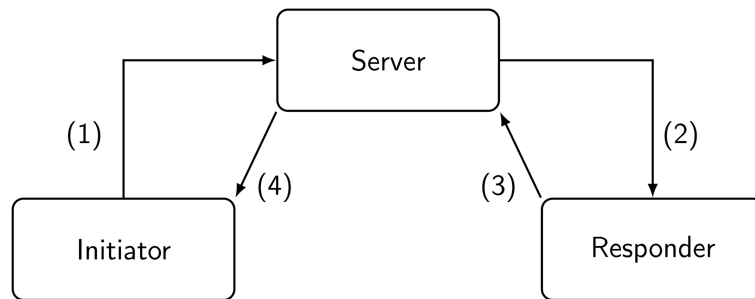


Figure 7.1 – Message Exchange in TMN Protocol

We use the following notation: (i) A → B : X to indicate that in the ith step of the protocol agent A sends message X to agent B. We write A → B : X, Y to denote "A sends B the message X along with the message Y ".

The protocol can be defined as follows :

(i) A → S: B, $\{K_{as}\}PK_s$

(ii) S → B: A

(iii) B → S: A, $\{K_{ab}\}PK_s$

(iv) S→ A: B, $\{K_{ab}\}K_{as}$

The TMN protocol work as detailed below. The server possesses a public-private key pair. The public key is known to everyone in the system, while the private key belongs to the server alone.

1. When user A wishes to communicate with user B, it encrypts a random number with the server's public key, and sends the encrypted random number, along with A's and B's names.
2. When the server receives the request, it decrypts the random number and stores it as a key-encryption key for that conversation; it also notifies B that A wishes to speak to it.
3. User B, on receiving the notification from the server, generates a random number to be used as a session key, encrypts it with the server's public key, and sends it to the server.

4. The server decrypts the response, encrypts the key with A's random number using a private-key algorithm, and sends the result to A. In the specification of the protocol, the algorithm used for this step is commutative in the sense that, if A[B] represents the encryption of B under the key A, we have A[B]=B[A].

5. To obtain the session key, user A decrypts the message from the server using the original random number it had generated and assumes that the result is the session key.

In message (iv), K_{ab} is encrypted using a symmetric key algorithm with the key K_{as}. Hence, the encryption operators used in (iv) on one hand and in (i) and (iii) on the other hand differ (though the notation is the same).

7.3 Refinement Strategy

In this short section, we present our strategy for constructing the TMN protocol without taking into account the intruder. This will be done by means of an initial model followed by one refinement.

1. The initial model is high-level abstraction showing that the agents do not send messages to each other.

2. In the first refinement, we introduce the basic message exchange process between the agents.

7.3.1 Initial Model

The initial model of TMN protocol is presented as follows: First, we define two carrier sets Random_num and Agent, which describe, respectively, random number that is generated by the initiator or the responder and set of the agents that contain Initiator, Server, and the Responder as constants.

```
SETS
    Random_num
    Agent
CONSTANTS
    Initiator, Server, Responder
AXIOMS
    axm1 : partition(Agent, {Initiator}, {Server}, {Responder})
END
```

Then the variables num_A and num_B corresponding to the random numbers generated by the initiator and the responder, which are added at the final to the variable num_final.
We define three variables i1, i2, i3 and i4 that correspond to what Initiator record. The

variable i1 is a total function from the set num_A to the set Initiator.
The variable i2 is a total function from the set num_A to the set Responder. It means that the initiator i1(num_A) has started a protocol execution in order to speak to recipient R. The variable i4 is a Boolean function which indicates that the presumed destination of a transaction is active and involved in the current transaction.

INVARIANTS
- **inv1** : $num_A \subseteq Random_num$
- **inv2** : $num_final \subseteq (num_A \cup num_B)$
- **inv3** : $num_B \subseteq Random_num$
- **inv4** : $i1 \in num_A \rightarrow Initiator$
- **inv5** : $i2 \in num_A \rightarrow Responder$
- **inv6** : $i3 \in num_A \rightarrow num_B$
- **inv7** : $i4 \in num_A \rightarrow BOOL$
- **inv8** : $\forall\ num.num \in num_final \Rightarrow i4(num) = TRUE$

For our abstract model, we have three events which are: Event Init which corresponds to step 1 of the protocol where the Initiator A records its intention to speak to the responder B by choosing a random number. Event level, which indicates the Responder records that initiator A wants to speak to B and Event Final.
At the beginning of a transaction, the variable i4 is set to FALSE by the event INIT, at the end of the transaction (Final event), it is necessary that the agent who initiated the transaction to be able to say that the presumed destination is active.

Init $\triangleq$
ANY num_I, a, b
WHERE
- **grd1** : $num_I \notin num_A$
- **grd2** : $a \in Initiator$
- **grd3** : $b \in Responder$

THEN
- **act1** : $num_A := num_A \cup \{num_I\}$
- **act2** : $i1(num_I) := a$
- **act3** : $i2(num_I) := b$
- **act4** : $i4(num_I) := FALSE$

END

final $\widehat{=}$
ANY num_I
  WHERE
    grd1 : num_I ∈ num_A \ num_final
    grd2 : i4(num_I) =TRUE
  THEN
    act1 : num_final := num_final ∪ { num_I}
  END

level $\widehat{=}$
ANY num_I, a, b
  WHERE
    grd1 : num_I ∈ num_A
    grd2 : a = i1(num_I)
    grd3 : b = i2(num_I)
  THEN
    act1 : i4(num_I) := TRUE
  END

7.3.2 First Refinement

We are going to refine our abstract model to a more concrete one, by adding more events and more variables to our model. For this, we introduce set of MSG and set KEYS.

The variable msg is the set of messages. This set is partitioned into four sets msg1, msg2, msg3, and msg4. Then variables m(i)_id and m(i)_ki to model ith message.

INVARIANTS
  inv1 : msg ⊆ MSG
  inv2 : partition(msg ,msg1, msg2,msg3,msg4)
  inv3 : m1_id ∈ msg1 → Initiator
  inv4 : m1_ki ∈ msg1 → num_A
  inv5 : m2_id ∈ msg2 → Server
  inv6 : m3_id ∈ msg3 → Responder
  inv7 : m3_ki ∈ msg3 → num_B
  inv8 : m4_id ∈ msg4 → Server
  inv9 : session_key ⊆ KEYS
  inv10 : m4_ki ∈ msg4 → session_key

Now, we define our events of this refinement: In each event generates a new message belonging to msg(i). For the events Step 1, Step 2 and Final, they refine respectively events Init, level

and final. The emphasis guards and actions represent guards and actions of the previous machine.

Step1 $\widehat{=}$
REFINES Init
ANY m
  WHERE
    grd1 : *num_I $\notin$ num_A*
    grd2 : *a $\in$ Initiator*
    grd3 : *b $\in$ Responder*
    grd4 : m $\notin$ msg
  THEN
    act1 : *num_A := num_A $\cup$ {num_I}*
    act2 : *i1 := i1 $\cup$ {num_I $\mapsto$ a}*
    act3 : *i2 := i2 $\cup$ {num_I $\mapsto$ b}*
    act4 : *i4(num_I) := FALSE*
    act5 : msg := msg $\cup$ { m}
    act6 : msg1 := msg1 $\cup$ { m}
    act7 : m1_id := m1_id $\cup$ {m $\mapsto$ a}
    act8 : m1_ki := m1_ki $\cup$ {m $\mapsto$ num_I}
  END

Step2 $\widehat{=}$
REFINES level
ANY s, m2
  WHERE
    grd1 : *num_I $\in$ num_A*
    grd2 : *a = i1(num_I)*
    grd3 : *b = i2(num_I)*
    grd4 : s $\in$ Server
    grd5 : m2 $\notin$ msg
  THEN
    act1 : *i4(num_I) := TRUE*
    act2 : msg := msg $\cup$ { m2}
    act3 : msg2 := msg2 $\cup$ { m2}
    act4 : m2_id := m2_id $\cup$ {m2 $\mapsto$ s}
  END

```
Step3 ≙
ANY m2, m3, key
    WHERE
      grd1 : m3 ∉ msg
      grd2 : m2 ∈ msg2
      grd3 : key ∉ session_key
    THEN
      act1 : msg := msg ∪ { m3}
      act2 : msg3 := msg3 ∪ { m3}
      act3 : session_key := session_key ∪ { key}
    END
```

```
Step4 ≙
ANY key, m4, s
    WHERE
      grd1 : key ∈ session_key
      grd2 : m4 ∉ msg
      grd3 : s ∈ Server
    THEN
      act1 : msg := msg ∪ { m4}
      act2 : msg4 := msg4 ∪ { m4}
      act3 : m4_id := m4_id ∪ { m4 ↦ s}
      act4 : m4_ki(m4) := key
    END
```

```
Final ≙
REFINES final
ANY num_I, m4, key
    WHERE
      grd1 : num_I ∈ num_A \ num_final
      grd2 : i4(num_I) =TRUE
      grd3 : m4 ∈ msg4
      grd4 : key ∈ session_key
      grd5 : m4_ki(m4) = key
    THEN
      act1 : num_final := num_final ∪ { num_I}
    END
```

For this specification, most of the proof obligations (POs) are automatically discharged by RODIN. The complete development of the TMN protocol , in Table 7.1, results in 41 POs, within which 23 are proved automatically by the RODIN tool, the remaining 18 POs are proved interactively using the RODIN tool.

Table 7.1 – Proof Statistics for the TMN Protocol Development

Model	Total POs	Automatic Proof	Interactive proof
Abstract model	20	7	13
First refinement	21	16	5
Total(%)	41(100%)	23(56%)	18(44%)

TMN Protocol Attacks

The TMN protocol is subject to a number of attacks. In[166] G. Lowe and A. W. Roscoe discover 3 different attacks:

1. Authentication and secrecy failure: the intruder I pretend to be A, and uses a session supplementary key K_i of his selection to discover the established session key K_b in the last message.
   (a) I(A) → S : B, $\{K_i\}$PK(S)
   (b) S → B : A
   (c) B → S : A, $\{K_b\}$PK(S)
   (d) S → I(A) : B, $\{K_b\}K_i$
2. Authentication failure: the intruder I pretend to be B and establishes a new session key $K_i$of his selection
   (a) A→ S : B, $\{K_a\}$PK(S)
   (b) S → I(B) : A
   (c) I(B) → S : A, $\{K_i\}$PK(S)
   (d) S → I(A) : B, $\{K_i\}K_a$

Also, in [166] a third attacks Parallel session and replay attack combining the above attacks 1 and 2. In [167], Simmon found an attack to TMN by using the homomorphic property of the underlying public key algorithm.an attacker can discover an exchanger session key easily, and the server cannot expose any message reply.

7.4 A Modified Version of TMN Protocol

We propose in this section a modified version of TMN protocol. The user A will encrypts two different number with the server's public key.And user B will have another key (PK_{bs}) which will be shared with a server only. The user will encrypts the session key first with (PK_{bs}) then encrypted with server's public key, and sends it to the server. The server for him will decrypt the response, encrypts the key with both A's random number and sends the result to A. The Modified protocol can be defined as:

(i) A → S: B, $\{K_{as}\}PK_s, \{K_{a's}\}PK_s$

(ii) S → B: A

(iii) B → S: A, $\{K_{ab}, PK_{bs}\}PK_s$

(iv) S→ A: B, $\{K_{ab}\}K_{as} \wedge \{K_{ab}\}K_{a's}$

Event-B Model of the Modified Version

The specification of this new version of the protocol is almost the same like the previous one, the only difference is that we have two new variables, num_A1 which defines the second random number of A, and m(3)_k32 represents the key that B shared with the server. The strategy of the refinement will be the same, but the model will be enriched with more guards and actions. For the new specification,table 7.2

Table 7.2 – Proof Statistics for the Modified Version of TMN Protocol

Model	Total POs	Automatic Proof	Interactive proof
Abstract model	35	27	8
First refinement	30	25	5
Total(%)	65(100%)	52(80%)	13(20%)

7.5 Summary

This chapter describes the Tatebayashi, Matsuzaki, and Newman (TMN) protocol, which is proposed mainly as a cryptographic protocol for mobile communication systems. We model TMN using Event-B method and verify the correctness of refinement steps until the implementation. Also, We proposed a modified version of the protocol and applied the Event-B method to the new version.

8 Conclusion and Future Work

This chapter summarizes the thesis result and introduces possible future works. First, it is illustrated how research goals are fulfilled and are covered in different chapters. Finally, possible future work and research are listed.

8.1 Summary

Formal methods are mathematical techniques applied for developing large systems. The complexity of growing systems presents an increasing challenge in the task of formal development and requires a significant improvement of formal techniques and tool support. More and more various and complex communication protocols are being employed in distributed systems and computer networks for divers types. The informal techniques used to design these protocols have been generally successful, but have also yielded a disturbing number of errors or unexpected and undesirable behavior in most protocols.

Communication protocols usually involve a subtle interaction of a number of distributed components and have a high degree of parallelism. This is why their correctness is difficult to guarantee, and many protocols turned out to be erroneous. One of the most promising solutions to this problem is the use of formal verification, which requires the correct specification of the protocol in some specification language and a formal proof of its correctness by mathematical techniques. Different approaches have been used for the formal specification and verification of communication protocols. In our approach, we have chosen the Event-B method. Event-B, a formal specification language, has a high potential in dealing with the correctness due to its well-known refinement mechanism, well-defined proof obligations, and the RODIN platform.

The main reason to choose Event B as a modeling language is the refinement, which allows a progressive development of models. The refinement in Event-B is a mechanism of constructing a series of more abstract models before reaching a very detailed one. For instance, in a refinement step, new variables and new events can be introduced and the existing events can be made more concrete with the assumption (that must be formally proved) that the concrete guard is not weaker than the abstract one (i.e. the concrete guard logically implies the abstract one) [9]. Event B also is supported by a complete tool set: RODIN which provides features like refinement, proof obligations generation, proof assistants and model-checking facilities.

The most important condition to successfully complete verification is to know what exactly the verification task needs to achieve. This seems to be trivial. However, in practice, formal methods are often not applied due to a lack of awareness of the verification goal. In contrast to simulation where certain behaviors of the system are tested, formal verification needs an abstract understanding of the properties of the system. The protocols described in our thesis cover a wide range, from data link layer protocol, e-commerce protocol to a cryptographic protocol. In order to specify our protocols, we describe what the protocol should do, what the protocols should not do, and how the protocol should react. Our aims are constructing a model with a clear and accurate formulation of the communication protocol properties and discharge of all proof obligations. To satisfy these, attentive choice of invariants and machine theorems was important and eased the proof effort. A major focus of our work has been to explore the use of the Event-B method for formally specifying a communication protocol. Two major approaches from the general specification were identified and applied to a set of example protocols: (1) an abstract model that defines operations that may be caused and (2) concrete model where the services is an active process with message exchange.

The main contribution of this research is to demonstrate the applicability of Event-B method in the modeling and verification of properties in real-world protocols, specifically communication protocols. In conclusion, the Event-B method aims at improving the quality of the software with the overall gain in the productivity. The work presented in this thesis is a step towards this direction.

Our experience of using the RODIN platform was very positive. The supported tool was sufficiently expressive and all proof obligations could be discharged. We achieved a good degree of automatic proof. All interactive proofs involved a small number of steps and were straightforward to achieve.

Based on this experience, here are a few hints to get it right from the start:

Which Proofs Need to Be / Have Been Discharged? RODIN keeps track of all the proofs that need to be done and tries to prove them every time the file is saved. Their state can be seen by opening the Event B Explorer view (if not already open) and expanding the model, then the machine, then the Proof Obligations section. Discharged POs appear in green, POs still not discharged appear in brown.

Automatic Provers: RODIN sometimes needs manual help to prove some properties. Using the interface to the proving tools available for RODIN is not straightforward. However, it is not hard to write a model for the proposed problem in which RODIN automatically discharges all the proof obligations.

Building the Model: The starting point is figuring out which variables are necessary and which invariants can be used to capture the requirements. Some requirements can be dealt with invariants which are similar to type declarations (e.g., $a \in \mathbb{N}$). Others need logical formulas. Some requirements will however have to be dealt with in the guards of some events. Since Event-B requires invariant preservation, the more requirements are expressed as invariants, the stronger the model will be. Last, some invariants may not capture directly requirements, but properties of the model which we want to ensure are preserved. They may also help RODIN

to perform the proofs. Then determine which events should be observed in the model and fill in obvious guards and the actions one event at a time. Invariant preservation and well definedness proof obligations are proven on a per-event basis, so you can in principle work event by event trying to capture requirements. RODIN will retry simple proof strategies every time we save the project.

Seeing the Proofs: When a proof cannot be discharged automatically it is anyway useful to see in which node the proof stopped — i.e., what had to be proven but could not be proved. This can give hints as to what is missing in the model. In order to do this, double click on the brown undischarged proof in the Event B explorer window. That will open new tab related to the proof. Then we switch to the "Proving perspective" by selecting Window → Open Perspective → Proving. A "Proof Tree" window appear stating the formula remaining to be proven; a "Proof Control" window should appear as well, where we can add intermediate goals to be proven, hypotheses in the search tree, and select additional strategies from the installed theorem provers (e.g. pp, p1, ml). They can be tried, but they can take a long time to execute — until timeout (this is the reason why they are not used every time the project is saved).

At last, some versions of RODIN have simple bug: even if a proof is discharged, it is still shown brown in the Event-B explorer, so one would think it has not been proved. A simple quit and restart RODIN to make it show the correct status. Although the different parts of the guards are in logical conjunction, sometimes changing the order of the guards help the theorem provers to do their task.

8.2 Future Work

Guaranteeing software correctness "has long been the objective in Computer Science" ([168, 169]). Formal methods may be used beneficially for the specification, verification, and implementation of communication protocols. However, a great deal of work remains to be done in improving verification techniques and high-level system implementation languages, in integrating performance analysis with analysis for logical correctness, and in automating these analysis techniques. To guarantee the correctness of security protocol system, several formal modeling and analysis methods have been developed. These works are focused on validating the protocol specification. However, errors can also be introduced to the system in implementation phases, even if the specification is proven to be flawless. The formal verification technique applied to routing algorithms for wireless networks has been a quite unexplored field yet, and therefore there are lots of opportunities for new research. Indeed, the field is in need of more specific techniques and tools.

Routing is one of the most basic and valuable tasks in a collaborative computer network. Since the basic networking function of the routing protocols is to establish procedures for finding and maintenance of paths between different points on the Internet. Errors in routing protocols can lead to connection interruption which can be very expensive, given the existent volume of traffic circling through the Internet.

To any wireless network, having a correct, robust and efficient routing protocol is indispensable. However, a challenging problem is how to guarantee these desirable features. Neither simula-

tions nor tested implementations can ensure the quality required for these protocols. Because of that, it is important to verify, before a protocol is used, that its behavior will conform to expectations. This analysis can be carried out along two main directions: performance and correctness. It is essential to note that formal verification is not a substitute for testing or simulation. These three quality assurance techniques are much more complementary rather than competitive approaches. They should be adopted together to boost the system reliability once each one has a different approach and objective. The test is a way to think how the system works trying to detect situations where it may fail. Since we have a pretty good experience with Event-B and Rodin platform, we want to enlarge our knowledge to include other methods and techniques. So, what we propose for our next work methodology, is to apply tools for interactive theorem proving, which they have various logics and various levels of automation, three tools that interest us, which are:

- **First order logic without quantifiers, with induction** :

  A Computational Logic for Applicative Common Lisp (ACL2)[1] is an interactive theorem prover. It combines a Lisp-based programming language for developing formal models of systems with a reasoning engine that can prove properties about these models. We propose to use ACL2 with DrACuLa (an ACL2 IDE) [170] to create models and properties of routing protocols, to verify these properties with its mechanical prover, and to use its execution capability for effective simulation.

- **Higher order logic**:

  Prototype Verification System (PVS)[2] is a specification language with support tools and theorem prover.The Language of PVS is based on classical, strongly typed higher-order logic, it provides some integration with model-checking. We consider applying PVS to produce a detailed, precise and highly formal model and within the proposed model, we intend to be able to establish a key correctness norm.

- **Calculus of inductive constructions** :

  Coq[3] implements a program specification and mathematical higher-level language called Gallina that is based on an expressive formal language called the Calculus of Inductive Constructions that itself combines both a higher-order logic and a richly-typed functional programming language. Since in the theorem proving there is no counterexample. We will explore the use of "QuickChick" which is a property-based random testing tool for Coq that allows the user to quickly find the counterexamples to proposition before attempting to prove them. This is an approach where testing meets proving.

We intend to prove the overall purpose of the protocol and then check for more elaborate specifications that reflect the local behavior of the processes. This type of reasoning with these tools will make it possible to lead development on less and less abstract levels. We will be focusing on proving properties about the protocol, then apply it to all possible behaviors. In order to guarantee the correctness of a routing protocol, we need to be capable of formally express the requirements. Given a specification and a set of requirements, our aim is to explore the use of the cited tools and to verify whether the specification satisfies the requirement using these tools.

[1] http://www.cs.utexas.edu/ moore/acl2/
[2] http://pvs.csl.sri.com/
[3] http://coq.inria.fr/

A Event-B Model of Sliding Window Protocol

A.1 Abstract Model

CONTEXT C7
SETS
    Max
CONSTANTS
    max_retransmit
AXIOMS
    axm1: $max_retransmit \in \mathbb{N}$
    axm2: $max_retransmit > 0$
END

CONTEXT C6
SETS
    life
CONSTANTS
    exist
    never_sent
    lost
AXIOMS
    axm1: $life = \{exist, never_sent, lost\}$
    axm2: $exist \neq never_sent$
    axm3: $exist \neq lost$
    axm4: $never_sent \neq lost$
END

CONTEXT C4
SETS
    size_r
CONSTANTS
    RWS
AXIOMS
    axm1: $RWS = 1$
END

CONTEXT C5

EXTENDS C4
SETS
    size_s
CONSTANTS
    SWS
AXIOMS
    axm1: $SWS \in \mathbb{N}$
    axm2: $SWS \geq RWS$
END

CONTEXT C8
EXTENDS C5
CONSTANTS
    sequence_num
AXIOMS
    axm1: $sequence_num \in \mathbb{N}$
    axm2: $sequence_num > SWS + 1$
END

CONTEXT C3
SETS
    frame_statut
CONSTANTS
    acked
    nacked
    nsent
AXIOMS
    axm1: $frame_statut = \{acked, nacked, nsent\}$
    axm2: $acked \neq nacked$
    axm3: $nacked \neq nsent$
    axm4: $acked \neq nsent$
END

CONTEXT C0
SETS
    D
CONSTANTS
    n
    f
AXIOMS
    axm1: $0 < n$
    axm2: $f \in 1\,..\,n \longrightarrow D$
END

CONTEXT C1
SETS
    STATUS
CONSTANTS
    working
    success
    fail

AXIOMS

axm1: $STATUS = \{working, success, fail\}$
axm2: $working \neq success$
axm3: $working \neq fail$
axm4: $success \neq fail$

END

MACHINE gbn_0
SEES C0
VARIABLES

i
g

INVARIANTS

inv1: $i \in 0 \mathinner{..} n$
inv2: $g \in 1 \mathinner{..} i \gtrless D$

EVENTS
Initialisation

begin

act1: $i := 0$
act2: $g := \emptyset$

end

Event swp ⟨ordinary⟩ $\widehat{=}$

begin

act1: $i, g :| i' \in 0 \mathinner{..} n \land g' = (1 \mathinner{..} i') \lhd f$

end

END

A.2 First Refinement

MACHINE gbn_1
REFINES gbn_0
SEES C0,C1
VARIABLES

h , k, s_st, r_st

INVARIANTS

inv1 : $k \in 0 \mathinner{..} n$
inv2 : $h = (1 \mathinner{..} k) \lhd f$
inv4 : $s_st = success \Rightarrow r_st = success$
inv5 : $r_st \in STATUS$
inv6 : $s_st \in STATUS$
inv7 : $h \subseteq f$
inv8 : $dom(h) \subseteq 1 \mathinner{..} n$

EVENTS
Initialisation

begin

act1: $k := 0$
act2: $h := \emptyset$
act3: $r_st := working$
act4: $s_st := working$

end

Event swp ⟨ordinary⟩ $\widehat{=}$
refines swp

when

grd1: $s_st \neq working$
grd2: $r_st \neq working$

with

i': $i' = k$

then g': $g' = h$
end *skip*
Event send_success ⟨ordinary⟩ $\widehat{=}$
when
grd1: $s_st = working$
grd2: $r_st = success$
then
act1: $s_st := success$
end
Event send_fail ⟨ordinary⟩ $\widehat{=}$
when
grd1: $s_st = working$
then
act1: $s_st := fail$
end
Event receiv_success ⟨ordinary⟩ $\widehat{=}$
when
grd1: $r_st = working$
then
act1: $r_st := working$
end
Event receiv_fail ⟨ordinary⟩ $\widehat{=}$
when
grd1: $r_st = working$
grd2: $s_st = fail$
then
act1: $r_st := fail$
end
END

A.3 Second Refinement

MACHINE gbn_2
REFINES gbn_1
SEES C0,C1
VARIABLES
h, k, s_st, r_st
INVARIANTS
inv1: $k < n \Rightarrow k + 1 \notin dom(h)$
inv2: $k \in 0 \,..\, n$
EVENTS
Event receiv_success ⟨ordinary⟩ $\widehat{=}$
extends receiv_success
when
grd1: $r_st = working$
grd2: $k + 1 = n$
grd3: $k + 1 \notin dom(h)$
then
act1: $r_st := working$
act2: $k := n$
act3: $h := h \cup \{n \mapsto f(n)\}$
end
Event receiv_current_data ⟨ordinary⟩ $\widehat{=}$
refines receiv_success
when
grd1: $r_st = working$
grd2: $k + 1 < n$
then
act1: $k := k + 1$
act2: $h := h \cup \{k + 1 \mapsto f(k + 1)\}$
end
END

A.4 Third Refinement

MACHINE gbn_3
REFINES gbn_2
SEES C0,C1,C3,C5
VARIABLES

h, k, s_st, r_st, buffer_s, data_chan, ack_chan, w

INVARIANTS

inv1: $buffer_s \in 0\,..\,n$
inv2: $data_chan \in 1\,..\,n \nrightarrow D$
inv3: $ack_chan \subseteq 1\,..\,n$
inv4: $w \in 1\,..\,n \nrightarrow frame_statut$
inv5: $w \in \mathbb{Z} \nrightarrow frame_statut$
inv6: $f \in \mathbb{Z} \nrightarrow D$

EVENTS
Initialisation ⟨extended⟩

begin

act5: $buffer_s := 0$
act6: $data_chan := \emptyset$
act7: $ack_chan := \emptyset$
act8: $w := 1\,..\,n \times \{nsent\}$

end

Event send_success ⟨ordinary⟩ $\widehat{=}$
extends send_success

any

wherea

grd1: $s_st = working$
grd2: $r_st = success$
grd3: $buffer_s + 1 \in ack_chan$
grd4: $a \in 1\,..\,n$

then

act1: $s_st := success$
act2: $w(a) := acked$
act3: $ack_chan := ack_chan \setminus \{buffer_s + 1\}$
act4: $buffer_s := buffer_s + 1$

end

Event receiv_success ⟨ordinary⟩ $\widehat{=}$
extends receiv_success

when

grd1: $r_st = working$
grd2: $k + 1 = n$
grd3: $k + 1 \notin dom(h)$
grd4: $k + 1 \in dom(data_chan)$

then

act1: $r_st := working$
act2: $k := n$
act3: $h := h \cup \{n \mapsto f(n)\}$
act4: $ack_chan := ack_chan \cup \{k + 1\}$
act5: $data_chan := \{k + 1\} \lhd data_chan$

end

Event receiv_sn_ack ⟨ordinary⟩ $\widehat{=}$

any

whereframe

grd1: $frame \in dom(h)$
grd2: $r_st = working$
grd3: $frame \leq k + RWS$
grd4: $w(frame) = nacked$

then

act1: $ack_chan := ack_chan \cup \{frame\}$
act2: $data_chan := \{frame\} \lhd data_chan$

end

Event rcv_ignor_data_out_win ⟨ordinary⟩ $\widehat{=}$

any

where frame

grd1: $frame \in dom(data_chan)$
grd2: $r_st = working$
grd3: $frame > k + RWS$

then

act1: $data_chan := \{frame\} \lhd data_chan$

end

Event sender_rcv_ack_advanc_win ⟨ordinary⟩ $\widehat{=}$

when

grd1: $s_st = working$
grd2: $buffer_s + 1 < n$
grd3: $buffer_s \in ack_chan$

then

act1: $ack_chan := ack_chan \setminus \{buffer_s + 1\}$
act2: $w(buffer_s) := acked$
act3: $buffer_s := buffer_s + 1$

end

Event sender_rcv_ack ⟨ordinary⟩ $\widehat{=}$

any

where frame

grd1: $frame \in ack_chan$
grd2: $s_st = working$
grd3: $frame \neq buffer_s + 1$

then

act1: $w(frame) := acked$
act2: $ack_chan := ack_chan \setminus \{frame\}$

end

Event sender_send_data ⟨ordinary⟩ $\widehat{=}$

when

grd1: $s_st = working$
grd2: $k + 1 < n$
grd3: $k + 1 < buffer_s + SWS$
grd4: $k \in dom(f)$

then

act1: $data_chan := data_chan \cup \{k \mapsto f(k)\}$
act2: $w(k) := nacked$
act3: $k := k + 1$

end

END

A.5 Fourth Refinement

MACHINE gbn_4
REFINES gbn_3
SEES C0,C1,C3,C5,C6,C7
VARIABLES

h, k, s_st, r_st, buffer_s, data_chan, ack_chan, w, unreliable_data, unreliable_ack, frame_life, retries

INVARIANTS

inv1: $unreliable_data \in 1 \,..\, n \rightarrowtail D$
inv2: $unreliable_ack \subseteq 1 \,..\, n$
inv3: $frame_life \in 1 \,..\, n \longrightarrow life$
inv4: $retries \in 1 \,..\, n \longrightarrow \mathbb{N}$
inv5: $\forall frame \cdot frame \in 1 \,..\, n \land retries(frame) > max_retransmit \Rightarrow r_st = fail$
inv6: $s_st = fail \Rightarrow (\exists p \cdot p \in 1 \,..\, n \land retries(p) > max_retransmit)$

EVENTS

Initialisation ⟨extended⟩

begin

act9: $unreliable_data := \emptyset$
act10: $unreliable_ack := \emptyset$
act11: $frame_life := 1 \,..\, n \times \{never_sent\}$
act12: $retries := 1 \,..\, n \times \{0\}$

end
Event send_success ⟨ordinary⟩ $\widehat{=}$
extends send_success
any
where[a]
grd1: $s_st = working$
grd2: $r_st = success$
grd3: $buffer_s + 1 \in ack_chan$
grd4: $a \in 1 \,..\, n$
grd5: $buffer_s + 1 \in unreliable_ack$
then
act1: $s_st := success$
act2: $w(a) := acked$
act3: $ack_chan := ack_chan \setminus \{buffer_s + 1\}$
act4: $buffer_s := buffer_s + 1$
act5: $unreliable_ack := unreliable_ack \setminus \{buffer_s + 1\}$
end
Event send_fail ⟨ordinary⟩ $\widehat{=}$
extends send_fail
any
where[frame]
grd1: $s_st = working$
grd2: $frame \in 1 \,..\, n$
grd3: $retries(frame) = max_retransmit$
grd4: $frame_life(frame) = lost$
grd5: $retries(frame) > max_retransmit$
then
act1: $s_st := fail$
act2: $retries(frame) := retries(frame) + 1$
end
Event receiv_success ⟨ordinary⟩ $\widehat{=}$
extends receiv_success
when
grd1: $r_st = working$
grd2: $k + 1 = n$
grd3: $k + 1 \notin dom(h)$
grd4: $k + 1 \in dom(data_chan)$
grd5: $k + 1 \in dom(unreliable_data)$
then
act1: $r_st := working$
act2: $k := n$
act3: $h := h \cup \{n \mapsto f(n)\}$
act4: $ack_chan := ack_chan \cup \{k + 1\}$
act5: $data_chan := \{k + 1\} \lhd data_chan$
act6: $unreliable_ack := unreliable_ack \cup \{k + 1\}$
act7: $unreliable_data := \{k + 1\} \lhd unreliable_data$
end
Event receiv_fail ⟨ordinary⟩ $\widehat{=}$
extends receiv_fail
any
where[frame]
grd1: $r_st = working$
grd2: $s_st = fail$
grd3: $frame \in 1 \,..\, n$
grd4: $retries(frame) > max_retransmit$
then
act1: $r_st := fail$
end
Event receiv_sn_ack ⟨ordinary⟩ $\widehat{=}$
extends receiv_sn_ack
any
where[frame]
grd1: $frame \in dom(h)$
grd2: $r_st = working$
grd3: $frame \leq k + RWS$

grd4: $w(frame) = nacked$
then grd5: $frame \in dom(unreliable_data)$

act1: $ack_chan := ack_chan \cup \{frame\}$
act2: $data_chan := \{frame\} \lhd data_chan$
act3: $unreliable_ack := unreliable_ack \cup \{frame\}$
end act4: $unreliable_data := \{frame\} \lhd unreliable_data$

Event rcv_ignor_data_out_win ⟨ordinary⟩ $\widehat{=}$
extends rcv_ignor_data_out_win

any

where *frame*

grd1: $frame \in dom(data_chan)$
grd2: $r_st = working$
grd3: $frame > k + RWS$
then grd4: $frame \in dom(unreliable_data)$

act1: $data_chan := \{frame\} \lhd data_chan$
end act2: $unreliable_data := \{frame\} \lhd unreliable_data$

Event sender_rcv_ack_advanc_win ⟨ordinary⟩ $\widehat{=}$
extends sender_rcv_ack_advanc_win

when

grd1: $s_st = working$
grd2: $buffer_s + 1 < n$
grd3: $buffer_s \in ack_chan$
then grd4: $buffer_s + 1 \in unreliable_ack$

act1: $ack_chan := ack_chan \setminus \{buffer_s + 1\}$
act2: $w(buffer_s) := acked$
act3: $buffer_s := buffer_s + 1$
end act4: $unreliable_ack := unreliable_ack \setminus \{buffer_s + 1\}$

Event sender_rcv_ack ⟨ordinary⟩ $\widehat{=}$
extends sender_rcv_ack

any

where *frame*

grd1: $frame \in ack_chan$
grd2: $s_st = working$
grd3: $frame \neq buffer_s + 1$
then grd4: $frame \in unreliable_ack$

act1: $w(frame) := acked$
act2: $ack_chan := ack_chan \setminus \{frame\}$
end act3: $unreliable_ack := unreliable_ack \setminus \{frame\}$

Event DMN_data_channel ⟨ordinary⟩ $\widehat{=}$

any

where frame

then grd1: $frame \in dom(unreliable_data)$

act1: $unreliable_data := \{frame\} \lhd unreliable_data$
end act2: $frame_life(frame) := lost$

Event DMN_ack_channel ⟨ordinary⟩ $\widehat{=}$

any

where frame

then grd1: $frame \in unreliable_ack$

act1: $unreliable_ack := unreliable_ack \setminus \{frame\}$
end act2: $frame_life(frame) := lost$

Event SND_resnd_data ⟨ordinary⟩ $\widehat{=}$

any

where frame

grd1: $frame \in 1 \,..\, n$
grd2: $s_st = working$
grd3: $frame_life(frame) = lost$
grd4: $retries(frame) < max_retransmit$

then

act1: $unreliable_data := unreliable_data \cup \{frame \mapsto f(frame)\}$
act2: $frame_life(frame) := exist$
act3: $retries(frame) := retries(frame) + 1$

end

END

A.6 Fifth Refinement

MACHINE gbn_5
REFINES gbn_4
SEES C0,C1,C3,C5,C6,C7
VARIABLES

h, k, s_st, r_st, buffer_s, data_chan, ack_chan, w, unreliable_data, unreliable_ack, frame_life, retries, time, at, sleep, send_time_msg, propagation, sleep_time

INVARIANTS

inv1: $time \in \mathbb{N}$
inv2: $at \subseteq \mathbb{N}$
inv3: $at \neq \emptyset \Rightarrow time \leq min(at)$
inv4: $sleep \in \mathbb{N}$
inv5: $send_time_msg \in \mathbb{N}$
inv6: $propagation \in \mathbb{N}$
inv7: $sleep_time \in \mathbb{N}$
inv8: $s_st = success \wedge send_time_msg + propagation \notin at \wedge time \geq send_time_msg + propagation \Rightarrow r_st = success$
inv9: $s_st = working \wedge at = \{sleep\} \Rightarrow time \geq send_time_msg + propagation$

EVENTS

Initialisation ⟨extended⟩

begin

act13: $time := 0$
act14: $at := \emptyset$
act15: $sleep := 0$
act16: $send_time_msg := 0$
act17: $propagation := 0$
act18: $sleep_time := 0$

end

Event send_success ⟨ordinary⟩ $\widehat{=}$
extends send_success

any

where[a]

grd1: $s_st = working$
grd2: $r_st = success$
grd3: $buffer_s + 1 \in ack_chan$
grd4: $a \in 1 \,..\, n$
grd5: $buffer_s + 1 \in unreliable_ack$

then

act1: $s_st := success$
act2: $w(a) := acked$
act3: $ack_chan := ack_chan \setminus \{buffer_s + 1\}$
act4: $buffer_s := buffer_s + 1$
act5: $unreliable_ack := unreliable_ack \setminus \{buffer_s + 1\}$
act6: $at := at \cup \{time + propagation\} \cup \{time + sleep_time\}$

end

Event receiv_success ⟨ordinary⟩ $\widehat{=}$
extends receiv_success

when

grd1: $r_st = working$

grd2: $k + 1 = n$
grd3: $k + 1 \notin dom(h)$
grd4: $k + 1 \in dom(data_chan)$
grd5: $k + 1 \in dom(unreliable_data)$
grd6: $time = send_time_msg + propagation$

then

act1: $r_st := working$
act2: $k := n$
act3: $h := h \cup \{n \mapsto f(n)\}$
act4: $ack_chan := ack_chan \cup \{k + 1\}$
act5: $data_chan := \{k + 1\} \lhd data_chan$
act6: $unreliable_ack := unreliable_ack \cup \{k + 1\}$
act7: $unreliable_data := \{k + 1\} \lhd unreliable_data$
act8: $at := at \setminus \{time\}$

end

Event receiv_current_data ⟨ordinary⟩ $\widehat{=}$
extends receiv_current_data

when

grd1: $r_st = working$
grd2: $k + 1 < n$
grd3: $time = send_time_msg + propagation$

then

act1: $k := k + 1$
act2: $h := h \cup \{k + 1 \mapsto f(k + 1)\}$
act3: $at := at \setminus \{time\}$

end

Event sender_send_data ⟨ordinary⟩ $\widehat{=}$
extends sender_send_data

when

grd1: $s_st = working$
grd2: $k + 1 < n$
grd3: $k + 1 < buffer_s + SWS$
grd4: $k \in dom(f)$

then

act1: $data_chan := data_chan \cup \{k \mapsto f(k)\}$
act2: $w(k) := nacked$
act3: $k := k + 1$
act4: $at := at \cup \{time + propagation\} \cup \{time + sleep_time\}$

end

Event SND_resnd_data ⟨ordinary⟩ $\widehat{=}$
extends SND_resnd_data

any

where *frame*

grd1: $frame \in 1 \,..\, n$
grd2: $s_st = working$
grd3: $frame_life(frame) = lost$
grd4: $retries(frame) < max_retransmit$

then

act1: $unreliable_data := unreliable_data \cup \{frame \mapsto f(frame)\}$
act2: $frame_life(frame) := exist$
act3: $retries(frame) := retries(frame) + 1$
act4: $at := at \cup \{time + propagation\} \cup \{time + sleep_time\}$

end

Event Time ⟨ordinary⟩ $\widehat{=}$

time progression

any

where tm

grd1: $tm \in \mathbb{N}$
grd2: $tm > time$
grd3: $at \neq \emptyset \Rightarrow tm \leq min(at)$

then

act1: $time := tm$

end

END

A.7 Sixth Refinement

MACHINE gbn_6
REFINES gbn_5
SEES C0,C1,C3,C6,C7,C8
VARIABLES

h, k , s_st, r_st, buffer_s, data_chan, ack_chan, w, unreliable_data, unreliable_ack, frame_life, retries, time, at, sleep, send_time_msg, propagation, sleep_time

INVARIANTS

inv1: $\forall frame1, frame2 \cdot frame1 \in dom(unreliable_data) \land frame2 \in dom(unreliable_data) \land (frame1 mod sequence_num = frame2 mod sequence_num) \Rightarrow frame1 = frame2$

EVENTS

Event send_success ⟨ordinary⟩ $\widehat{=}$
extends send_success

any
a
seq_num
where
grd1: $s_st = working$
grd2: $r_st = success$
grd3: $buffer_s + 1 \in ack_chan$
grd4: $a \in 1 \,..\, n$
grd5: $buffer_s + 1 \in unreliable_ack$
grd6: $seq_num \in unreliable_ack$
grd7: $seq_num = buffer_s + 1 mod sequence_num$
then
act1: $s_st := success$
act2: $w(a) := acked$
act3: $ack_chan := ack_chan \setminus \{buffer_s + 1\}$
act4: $buffer_s := buffer_s + 1$
act5: $unreliable_ack := unreliable_ack \setminus \{buffer_s + 1\}$
act6: $at := at \cup \{time + propagation\} \cup \{time + sleep_time\}$
end

Event receiv_success ⟨ordinary⟩ $\widehat{=}$
extends receiv_success

any
sq_num
where
grd1: $r_st = working$
grd2: $k + 1 = n$
grd3: $k + 1 \notin dom(h)$
grd4: $k + 1 \in dom(data_chan)$
grd5: $k + 1 \in dom(unreliable_data)$
grd6: $time = send_time_msg + propagation$
grd7: $sq_num \in dom(data_chan)$
grd8: $sq_num = k + 1 mod sequence_num$
then
act1: $r_st := working$
act2: $k := n$
act3: $h := h \cup \{n \mapsto f(n)\}$
act4: $ack_chan := ack_chan \cup \{k + 1\}$
act5: $data_chan := \{k + 1\} \triangleleft data_chan$
act6: $unreliable_ack := unreliable_ack \cup \{k + 1\}$
act7: $unreliable_data := \{k + 1\} \triangleleft unreliable_data$
act8: $at := at \setminus \{time\}$
end

Event receiv_sn_ack ⟨ordinary⟩ $\widehat{=}$
extends receiv_sn_ack

any
frame
seq_num
where
grd1: $frame \in dom(h)$
grd2: $r_st = working$
grd3: $frame \leq k + RWS$
grd4: $w(frame) = nacked$

grd5: $frame \in dom(unreliable_data)$
grd6: $seq_num \in dom(unreliable_data)$
then grd7: $seq_num = frame mod sequence_num$

act1: $ack_chan := ack_chan \cup \{frame\}$
act2: $data_chan := \{frame\} \lhd data_chan$
act3: $unreliable_ack := unreliable_ack \cup \{frame\}$
end act4: $unreliable_data := \{frame\} \lhd unreliable_data$

Event rcv_ignor_data_out_win ⟨ordinary⟩ $\widehat{=}$
extends rcv_ignor_data_out_win
any
frame
where seq_num
grd1: $frame \in dom(data_chan)$
grd2: $r_st = working$
grd3: $frame > k + RWS$
grd4: $frame \in dom(unreliable_data)$
grd5: $seq_num \in dom(data_chan)$
then grd6: $seq_num = frame mod sequence_num$

act1: $data_chan := \{frame\} \lhd data_chan$
end act2: $unreliable_data := \{frame\} \lhd unreliable_data$

Event sender_rcv_ack_advanc_win ⟨ordinary⟩ $\widehat{=}$
extends sender_rcv_ack_advanc_win
any
where seq_num
grd1: $s_st = working$
grd2: $buffer_s + 1 < n$
grd3: $buffer_s \in ack_chan$
grd4: $buffer_s + 1 \in unreliable_ack$
grd5: $seq_num \in unreliable_ack$
then grd6: $seq_num = buffer_s + 1 mod sequence_num$

act1: $ack_chan := ack_chan \setminus \{buffer_s + 1\}$
act2: $w(buffer_s) := acked$
act3: $buffer_s := buffer_s + 1$
end act4: $unreliable_ack := unreliable_ack \setminus \{buffer_s + 1\}$

Event sender_rcv_ack ⟨ordinary⟩ $\widehat{=}$
extends sender_rcv_ack
any
frame
where seq_num
grd1: $frame \in ack_chan$
grd2: $s_st = working$
grd3: $frame \neq buffer_s + 1$
grd4: $frame \in unreliable_ack$
grd5: $seq_num \in ack_chan$
then grd6: $seq_num = frame mod sequence_num$

act1: $w(frame) := acked$
act2: $ack_chan := ack_chan \setminus \{frame\}$
end act3: $unreliable_ack := unreliable_ack \setminus \{frame\}$

Event sender_send_data ⟨ordinary⟩ $\widehat{=}$
extends sender_send_data
any
where seq_num
grd1: $s_st = working$
grd2: $k + 1 < n$
grd3: $k + 1 < buffer_s + SWS$
grd4: $k \in dom(f)$
grd5: $seq_num \in dom(data_chan)$
then grd6: $seq_num = k mod sequence_num$

act1: $data_chan := data_chan \cup \{k \mapsto f(k)\}$
act2: $w(k) := nacked$
act3: $k := k + 1$
act4: $at := at \cup \{time + propagation\} \cup \{time + sleep_time\}$

end

Event DMN_data_channel ⟨ordinary⟩ $\widehat{=}$
extends DMN_data_channel

any
frame
seq_num
where
grd1: $frame \in dom(unreliable_data)$
grd2: $seq_num \in dom(data_chan)$
grd3: $seq_num = frame mod sequence_num$
then
act1: $unreliable_data := \{frame\} \lhd unreliable_data$
act2: $frame_life(frame) := lost$
end

Event DMN_ack_channel ⟨ordinary⟩ $\widehat{=}$
extends DMN_ack_channel

any
frame
seq_num
where
grd1: $frame \in unreliable_ack$
grd2: $seq_num \in ack_chan$
grd3: $seq_num = frame mod sequence_num$
then
act1: $unreliable_ack := unreliable_ack \setminus \{frame\}$
act2: $frame_life(frame) := lost$
end

Event SND_resnd_data ⟨ordinary⟩ $\widehat{=}$
extends SND_resnd_data

any
frame
seq_num
where
grd1: $frame \in 1\,..\,n$
grd2: $s_st = working$
grd3: $frame_life(frame) = lost$
grd4: $retries(frame) < max_retransmit$
grd5: $seq_num \in dom(unreliable_data)$
grd6: $seq_num = (frame mod sequence_num)$
then
act1: $unreliable_data := unreliable_data \cup \{frame \mapsto f(frame)\}$
act2: $frame_life(frame) := exist$
act3: $retries(frame) := retries(frame) + 1$
act4: $at := at \cup \{time + propagation\} \cup \{time + sleep_time\}$
end

END

B Event-B Model of NetBill Protocol

B.1 Abstract Model

CONTEXT ACCOUNT
SETS
ACCOUNT
CONSTANTS
Account
AXIOMS
axm1: $Account \subseteq \mathbb{N}$
axm2: $finite(Account)$
axm3: $Account \neq \emptyset$
END

CONTEXT GOODS
SETS
Goods
CONSTANTS
PRD
PRICE
KEY
AXIOMS
axm1: $PRD \subseteq \mathbb{N}$
axm2: $finite(PRD)$
axm3: $PRD \neq \emptyset$
axm4: $PRD \neq PRICE$
axm5: $PRICE \subseteq \mathbb{N}$
axm6: $PRICE \neq \emptyset$
axm7: $KEY \subseteq \mathbb{N}$
axm8: $finite(KEY)$
axm9: $KEY \neq \emptyset$
END

CONTEXT cxt1
SETS
D
CONSTANTS

n
AXIOMS
axm1: $0 < n$
END

CONTEXT Transaction
SETS
Transaction
CONSTANTS
idle
order
confirmed
delivered
cashing
ended
AXIOMS
axm1: $Transaction = \{idle, order, confirmed, delivered, cashing, ended\}$
END

MACHINE M0
SEES Agents, Transaction, GOODS
VARIABLES
customer, Merchant, NetBill_Server, goods, trans, statut, agreed
INVARIANTS
inv1: $customer \subseteq AGENTS$
inv2: $Merchant \subseteq AGENTS$
inv3: $NetBill_Server \subseteq AGENTS$
inv4: $goods \subseteq Goods$
inv5: $goods \neq \emptyset \Rightarrow (\exists g \cdot goods = \{g\})$
inv6: $trans \subseteq Transaction$
inv7: $statut \in trans \gtrless Transaction$
inv8: $agreed \in BOOL$
inv9: $\forall t \cdot t \in trans \wedge statut(t) = ended \Rightarrow agreed = TRUE$
inv10: $agreed = TRUE \Rightarrow (\forall t \cdot t \in trans \wedge statut(t) = ended)$
EVENTS
Initialisation
begin
act1: $customer := \emptyset$
act2: $Merchant := \emptyset$
act3: $NetBill_Server := \emptyset$
act4: $goods := \emptyset$
act5: $trans := \emptyset$
act6: $statut := \emptyset \times \{idle\}$
act7: $agreed := FALSE$
end
Event order ⟨ordinary⟩ $\widehat{=}$
any
whereg
grd1: $g \in goods$
grd2: $goods = \emptyset$
then
act1: $goods := \{g\}$
end
Event Terminate ⟨ordinary⟩ $\widehat{=}$
any
wheret
grd1: $t \in trans$
grd2: $agreed = TRUE$
then
act1: $statut(t) := ended$
end
END

B.2 First Refinement

MACHINE M1
REFINES M0
SEES Agents,Transaction,GOODS,ACCOUNT,cxt1
VARIABLES

customer, Merchant, NetBill_Server, goods, trans, statut, agreed, t_Id, value, encry, EPO, price

INVARIANTS

inv1: $\forall t \cdot t \in trans \Rightarrow statut(t) = idle$
inv2: $statut \in Transaction \gtrless Transaction$
inv3: $t_Id \in 0 \,..\, n$
inv4: $value \in Goods \longrightarrow \mathbb{N}$
inv5: $\forall t \cdot t \in trans \wedge statut(t) = idle \Rightarrow statut(t) = delivered$
inv6: $encry \in Goods \longrightarrow \mathbb{N}$
inv7: $\forall t \cdot t \in trans \wedge statut(t) = order \Rightarrow statut(t) = confirmed$
inv8: $EPO \in AGENTS \longrightarrow \mathbb{N}$
inv9: $price \in \mathbb{N}_1$

EVENTS
Initialisation ⟨extended⟩
begin

act1: $customer := \emptyset$
act2: $Merchant := \emptyset$
act3: $NetBill_Server := \emptyset$
act4: $goods := \emptyset$
act5: $trans := \emptyset$
act6: $statut := \emptyset \times \{idle\}$
act7: $agreed := FALSE$
act8: $t_Id := 0$
act9: $value := Goods \times \{0\}$
act10: $encry := Goods \times \{0\}$
act11: $EPO := AGENTS \times \{0\}$
act12: $price :\in \mathbb{N}_1$

end
Event order ⟨ordinary⟩ $\widehat{=}$
extends order
any
where g, m, t

grd1: $g \in goods$
grd2: $goods = \emptyset$
grd3: $m \notin Merchant$
grd4: $t \in trans$

then

act1: $goods := \{g\}$
act2: $statut(t) := order$
act3: $Merchant := Merchant \cup \{m\}$
act4: $value(g) := price$
act5: $t_Id := t_Id + 1$

end
Event start ⟨ordinary⟩ $\widehat{=}$
any
where t

grd1: $t \in Transaction \setminus trans$
grd2: $t \in dom(statut)$

then

act1: $trans := trans \cup \{t\}$
act2: $statut(t) := idle$

end
Event Goods_delivery ⟨ordinary⟩ $\widehat{=}$
any
where t, g, k

grd1: $t \in trans$

grd2: $g \in goods$
grd3: $statut(t) = idle$
grd4: $k \in KEY$

then

act1: $goods := \{g\}$
act2: $statut(t) := delivered$
act3: $value(g) := price$
act4: $encry(g) := k$

end

Event pay ⟨ordinary⟩ $\widehat{=}$

any

where t, g, k, c, ID

grd1: $t \in trans$
grd2: $g \in goods$
grd3: $statut(t) = order$
grd4: $k \in KEY$
grd5: $encry(g) = k$
grd6: $value(g) \leq price$
grd7: $c \in customer$
grd8: $ID \in KEY$

then

act1: $statut(t) := confirmed$
act2: $value(g) := price$
act3: $EPO(c) := ID$

end

Event FixPrice ⟨ordinary⟩ $\widehat{=}$

any

where amount

grd1: $amount \in \mathbb{N}_1$

then

act1: $price := amount$

end

END

B.3 Second Refinement

MACHINE M2
REFINES M1
SEES Agents,Transaction,GOODS,cxt1,ACCOUNT
VARIABLES

customer, Merchant, NetBill_Server, goods, trans, statut, agreed, t_Id, value, encry, EPO, price, decry, account

INVARIANTS

inv1: $decry \in Goods \longrightarrow \mathbb{N}$
inv2: $account \in AGENTS \longrightarrow \mathbb{N}$
inv3: $\forall t \cdot t \in trans \land statut(t) = delivered \Rightarrow statut(t) = cashing$
inv4: $\forall t \cdot t \in trans \land statut(t) = confirmed \Rightarrow statut(t) = ended$

EVENTS

Initialisation ⟨extended⟩

begin

act1: $customer := \emptyset$
act2: $Merchant := \emptyset$
act3: $NetBill_Server := \emptyset$
act4: $goods := \emptyset$
act5: $trans := \emptyset$
act6: $statut := \emptyset \times \{idle\}$
act7: $agreed := FALSE$
act8: $t_Id := 0$
act9: $value := Goods \times \{0\}$
act10: $encry := Goods \times \{0\}$
act11: $EPO := AGENTS \times \{0\}$
act12: $price :\in \mathbb{N}_1$

act13: $decry := Goods \times \{0\}$
act14: $account := AGENTS \times \{0\}$
end
Event Terminate ⟨ordinary⟩ $\widehat{=}$
extends Terminate
any
where t, g, c, m

grd1: $t \in trans$
grd2: $agreed = TRUE$
grd3: $g \in goods$
grd4: $c \in customer$
grd5: $m \in Merchant$
grd6: $price = value(g)$
grd7: $account(c) = account(c) - price$
grd8: $account(c) \neq account(m)$

then
act1: $statut(t) := ended$
act2: $account(m) := account(m) + price$
end
Event Accept ⟨ordinary⟩ $\widehat{=}$
any
where g, k, t

grd1: $g \in goods$
grd2: $k \in KEY$
grd3: $t \in trans$
grd4: $statut(t) = confirmed$

then
act1: $decry(g) := k$
act2: $statut(t) := ended$
end
Event Endored_EPO ⟨ordinary⟩ $\widehat{=}$
any
where t, m, c, ID

grd1: $t \in trans$
grd2: $m \in Merchant$
grd3: $c \in customer$
grd4: $ID \in KEY$
grd5: $statut(t) = delivered$
grd6: $EPO(c) = ID$
grd7: $EPO(m) = ID$

then
act1: $statut(t) := cashing$
act2: $EPO(c) := EPO(m)$
end
Event terminate1 ⟨ordinary⟩ $\widehat{=}$
any
where g, m, c

grd1: $g \in goods$
grd2: $m \in Merchant$
grd3: $c \in customer$
grd4: $account(c) \geq value(g)$

then
act1: $account(c) := account(c) - price$
end
END

C Event-B Model of TMN Protocol

C.1 Abstract Specification

CONTEXT cxt1
SETS

Agent
Random_num
Initiator
Server
Responder

END

CONTEXT ctx
EXTENDS cxt1
SETS

KEYS

CONSTANTS

key_s
key_a

AXIOMS

axm1: $key_s \in KEYS \longrightarrow Agent$
axm2: $key_a \in KEYS \longrightarrow Agent$

END

MACHINE M0
SEES cxt1
VARIABLES

num_A, num_final, num_B, i1, i2, i3, i4

INVARIANTS

inv1: $num_A \subseteq Random_num$
inv2: $num_final \subseteq (num_A \cup num_B)$
inv3: $num_B \subseteq Random_num$
inv4: $i1 \in num_A \longrightarrow Initiator$
inv5: $i2 \in num_A \longrightarrow Responder$
inv6: $i3 \in num_A \rightarrowtail\!\!\!\rightarrow num_B$
inv7: $i4 \in num_A \longrightarrow BOOL$
inv8: $\forall num \cdot num \in num_final \land num \in dom(i4) \Rightarrow i4(num) = TRUE$

EVENTS

Initialisation

begin

act1: $num_A := \emptyset$
act2: $num_final := \emptyset$
act3: $num_B := \emptyset$
act4: $i1 := \emptyset$
act5: $i2 := \emptyset$
act6: $i3 := \emptyset$
act7: $i4 := \emptyset$

end

Event Init ⟨ordinary⟩ $\widehat{=}$

any

where num_I, a, b

grd1: $num_I \notin num_A$
grd2: $a \in Initiator$
grd3: $b \in Responder$
grd4: $num_I \in dom(i4)$

then

act1: $num_A := num_A \cup \{num_I\}$
act2: $i1 := i1 \cup \{num_I \mapsto a\}$
act3: $i2 := i2 \cup \{num_I \mapsto b\}$
act4: $i4(num_I) := FALSE$

end

Event final ⟨ordinary⟩ $\widehat{=}$

any

where num_I

grd1: $num_I \in num_A \setminus num_final$
grd2: $i4(num_I) = TRUE$

then

act1: $num_final := num_final \cup \{num_I\}$

end

Event level ⟨ordinary⟩ $\widehat{=}$

any

where num_I, a, b

grd2: $a \in Initiator$
grd7: $num_I \in num_A$
grd3: $b \in Responder$
grd5: $a = i1(num_I)$
grd6: $b = i2(num_I)$

then

act1: $i4(num_I) := TRUE$

end

END

C.2 First Refinement

MACHINE M1
REFINES M0
SEES ctx
VARIABLES

num_A, num_final, num_B, i1, i2, i3, i4, msg, msg1, msg2, msg3, msg4, m1_id, m1_ki, m2_id, m3_id, m3_ki, m4_id, session_key, m4_ki

INVARIANTS

inv1: $msg \subseteq MSG$
inv7: $partition(msg, msg1, msg2, msg3, msg4)$
inv8: $m1_id \in msg1 \longrightarrow Initiator$
inv9: $m1_ki \in msg1 \longrightarrow num_A$
inv10: $m2_id \in msg2 \longrightarrow Server$
inv11: $m3_id \in msg3 \longrightarrow Responder$
inv12: $m3_ki \in msg3 \longrightarrow KEYS$
inv13: $m4_id \in msg4 \longrightarrow Server$

inv14: $session_key \subseteq KEYS$
inv15: $m4_ki \in msg4 \longrightarrow session_key$

EVENTS

Initialisation ⟨extended⟩

begin

act8: $msg := \emptyset$
act9: $msg1 := \emptyset$
act10: $msg2 := \emptyset$
act11: $msg3 := \emptyset$
act12: $msg4 := \emptyset$
act13: $m1_id := \emptyset$
act14: $m1_ki := \emptyset$
act15: $m2_id := \emptyset$
act16: $m3_id := \emptyset$
act17: $m3_ki := \emptyset$
act18: $m4_id := \emptyset$
act19: $session_key := \emptyset$
act20: $m4_ki := \emptyset$

end

Event Step1 ⟨ordinary⟩ $\widehat{=}$
refines Init

any

wherenum_I, a, b, m

grd1: $num_I \notin num_A$
grd2: $a \in Initiator$
grd3: $b \in Responder$
grd4: $m \notin msg$
grd5: $num_I \in dom(i4)$

then

act1: $num_A := num_A \cup \{num_I\}$
act2: $i1 := i1 \cup \{num_I \mapsto a\}$
act3: $i2 := i2 \cup \{num_I \mapsto b\}$
act4: $i4(num_I) := FALSE$
act5: $msg := msg \cup \{m\}$
act6: $msg1 := msg1 \cup \{m\}$
act7: $m1_id := m1_id \cup \{m \mapsto a\}$
act8: $m1_ki := m1_ki \cup \{m \mapsto num_I\}$

end

Event final ⟨ordinary⟩ $\widehat{=}$
extends final

any

wherenum_I, m4, key

grd1: $num_I \in num_A \setminus num_final$
grd2: $i4(num_I) = TRUE$
grd3: $m4 \in msg4$
grd4: $key \in session_key$
grd5: $m4_ki(m4) = key$

then

act1: $num_final := num_final \cup \{num_I\}$

end

Event Step2 ⟨ordinary⟩ $\widehat{=}$
extends level

any

wherenum_I, a, b, s, m2

grd2: $a \in Initiator$
grd7: $num_I \in num_A$
grd3: $b \in Responder$
grd5: $a = i1(num_I)$
grd6: $b = i2(num_I)$
grd9: $s \in Server$
grd8: $m2 \notin msg$

then

act1: $i4(num_I) := TRUE$
act2: $msg := msg \cup \{m2\}$
act3: $msg2 := msg2 \cup \{m2\}$
act4: $m2_id := m2_id \cup \{m2 \mapsto s\}$

end

Event Step3 ⟨ordinary⟩ $\widehat{=}$

any m2, m3, key, b

where

grd1: $m3 \notin msg$
grd2: $m2 \in msg2$
grd3: $key \notin session_key$
grd4: $b \in Responder$

then

act1: $msg := msg \cup \{m3\}$
act2: $msg3 := msg3 \cup \{m3\}$
act3: $session_key := session_key \cup \{key\}$
act4: $m3_id := m3_id \cup \{m3 \mapsto b\}$
act5: $m3_ki := m3_ki \cup \{m3 \mapsto key\}$

end

Event Step4 ⟨ordinary⟩ $\widehat{=}$

any key, m4, s

where

grd1: $key \in session_key$
grd2: $m4 \notin msg$
grd3: $s \in Server$

then

act1: $msg := msg \cup \{m4\}$
act2: $msg4 := msg4 \cup \{m4\}$
act3: $m4_id := m4_id \cup \{m4 \mapsto s\}$
act4: $m4_ki(m4) := key$

end

END

Bibliography

[1] Vic Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1976.

[2] Donald E. Knuth. Verification of link-level protocols. *BIT*, 21(1):31–36, 1981.

[3] Jean-Luc Richier, Carlos Rodriguez, Joseph Sifakis, and Jacques Voiron. Verification in xesar of the sliding window protocol. In Harry Rudin and Colin H. West, editors, *PSTV*, pages 235–248. North-Holland, 1987.

[4] Roope Kaivola. Using compositional preorders in the verification of sliding window protocal. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 1997.

[5] Benjamin Cox. Netbill security and transaction protocol. In *USENIX Workshop on Electronic Commerce*. USENIX Association, 1995.

[6] Makoto Tatebayashi, Natsume Matsuzaki, and David B. Newman Jr. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 324–334, 1989.

[7] Issam Al-Azzoni, Douglas G. Down, and Ridha Khédri. Modeling and verification of cryptographic protocols using coloured petri nets and *Design/CPN*. *Nord. J. Comput.*, 12(3):200–228, 2005.

[8] Gregor von Bochmann. Usage of protocol development tools: The results of a survey. In *Protocol Specification, Testing and Verification VII, Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification, Zurich, Switzerland, 5-8 May, 1987*, pages 139–161, 1987.

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

[10] Stephen Gilmore, Jane Hillston, and Marina Ribaudo. *PEPA Nets: A Structured Performance Modelling Formalism*, pages 111–130. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[11] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Softw.*, 1(1):75–88, 1984.

[12] IEEE. Ieee standard glossary of software engineering terminology. 1990.

[13] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.

[14] Nancy G. Leveson and Clark Savage Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[15] National Aeronautics and Space Administration. Mars climate orbiter mishap investigation report. Technical report, National Aeronautics and Space Administration, Washington, DC, 2000.

[16] United States General Accounting Office. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Technical report, United States General Accounting Office, 1992.

[17] J.L. Lions. Ariane 5, flight 501 failure, report of the inquiry board. Technical report, European Space Agency, July 1996.

[18] Peter G. Neumann. Cause of AT & T network failure. *The Risks Digest*, 9(62), 1990.

[19] *The Economic Impact of Inadequate Infrastructure for Software Testing*. Number Planning Report 02-3. National Institute Of Standards & Technology, May 2002. URL http://www.nist.gov/director/planning/upload/report02-3.pdf.

[20] Nancy G. Leveson. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, 41:564–575, 2004.

[21] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok. Recent catastrophic accidents: Investigating how software was responsible. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 14–22, June 2010.

[22] C. W. Johnson and C. M. Holloway. The dangers of failure masking in fault-tolerant software: Aspects of a recent in-flight upset event. In *2007 2nd Institution of Engineering and Technology International Conference on System Safety*, pages 60–65, Oct 2007.

[23] Thomas Martyn. Knight capital software upgrade costs 440 m dollars. the risks digest forum on risks to the public in computers and related systems, August 2012. URL http://catless.ncl.ac.uk/Risks/.

[24] N. Leavitt. Internet security under attack: The undermining of digital certificates. *Computer*, 44(12):17–20, Dec 2011.

[25] RODIN. *Rigorous Open Development Environment for Complex Systems*. 2013.

[26] Wen Su, Jean-Raymond Abrial, Runlei Huang, and Huibiao Zhu. From requirements to development: Methodology and example. In *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, pages 437–455, 2011.

[27] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Complementary methodologies for developing hybrid systems with event-b. In *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, pages 230–248, 2012.

[28] Michael J. Butler, Jean-Raymond Abrial, and Richard Banach. Modelling and refining hybrid systems in event-b and rodin. In *From Action Systems to Distributed Systems - The Refinement Approach.*, pages 29–42. 2016.

[29] Abdolbaghi Rezazadeh, Neil Evans, and Michael Butler. Redevelopment of an Industrial Case Study Using Event-B and Rodin. In *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry*, December 2007. URL http://eprints.ecs.soton.ac.uk/15312/.

[30] Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael J. Butler. Applying atomicity and model decomposition to a space craft system in event-b. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 328–342, 2011.

[31] Renato Silva. Lessons learned/sharing the experience of developing a metro system case study. *CoRR*, abs/1210.7030, 2012.

[32] Patrice Godefroid and David E. Long. Symbolic protocol verification with queue bdds. *Formal Methods in System Design*, 14(3):257–271, 1999.

[33] Marvin A. Sirbu and J. Doug Tygar. Netbill: an internet commerce system optimized for network-delivered services. *IEEE Personal Commun.*, 2(4):34–39, 1995.

[34] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for event-b. In *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, pages 588–605, 2006.

[35] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition edition, 2005.

[36] Michael Leuschel and Michael J. Butler. Prob: A model checker for B. In *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, pages 855–874, 2003.

[37] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael J. Butler. Decomposition tool for event-b. *Softw., Pract. Exper.*, 41(2):199–208, 2011.

[38] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[39] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[40] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[41] Dan Craigen MSc Karen Summerskill BA (eds.) Dan Craigen MSc, Karen Summerskill BA (auth.). *Formal Methods for Trustworthy Computer Systems (FM89): Report from FM89: A Workshop on the Assessment of Formal Methods for Trustworthy Computer Systems. 23–27 July 1989, Halifax, Canada*. Workshops in Computing. Springer-Verlag London, 1 edition, 1990.

[42] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, dec 1993. Also issued under the title "Formal Methods and Digital Systems Validation for Airborne Systems" as NASA Contractor Report 4551, December 1993.

[43] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[44] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[45] Anthony Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, 1996.

[46] Babak Dehbonei and Fernando Mejia. Formal methods in the railways signalling industry. In *FME '94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 24-18, 1994, Proceedings*, pages 26–34, 1994.

[47] Steve M. Easterbrook, Robyn R. Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. Software Eng.*, 24(1):4–14, 1998.

[48] Jonathan Jacky. Specifying a safety-critical control system in Z. *IEEE Trans. Software Eng.*, 21(2):99–106, 1995.

[49] Martin Neil, Gary Ostrolenk, Mary Tobin, and Mark Southworth. Lessons from using Z to specify a software tool. *IEEE Transactions on Software Engineering*, 24(1):15–23, 1998.

[50] John Wordsworth. The CICS application programming interface definition. In *Z User Workshop, Oxford, UK, Proceedings of the Fifth Annual Z User Meeting, 17-18 December 1990*, pages 285–294, 1990.

[51] J. Michael Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5):21–28, 1990.

[52] Dr. Hubert Garavel and Dr. Susanne Graf. Formal Methods for Safe and Secure Computers Systems. Technical report, Federal Office for Information Security, 2013.

[53] Gregor von Bochmann, Dave Rayner, and Colin H. West. Some notes on the history of protocol engineering. *Computer Networks*, 54(18):3197–3209, 2010.

[54] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-182957-2.

[55] Guy Leduc and François Germeau. Verification of security protocols using lotos-method and application. *Computer Communications*, 23(12):1089–1103, 2000.

[56] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, November 1995.

[57] Andres Modetl John V. Guttag, James J. Horning. Report on the larch shared language: Version 2.3. Technical Report 58, Digital Systems Research Center, California, USA, April 1990.

[58] David C. Luckham and Friedrich W. von Henke. An overview of anna, a specification language for ada. *IEEE Software*, 2(2):9–22, 1985.

[59] David Luckham. *Programming with Specifications - An Introduction to ANNA, A Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer, 1990.

[60] Sam Owre John Rushby, Friedrich von Henke. An introduction to formal specification and verification using ehdm. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, 1991.

[61] Rod M. Burstall and Joseph A. Goguen. The semantics of CLEAR, A specification language. In *Abstract Software Specifications, 1979 Copenhagen Winter School, January 22 - February 2, 1979, Proceedings*, pages 292–332, 1979.

[62] P. Klint J.A. Bergstra, J. Heering. ASF - an algebraic specification formalism. Technical Report CS-R8705, Centre for Mathematics and Computer Science (CWI),, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, January 1987.

[63] W. Fey H. Ehrig and H. Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83-03, Technische Universita, Berlin, 1983.

[64] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency, Overviews and Tutorials*, pages 510–584. 1986.

[65] Dines Bjørner. Draft final report procos. Technical Report ID/DTH DB 13/1, Department of Computer Science, Technical University of Denmark, October 1991.

[66] S. Kromodimoeljo I. Meisels B. Pase Craigen, D. and M. Saaltink. EVES: An overview. Technical Report CP-91-5402-43, Odyssey Research Associates, Ontario, Canada, 1991.

[67] M.J.C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory,Cambridge University, UK, June 1987.

[68] Tomothy Winkler Joseph A. Goguen. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI Internationa, USA, August 1988.

[69] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[70] ISO 8807:1989. LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Standard, International Organization for Standardization(ISO), Geneva, CH, 1989.

[71] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[72] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[73] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[74] Brent Auernheimer and Richard A. Kemmerer. RT-ASLAN: A specification language for real-time systems. *IEEE Trans. Software Eng.*, 12(9):879–889, 1986.

[75] Pamela Zave and Raymond T. Yeh. Executable requirements for embedded systems. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, pages 295–304, 1981.

[76] Pamela Zave. An operational approach to requirements specification for embedded systems. *IEEE Trans. Software Eng.*, 8(3):250–269, 1982.

[77] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[78] David Harel. Statecharts: A visual approach to complex systems. Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.

[79] David Harel, Amir Pnueli, Jeanette P. Schmidt, and Rivi Sherman. On the formal semantics of statecharts (extended abstract). In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 54–64, 1987.

[80] David Harel. Biting the silver bullet - toward a brighter future for system development. *IEEE Computer*, 25(1):8–20, 1992.

[81] The RAISE Language Group. *TheRAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.

[82] Clifford B. Jones. *Systematic software development using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[83] John Michael Spivey. *The Z notation - a reference manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[84] J. P. Bowen. *Formal Specification and Documentation Using Z: A Case Study Approach*. International Thomson Computer Press, 1996.

[85] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal methods in safety-critical railway systems. In *in 'Proceedings of Brazilian Symposium on Formal Methods: SMBF 2007*, pages 26–30, 2007.

[86] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[87] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 131–142, 1983.

[88] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[89] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[90] Rod M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress*, pages 308–312, 1974.

[91] Amir Pnueli. The temporal semantics of concurrent programs. In *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979*, pages 1–20, 1979.

[92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[93] Robert Balzer, Thomas E. Cheatham Jr., and C. Cordell Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, 16(11):39–45, 1983.

[94] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[95] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[96] SPARKAda. http://www.praxis-his.com/sparkada/index.asp.

[97] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[98] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, Sep 1968.

[99] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4): 221–227, April 1971.

[100] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, 1978.

[101] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1st edition, 1998.

[102] Carroll C. Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice Hall, 1994.

[103] Willem P. De Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[104] C. Métayer and Jean-Raymond Abrial. Event-b language. Technical Report Deliverable 3.2, EU Project IST-511599 - RODIN., 2005.

[105] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in event_b. In *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, pages 222–241, 2005.

[106] Ralph-Johan Back and Kaisa Sere. Superposition refinement of reactive systems. *Formal Asp. Comput.*, 8(3):324–346, 1996.

[107] Shmuel Katz. A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(2):337–356, 1993.

[108] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.

[109] Jean Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *STTT*, 12(6):447–466, 2010.

[110] Jean-Raymond Abrial. Event model decomposition. Technical report, ETH Zurich, Private communication, 2009.

[111] Jean Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[112] Thai Son Hoang and Jean-Raymond Abrial. Event-b decomposition for parallel programs. In *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, pages 319–333, 2010.

[113] Michael J. Butler. Decomposition structures for event-b. In *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, pages 20–38, 2009.

[114] Michael Butler. Incremental design of distributed systems with event-b. Chapter: 4, 2009. URL https://eprints.soton.ac.uk/266910/.

[115] Renato Silva and Michael J. Butler. Supporting reuse of event-b developments through generic instantiation. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, pages 466–484, 2009.

[116] Stefan Hallerstede. On the purpose of event-b proof obligations. *Formal Asp. Comput.*, 23(1):133–150, 2011.

[117] Colin F. Snook and Michael J. Butler. UML-B: formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[118] Michael Jastram, Stefan Hallerstede, Michael Leuschel, and Aryldo G. Russo. An approach of requirements tracing in formal refinement. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, pages 97–111, 2010.

[119] Michael Jastram. *The ProR Approach: Traceability of Requirements and System Descriptions*. PhD thesis, Heinrich Heine University Düsseldorf, 2012.

[120] Jonathan Billington. Abstract specification of the iso transport service definition using labelled numerical petri nets. In *Protocol Specification, Testing, and Verification*, 1983.

[121] Geoffrey R. Wheeler, Michael C. Wilbur-Ham, Jonathan Billington, and J. A. Gilmour. Protocol analysis using numerical petri nets. In *Advances in Petri Nets 1985, Covers the 6th European Workshop on Applications and Theory in Petri Nets-selected Papers*, pages 435–452. Springer-Verlag, 1986.

[122] J. Billington, G. R. Wheeler, and M. C. Wilbur-Ham. Protean: a high-level petri net tool for the specification and verification of communication protocols. *IEEE Transactions on Software Engineering*, 14(3):301–316, 1988.

[123] E. Le Mer. Ovide: A software package for verifying and validating petri nets. *IFAC Proceedings Volumes*, 15(7):255 – 260, 1982. 3rd IFAC/IFIP Symposium on Software for Computer Control 1982, Madrid, Spain, 5-8 October.

[124] C H West. A validation of the OSI session layer protocol. *Comput. Netw. ISDN Syst.*, 11(3):173–182, 1986.

[125] C. H. West. General technique for communications protocol validation. *IBM J. Res. Dev.*, 22(4):393–404, July 1978.

[126] Harry Rudin, Colin H. West, and Pitro Zafiropulo. Automated protocol validation: One chain of development. *Computer Networks (1976)*, 2(4):373 – 380, 1978.

[127] Pitro Zafiropulo, Colin H. West, Harry Rudin, D. D. Cowan, and Daniel Brand. *Protocol Analysis and Synthesis using a State Transition Model*, pages 645–669. Springer US, 1982.

[128] Marten J. van Sinderen, I. Ajubi, and F. Caneschi. *The application of LOTOS for the formal description of the ISO session layer*, pages 263–277. ELSEVIER, 1989.

[129] ISO (International Organization for Standardization). Information processing systems – open systems interconnection –lotos description of the session service. Technical Report Technical Recommendation TR 9571:1989, ISO/IEC, Geneva, 1989.

[130] ISO (International Organization for Standardization). Information technology – telecommunications and information exchange between systems – formal description of iso 8072 in lotos. Technical Report Technical Recommendation TR 10023:1992, ISO/IEC, Geneva, 1992. (LOTOS description of the connection-oriented transport service) – Withdrawn on 2004-04-23.

[131] Chris A. Vissers and Giuseppe Scollo. *Formal specification in OSI*, pages 338–359. Springer Berlin Heidelberg, 1987.

[132] Gregor V Bochmann. Protocol specification for OSI. *Computer Networks and ISDN Systems*, 18(3):167 – 184, 1990. Application of Formal Techniques ot the OSI Protocols.

[133] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, March 1987.

[134] Katalin Tarnay, Katalin Tarnay, Gusztv Adamis, and Tibor Dulai. *Advanced Communication Protocol Technologies: Solutions, Methods, and Applications*. IGI Global, Hershey, PA, USA, 1st edition, 2011.

[135] Richard Lai and Ajin Jirachiefpattana. *Communication Protocol Specification and Verification*. 464. Springer US, 1 edition, 1998.

[136] Gerard J. Holzmann. Protocol design: Redefining the state of the art. *IEEE Software*, 9(1):17–22, 1992.

[137] Robin Sharp. *Principles of protocol design*. Springer, 2008.

[138] P. Merlin. Specification and validation of protocols. *IEEE Transactions on Communications*, 27(11):1671–1680, Nov 1979.

[139] G. Bochmann and C. Sunshine. Formal methods in communication protocol design. *IEEE Transactions on Communications*, 28(4):624–631, Apr 1980.

[140] P. Merlin and D. Farber. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, September 1976.

[141] D. Rayner. Towards an objective understanding of conformance. In *Protocol Specification, Testing, and Verification*, pages 493–503, 1983.

[142] D. Rayner. Towards standardized OSI conformance tests. In *Protocol Specification, Testing and Verification V, Proceedings of the IFIP WG6.1 Fifth International Conference on Protocol Specification, Testing and Verification, Toulouse-Moissac, France, June 10-13*, pages 441–460, 1985.

[143] Giessler A Baumgarten, B. *OSI Conformance Testing Methodology and TTCN*. North Holland Elsevier, 1 edition, 1994.

[144] Stephan Tobies Stefan Keil Federico Engler Stephan Schulz Colin Willcock, Thomas Deiß and Anthony Wiles. *An Introduction to TTCN-3*. Wiley, 2 edition, 2011.

[145] Richard Jerry Linn and M Ümit Uyar. *Conformance testing methodologies and architectures for OSI protocols*. IEEE Computer Society Press, 1995.

[146] Andrew S. Tanenbaum. *Computer networks (4. ed.)*. Prentice Hall, 2002.

[147] Eric Madelaine and Didier Vergamini. Specification and Verification of a Sliding Window Protocol in LOTOS. In Ken R. Parker and Gordon A. Rose, editors, *FORTE*, volume C-2 of *IFIP Transactions*, pages 495–510. North-Holland, 1991.

[148] Mark A. Smith and Nils Klarlund. Verification of a Sliding Window Protocol Using IOA and MONA. In Tommaso Bolognesi and Diego Latella, editors, *FORTE*, volume 183 of *IFIP Conference Proceedings*, pages 19–34. Kluwer, 2000.

[149] Dmitri Chkliaev, Jozef Hooman, and Erik P. de Vink. Verification and improvement of the sliding window protocol. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 113–127, 2003.

[150] Patrice Godefroid and David E. Long. Symbolic Protocol Verification with Queue BDDs. *Formal Methods in System Design*, 14(3):257–271, 1999.

[151] A. Udaya Shankar and Simon S. Lam. Construction of network protocols by stepwise refinement. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 669–695. Springer, 1989.

[152] Behrouz A. Forouzan. *Data Communications and Networking*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[153] Jean-Luc Richier, Carlos Rodriguez, Joseph Sifakis, and Jacques Voiron. Verification in XESAR of the sliding window protocol. In *Protocol Specification, Testing and Verification VII, Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification, Zurich, Switzerland, 5-8 May, 1987*, pages 235–248, 1987.

[154] Eric Madelaine and Didier Vergamini. Specification and verification of a sliding window protocol in LOTOS. In *Formal Description Techniques, IV, Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '91, Sydney, Australia, 19-22 November 1991*, pages 495–510, 1991.

[155] Bahareh Badban, Wan Fokkink, Jan Friso Groote, Jun Pang, and Jaco van de Pol. Verification of a sliding window protocol in μcrl and PVS. *Formal Asp. Comput.*, 17(3): 342–388, 2005.

[156] Karsten Stahl, Kai Baukus, Yassine Lakhnech, and Martin Steffen. Divide, abstract, and model-check. In *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings*, pages 57–76, 1999.

[157] Dominique Cansell, Dominique Méry, and Joris Rehm. Time constraint patterns for event B development. In *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, pages 140–154, 2007.

[158] Alistair Kelman. Secure electronic transactions - introduction and technical reference by loeb l. (1998). *Journal of Information, Law and Technology*, 1998(2), 1998.

[159] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[160] Mauricio Papa, Oliver Bremer, John Hale, and Sujeet Shenoi. Formal analysis of e-commerce protocols. In *Fifth International Symposium on Autonomous Decentralized Systems, ISADS 2001, Dallas, Texas, USA, March 26-28, 2001*, pages 19–28, 2001.

[161] Shiyong Lu and Scott A. Smolka. Model checking the secure electronic transaction (set) protocol. In *MASCOTS*, pages 358–364, 1999.

[162] 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.

[163] Max Breitling and Jan Philipps. Transitions into Black Box Views -The NetBill Protocol Revisited-. Technical report, Institut fur Informatik Technische Universitat Munchen, 2000.

[164] Bruce Schneier. *Applied cryptography - protocols, algorithms, and source code in C (2. ed.)*. Wiley, 1996.

[165] Peter Y. A. Ryan and Steve A. Schneider. *Modelling and analysis of security protocols*. Addison-Wesley-Longman, 2001.

[166] Gavin Lowe and A. W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Trans. Software Eng.*, 23(10):659–669, 1997.

[167] Gustavus J. Simmons. Cryptanalysis and protocol failures. *Commun. ACM*, 37(11): 56–65, 1994.

[168] Tony Hoare and Jayadev Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 1–18, 2005.

[169] Tony Hoare. The ideal of program correctness: *Third Computer Journal* lecture. *Comput. J.*, 50(3):254–260, 2007.

[170] Dale Vaillancourt, Rex L. Page, and Matthias Felleisen. ACL2 in drscheme. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2006, Seattle, Washington, USA, August 15-16, 2006*, pages 107–116, 2006.

Résumé

Différentes approches ont été utilisées pour la spécification formelle et la vérification des protocoles de communication. L'approche considérée dans ce travail se concentre sur la vérification déductive dans laquelle la correction d'un modèle par déduction se ramène à la preuve de formules mathématiques appelées obligations de preuve. Dans notre approche, nous avons choisi la méthode Event-B. Event-B est un langage de spécification formel, elle offre un potentiel élevé en termes de correction grâce à son mécanisme de raffinement bien connu, de ses obligations de preuve bien définies et de la plate-forme RODIN. Notre objectif été de construire un modèle avec une formulation claire et précise des propriétés liées aux protocoles de communication et de décharger toutes les obligations de preuve. Pour cela, nous partons d'exigences abstraites, nous les raffinons progressivement pour obtenir une description concrète et détaillée du modèle et nous vérifions chaque niveau de raffinement par rapport à la spécification construite dans le raffinement précédent. Ce qui va nous permettre de vérifier la correction, c'est-à-dire la préservation des propriétés de l'étape précédente, par rapport aux raffinements successifs. Ceci est atteint par la création et la démonstration de logique.

Mots-clefs : Méthodes formelles, Vérification formelle, Event-B, Raffinement, Protocoles de communication

Abstract

Different approaches have been used for the formal specification and verification of communication protocols. The approach considered in this work focuses on the deductive verification in which the correction of a model by deduction is reduced to the proof of mathematical formulas called obligations of proof. In our approach, we chose the Event-B method. Event-B is a formal specification language, offering a high potential for correction through its well-known refinement mechanism, well-defined proof obligations, and the RODIN platform. Our goal has been to build a model with a clear and precise formulation of properties related to communication protocols and to discharge all the proof obligations. For that, we start from abstract requirements, we gradually refine them to obtain a concrete and detailed description of the model and we check each level of refinement with respect to the specification built in the previous refinement. This will allow us to verify the correction, that is to say, the preservation of the properties of the previous step, compared to successive refinements. This is achieved by creating and demonstrating logic.

Key Words : Formal Method, Formal Verification, Event-B, Refinement, Communication protocol