UNIVERSITÉ DE NAMUR

Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

On constructing applications in secure embedded peer-to-peer networks formalization and implementation issues in SMEPP

Gérard, Sébastien

Award date:
2009

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique

Année académique 2008-2009

On Constructing Applications in Secure Embedded Peer-to-Peer Networks : Formalization and Implementation Issues in SMEPP

Gérard Sébastien

Mémoire présenté en vue de l'obtention
du grade de master en informatique.

Résumé

La croissance et la globalisation des réseaux informatiques ainsi que les améliorations apportéees aux moyens de communication ont initié des nouvelles tendances dans la topologie des réseaux, notamment le modèle dit « Peer-to-Peer (P2P) ». Ce modèle est basé sur une hiérarchie non stricte des éléments qui n'ont pas de rôle pré-assignés. Cette évolution est également combinée avec l'évolution des infrastructures de communication. Les appareils impliqués dans ces infrastructures ne peuvent plus compter sur une infrastructure préexistante mais doivent collaborer activement et de façon dynamique afin de créer eux-mêmes une infrastructure leur permettant de communiquer tous ensemble. La combinaison de ces deux évolutions mène au modèle "Embedded Peer-to-Peer Model (EP2P)". Bien que ce modèle peut être préférable à d'autres modèles, celui-ci soulève de nombreuses questions : l'hétérogénité des appareils impliqués dans ce type de réseaux, le manque de fiabilité des moyens de communication ayant pour conséquences la gestion complexe des connexions temporaires et dynamiques entre les éléments constitutifs du réseau ainsi que la prise en charge par le programmeur lui-même de la sécurité. La définition d'un middleware supprimant la complexité induite par ces réseaux est l'approche classique pour faciliter la tâche des programmeurs. Tel est l'enjeu du projet Secure Middleware for Embedded Peer-to-Peer Project (SMEPP).

Le middleware SMEPP offre des fonctions de haut niveaux qui rassemblées au sein de l'API SMEPP, l'API étant le seul moyen offert aux applications d'accéder au middleware SMEPP. De plus, SMEPP fournit un langage de spécification appelé « SMEPP Modeling Language (SMoL) ». Ce langage permet de concevoir aisément des applications EP2P relativement complexes à écrire dans des langages tel que Java. Lors de notre stage à l'Université de Pise (Italie), notre contribution à ce projet fût la création d'un traducteur capable de produire automatiquement une quelconque application écrite en SMoL dans le langage Java.

La première partie de ce mémoire a pour objectif de modéliser dans le langage B, les primitives de l'API SMEPP et leur interactions sur le réseau SMEPP. Dans un second temps, nous documenterons la conception et l'implémentation de notre traducteur. Finallement, nous apporterons quelques critiques à notre modèle ainsi qu'à l'emploi du langage B dans la création de ce type de modèle. Nous terminerons par une analyse critique de notre traducteur, laissant entrevoir de nouvelles perspectives d'améliorations de celui-ci.

Mots clés: EP2P middleware, SMEPP, Méthode B, traducteur SMoL.

Abstract

The growth of the Internet and the improvement of communication technologies have initiated various new trends. New networking models have emerged. Among them, the "Peer-to-Peer model (P2P)", which is no longer based on a strongly hierarchical model, becomes more suitable; it has been specifically addressed in the context of this thesis. This evolution, from a strongly hierarchical model to a model where all the elements of the network are symmetrical, is also combined with the evolution of communication infrastructures. New mechanisms of communication are no longer based on pre-existing infrastructures, but rather on dynamic ad-hoc networks among peers. Combining both evolutions leads to the Embedded Peer-to-Peer Model (EP2P). Nevertheless, EP2P applications raise many issues to be faced: heterogeneity of the involved devices, unreliable communication technologies means to handle highly dynamic connections between network elements and security aspects. Defining a middleware, hiding the complexity induced by these networks, keeps this complexity out of the programmer's mind. This is the support of the Secure Middleware for Embedded Peer-to-peer Project (SMEPP).

The SMEPP middleware can only be accessed through high level functions called "primitives". These primitives are gathered inside the SMEPP API. Moreover, SMEPP provides a specification language called SMEPP Modeling Language (SMoL). This language allows to build complex Embedded Peer-to-Peer programs by describing how they interact with the middleware via its primitives. Our contribution to the project was the development during our internship in the University of Pisa, of a SMoL translator producing automatically Java applications from any kind of program descriptions written in SMoL.

The first part of this thesis intends to model the primitives provided by the SMEPP middleware. The model, written in the B-language, formalizes how B-operations (i.e., formalized API functions) interact with the network via the SMEPP middleware. Secondly, we document the conception and the implementation of the SMoL2Java translator created during our internship. Finally, we propose a feedback on the B-method used in the context of distributed application models, followed by a critical point of view on our SMoL translator opening new work perspectives.

Keywords: EP2P middleware, SMEPP, B-method, SMoL translator.

Acknowledgements

Several people deserve an acknowledgment for contributing with ideas, motivation or any other important aspects.

First of all I want to thank Prof. Jean-Marie Jacquet (University of Namur) for the opportunity he gave to me, for his support, for the time he gave me and for his help in identifying the objectives of this thesis.

Furthermore, I want to thank Prof. Antonio Brogi (University of Pisa) for the opportunity he gave me to work on an European project and for his support during my internship in his University.

Many thanks are also due to Fabrizio Benigni and Sara Corfini (University of Pisa) for all the interesting discussions we had together and their advice during our work. Without them the project would not be as successful as it is. For that, I am eternally grateful.

Finally, I would like to thank the readers of this thesis who have provided valuable advice.

Contents

1 Introduction **1**

2 Context **3**
2.1 State of the Art . . . 3
2.2 Overview of SMEPP . . . 5
2.2.1 The Project . . . 5
2.2.2 Overview of the SMEPP Middleware Issues . . . 6
2.2.3 An Health-Care Application . . . 9
2.2.4 Traffic Management Application . . . 16

3 SMEPP service model **21**
3.1 Primitives of the SMEPP API . . . 21
3.1.1 Peer Management Primitives . . . 21
3.1.2 Group Management Primitives . . . 22
3.1.3 Service Management Primitives . . . 25
3.1.4 Message Management Primitives . . . 27
3.1.5 Event Management Primitives . . . 30
3.2 SMEPP Modelling Language . . . 32
3.2.1 Basic Commands . . . 34
3.2.2 Structured Commands . . . 36
3.2.3 Examples . . . 40
3.3 Formalization Languages . . . 41
3.3.1 Mathematical Theory Based . . . 42
3.3.2 Data Specification . . . 42
3.3.3 Machine Operations . . . 44
3.4 SMEPP Primitives Formalization . . . 45
3.4.1 Data Structures Definitions . . . 45
3.4.2 Direct Access to Structures . . . 63
3.4.3 Data Structures Constraints . . . 66
3.4.4 Peer Management Primitives . . . 81
3.4.5 Group Management Primitives . . . 82
3.4.6 Service Management Primitives . . . 85
3.4.7 Message Management Primitives . . . 87
3.4.8 Event Management Primitives . . . 89
3.5 Conclusion . . . 93

4 SMoL Translator **95**
4.1 Specifications . . . 95
4.2 SMoL Translation . . . 95
4.2.1 Expressions Management . . . 96
4.2.2 Exceptions Handling . . . 98
4.2.3 Structured Command Translation . . . 100
4.2.4 Basic Commands . . . 106
4.3 Translator Architecture . . . 108
4.3.1 Peers Translator . . . 109

4.3.2 Services Translator . . . 110
4.3.3 Events Manager . . . 110
4.3.4 Exceptions Manager . . . 110
4.3.5 Expressions Manager . . . 111
4.3.6 Types Manager . . . 111
4.3.7 Variables Manager . . . 113
4.4 Conclusion . . . 114

5 Perspectives 115
5.1 Service Model . . . 115
5.1.1 Unsupported Constructions . . . 116
5.1.2 SMoL Commands . . . 116
5.1.3 Feed Back on the B-Method . . . 117
5.2 Future Work on the SMoL Translator . . . 119
5.2.1 Structured Commands Implementation . . . 119
5.2.2 XPath Queries . . . 120
5.2.3 Implementation Correctness . . . 120

6 Conclusion 123

A B Summary 133

B B-Operations 139
B.1 Peer Management Primitives . . . 139
B.1.1 newPeer . . . 139
B.1.2 getPeerId . . . 139
B.2 Group Management Primitives . . . 141
B.2.1 createGroup . . . 141
B.2.2 getGroups . . . 141
B.2.3 getGroupDescription . . . 142
B.2.4 joinGroup . . . 142
B.2.5 leaveGroup . . . 143
B.2.6 getIncludingGroups . . . 143
B.2.7 getPublishingGroup . . . 144
B.2.8 getPeers . . . 145
B.3 Service Management Primitives . . . 146
B.3.1 publish . . . 146
B.3.2 unpublish . . . 147
B.3.3 getServices . . . 148
B.3.4 getServiceContract . . . 149
B.3.5 startSession . . . 150
B.4 Message Management Primitives . . . 151
B.4.1 invoke . . . 151
B.4.2 receiveMessage . . . 152
B.4.3 reply . . . 153
B.4.4 receiveResponse . . . 154
B.5 Event Management Primitives . . . 155
B.5.1 event . . . 155
B.5.2 receiveEvent . . . 156
B.5.3 subscribe . . . 157
B.5.4 unsubscribe . . . 162

C SequiTel Examples 167
C.1 Pseudo-SMoL Codes . . . 167
C.1.1 Equipment . . . 167
C.1.2 Operator . . . 169
C.2 SMoL codes . . . 170
C.2.1 Equipment . . . 170
C.2.2 Operator . . . 177
C.3 Translated Codes . . . 182
C.3.1 Equipment . . . 182
C.3.2 Operator . . . 189
C.4 Implemented Codes . . . 193
C.4.1 Equipment . . . 193
C.4.2 Operator . . . 200

D SMoL Codes Translation 205
D.1 Expressions Management . . . 205
D.2 Exceptions Manager . . . 208
D.3 Structured Command . . . 210
D.3.1 SMoL Flow . . . 210
D.3.2 SMoL Pick . . . 212
D.3.3 SMoL Information Handler . . . 218

E Translator Architecture 225
E.1 Peers Translator . . . 229
E.2 Services Translator . . . 230
E.3 Events Manager . . . 231
E.4 Exceptions Manager . . . 231
E.5 Expressions Manager . . . 231
E.6 Types Manager . . . 233
E.7 Variables Manager . . . 238

List of Figures

2.1 Networking Taxonomy . . . 4
2.2 Peer Example . . . 7
2.3 Contract.xsd Structure . . . 7
2.4 State-less, Session-Less and Session-Full Services . . . 9
2.5 SequiTel Example . . . 11
2.6 SequiTel Peer Services . . . 12
2.7 Traffic Management Application . . . 17
2.8 Traffic Application Diagram . . . 17

3.1 B-machine Example . . . 43
3.2 Identifiers Sets Hierarchy . . . 46
3.3 Identifiers Sets Hierarchy, Mathematical Representation . . . 47
3.4 Constraint CS01 . . . 66
3.5 Constraint CS02 . . . 66
3.6 Constraint CS03 . . . 67
3.7 Constraint CS04 . . . 67
3.8 Constraint CS05 . . . 67
3.9 Constraint CS06 . . . 67
3.10 Constraint CS07 . . . 68
3.11 Constraint CP01 . . . 68
3.12 Constraint CG01 . . . 68
3.13 Constraint CG02 . . . 68
3.14 Constraint CG03 . . . 68
3.15 Constraint CSUB01 . . . 69
3.16 Constraint CSUB01 - Part 1 . . . 70
3.17 Constraint CSUB01 - Part 2 . . . 70
3.18 Constraint CSUB01 - Part 3 . . . 70
3.19 Constraint CSUB01 - Part 4 . . . 71
3.20 Constraint CSUB01 - Part 5 . . . 71
3.21 Constraint CSUB01 - Part 6 . . . 71
3.22 Constraint CSUB01 - Part 7 . . . 72
3.23 Constraint CSUB01 - Part 8 . . . 72
3.24 Constraint CSUB02 . . . 73
3.25 Constraint CSUB03 . . . 73
3.26 Constraint CSUB04 . . . 74
3.27 Constraint CUSUB01 . . . 75
3.28 Constraint CUSUB02 . . . 76
3.29 Constraint CUSUB03 . . . 76
3.30 Constraint CESD01 . . . 77
3.31 Constraint CI01 . . . 78
3.32 Constraint CI01 - Part 1 . . . 78
3.33 Constraint CI01 - Part 2 . . . 79
3.34 Constraint CI01 - Part 3 . . . 79
3.35 Constraint CR01 . . . 80

3.36 Constraint Part Shared by CR01 and CR02 . . . 80
3.37 Constraint CR02 . . . 81
3.38 Constraint CN01 . . . 81
3.39 Constraint CN02 . . . 81
3.40 Constraint CV01 . . . 81
3.41 A Sample Pseudo-SMoL Code . . . 94
3.42 A Sample CSP Code . . . 94

4.1 waitFor Pseudo-Implementation . . . 107
4.2 waitUntil Pseudo-Implementation . . . 107
4.3 How Translating SequiTel . . . 114

5.1 getPeers Implementation With Exceptions Support . . . 116
5.2 Implementation of A FaultHandler in CSP . . . 117

B.1 newPeer . . . 139
B.2 getPeerId . . . 139
B.3 getPeerId with Arguments . . . 140
B.4 createGroup . . . 141
B.5 getGroups . . . 141
B.6 getGroupDescription . . . 142
B.7 joinGroup . . . 142
B.8 leaveGroup . . . 143
B.9 getIncludingGroups . . . 143
B.10 getPublishingGroup . . . 144
B.11 getPeers . . . 145
B.12 publish . . . 146
B.13 unpublish . . . 147
B.14 getServices . . . 148
B.15 getServiceContract . . . 149
B.16 startSession . . . 150
B.17 invoke . . . 151
B.18 receiveMessage . . . 152
B.19 reply . . . 153
B.20 receiveResponse . . . 154
B.21 event Primitive for Peers (no groupId) . . . 155
B.22 receiveEvent . . . 156
B.23 subscribe - First Type . . . 157
B.24 subscribe - Second Type . . . 159
B.25 subscribe - Third Type . . . 160
B.26 subscribe - Fourth Type . . . 161
B.27 unsubscribe - Fourth Type . . . 162
B.28 unsubscribe - Third Type . . . 164
B.29 unsubscribe - Second Type . . . 165
B.30 unsubscribe . . . 166

C.1 Equipment Peer - Pseudo-SMoL . . . 167
C.2 Equipment Monitoring - Pseudo-SMoL . . . 168
C.3 Operator Peer - Pseudo-SMoL . . . 169
C.4 Equipment Peer - SMoL - Part 1 . . . 170
C.5 Equipment Peer - SMoL - Part 2 . . . 171
C.6 Equipment Peer - SMoL - Part 3 . . . 172
C.7 Equipment Monitoring - SMoL - Part 1 . . . 173
C.8 Equipment Monitoring - SMoL - Part 2 . . . 174
C.9 Equipment Monitoring - SMoL - Part 3 . . . 175
C.10 Equipment Monitoring - SMoL - Part 4 . . . 176
C.11 Operator Peer - SMoL - Part 1 . . . 177

C.12 Operator Peer - SMoL - Part 2 178
C.13 Operator Peer - SMoL - Part 3 179
C.14 Operator Peer - SMoL - Part 4 180
C.15 Operator Peer - SMoL - Part 5 181
C.16 Equipment Peer- Translated - Part 1 182
C.17 Equipment Peer - Translated - Part 2 183
C.18 Equipment Peer - Translated - Part 3 184
C.19 Equipment Monitoring - Translated - Part 1 185
C.20 Equipment Monitoring - Translated - Part 2 186
C.21 Equipment Monitoring - Translated - Part 3 187
C.22 Equipment Monitoring - Translated - Part 4 188
C.23 Operator Peer - Translated - Part 1 189
C.24 Operator Peer - Translated - Part 2 190
C.25 Operator Peer - Translated - Part 3 191
C.26 Operator Peer - Translated - Part 4 192
C.27 Equipment Peer - Implemented - Part 1 193
C.28 Equipment Peer - Implemented - Part 2 194
C.29 Equipment Peer - Implemented - Part 3 195
C.30 Equipment Monitoring - Implemented - Part 1 196
C.31 Equipment Monitoring - Implemented - Part 2 197
C.32 Equipment Monitoring - Implemented - Part 3 198
C.33 Equipment Monitoring - Implemented - Part 4 199
C.34 Operator Peer - Implemented - Part 1 200
C.35 Operator Peer - Implemented - Part 2 201
C.36 Operator Peer - Implemented - Part 3 202
C.37 Operator Peer - Implemented - Part 4 203

D.1 Variables Declaration 205
D.2 Types Declaration 206
D.3 SMoL Left Expression 206
D.4 SMoL Right Expression 207
D.5 Java try-catch 208
D.6 Java Exception Hiearchy 208
D.7 Exception Usage Example 209
D.8 Translation of Exception Commands 209
D.9 Translated SMoL Exception 209
D.10 SMoL Flow - Class Diagram 210
D.11 Translation of A SMoL Flow 211
D.12 SMoL Flow - Implementation 211
D.13 SMoL Pick - Class Diagram 212
D.14 Translation of A SMoL Pick - Part 1 213
D.15 Translation of A SMoL Pick - Part 2 214
D.16 SMoL Pick - Behaviour 215
D.17 SMoL Pick - Events Round Robin 216
D.18 SMoL Pick - Receive Messages, Response and Alarms Round Robin 217
D.19 SMoL Information Handler - Class Diagram 219
D.20 Translation of a SMoL Information Handler - Part 1 220
D.21 Translation of a SMoL Information Handler - Part 2 221
D.22 SMoL Information Handler - Behaviour 222
D.23 SMoL Information Handler - ReceiveMessage, ReceiveEvent and alarms round robin . . . 223
D.24 SMoL Information Handler - ReceiveEvent Round Robin 224

E.1 Common Translator Classes - Part 1 226
E.2 Common Translator Classes - Part 2 227
E.3 Common Translator Classes - Part 3 228
E.4 Peers Translator 229

E.5 Services Translator . . . 230
E.6 Events Manager . . . 231
E.7 Exceptions Manager . . . 231
E.8 Expressions Manager - Part 1 . . . 232
E.9 Expressions Manager - Part 2 . . . 233
E.10 Types Manager . . . 233
E.11 Type Inferring - Variable used in A Right Expression . . . 234
E.12 Type Inferring - Variable used in A Right Expression - Case 1 . . . 235
E.13 Type Inferring - Variable used in A Right Expression - Case 2 . . . 235
E.14 Type Inferring - Variable used in A Right Expression - Case 3 . . . 235
E.15 Type Inferring - Variable used in A Right Expression - Case 4 . . . 235
E.16 Type Inferring - Variable Used in A Left Expression . . . 236
E.17 Type Inferring - Variable Used in A Left Expression - Case 1 . . . 237
E.18 Type Inferring - Variable Used in A Left Expression - Case 2 . . . 237
E.19 Type Inferring - Variable Used in A Left Expression - Case 3 . . . 237
E.20 Type Inferring - Variable Used in A Left Expression - Case 4 . . . 237
E.21 Variables Manager . . . 238

List of Tables

3.1 Summarize of Existing Exceptions . . . 33
3.2 Sets Operators . . . 41
3.3 Relation Operators . . . 41
3.4 Predicate Operators . . . 42
3.5 Substitution Operators . . . 42

B.1 Subscriptions State Evolution . . . 158
B.2 Unsubscriptions Evolution . . . 163

Chapter 1

Introduction

The growth of the Internet and the improvement of communication technologies have initiated various new trends. Until recently, network infrastructures are based on the assumption that users are simple clients making small requests and receiving, possibly large, replies. However, some applications require more than the simpler client/server architecture. As an answer, new networking models like the peer-to-peer model which is not a strongly hierarchical model, have been proposed and are indeed more suitable for these applications. But, this model requires new protocols and new programming tools.

The peer-to-peer model offers many advantages (see [Doy00, p. 143-144]):

- **Capacity.** All kinds of resources can be shared: computing power, storage capacity, access to physical resources, etc. Once a node is connected to the network, demand on the system increases, but the total capacity of the system also increases.
- **Robustness.** In the client-server model, all requests are executed by one or few centralized servers. This single point of failure in the system is no more present in the P2P model.
- **Low cost.** Only ordinary computers are needed; there is no need for a server or sophisticated network operating system. Moreover, application deployment becomes an easy task and consequently P2P networks are less expensive to set up.

This evolution, from a strongly hierarchical model to a model where all the elements of the network are symmetrical, is also combined with the evolution of communication infrastructures. New mechanisms of communication are no more based on pre-existing infrastructures, but rather on dynamic ad-hoc networks among peers. This evolution has been reinforced by the growth of new generation mobile phones, PDA, electronic sensors which feature more and more wireless communication. These new devices have induced new communication needs resulting in new areas of applications where mobile devices can collaborate using wireless channels [Coa07b, p. 6].

In an ad-hoc network, terminals communicate on non-predefined infrastructures. Thus, ad-hoc networks are able to easily change their topologies. If some peers disconnect or new peers appear, the collaboration can continue. They act **autonomously**. Nodes have routing and information processing capabilities. This makes the network to be **fault tolerant** [Coa07b, p. 12] and **self-healing**. Nodes can re-configure themselves to keep routing information updated. So, their environments can change without disturbing communication.

Despite these important features, the peer-to-peer model has not yet expanded beyond a few niche applications. The main drawbacks are also due to those characteristics (see [NBZ04] and [Loo07, p. 27]):

- **Heterogeneity.** Ad-hoc networks are often composed of heterogeneous software and hardware. These devices communicate on different networks, so programmers have to find a generic way of communication over these networks.

- **Reliability.** Each node is free to leave the network whenever it wants or whenever it moves out of range. Communication protocols have to manage efficiently topology changes. Nodes have to manage autonomously, at run-time, their interactions and interconnections with its environment.
- **Security.** Ad-hoc networks use wireless communication which can lead to security attacks. Communication encryption can avoid this problem.
- **Waste of performance.** Operations performed on a single computer can be completed within a very short time. However, on a distributed system, performing these operations might require many communication messages and so take a much longer time. This problem is accentuated by the weak computational power of ad-hoc network nodes.

All these concerns are typically in charge of the application developers. But, thanks to a high-level programming language, these aspects can be handled by the underlying language, keeping them out of the programmer's mind.

During our internship, we had the opportunity to take part of the Secure Middleware for Embedded Peer-to-peer Systems (SMEPP) project. "*Its goal is to develop a middleware that will have to be secure, generic and highly customisable allowing for its adaptation to different devices (PDA, smart phones, embedded sensor actuator systems) and different domains (critical systems, consumer entertainment or communication).*"[BP08a]. The SMEPP middleware can be used by programmers through the usage of the SMEPP API which provides a set of abstract Java primitives interacting with the middleware. These primitives are the only way to interact with it.

In the context of the SMEPP project, a language called SMEPP Modeling Language (SMoL) has been imagined. SMoL allows to build complex Embedded Peer-to-Peer programs describing how the application behaviour interacts with the middleware via its primitives. This language notably simplifies the time-consuming and error-prone task of specifying the interactions of a complex peer-to-peer system. The purpose of our internship was the development of a SMoL translator automatically producing (without any human manipulation) Java application from any kind of program descriptions written in SMoL. More details about SMoL in Chapter 2.

The contribution of this thesis can be divided in two parts. First, this thesis contains a B-model (written in the B-language) of the middleware developed by this project. The goal of this model is to clearly define functions provided by this middleware. Thanks to this model, it is possible to have an another point of view that the current documentation. It's also possible to define its own network state and make simulation on it. The second part of this thesis concerns the design and the implementation of the SMoL translator developed during our internship in Pisa. This translator is able to translate a code written in the abstract language SMoL to a Java application. Of course, this translator is independent from the source code and can be reused in any applications written in SMoL.

The rest of this master thesis is structured as follows: Chapter 2 underlines the opportunity of a new peer-to-peer middleware like SMEPP which allows to easily create EP2P applications interacting together. In a second time, a short presentation of the SMEPP European project will be followed by a description of the the middleware developed by this project. The two following chapters constitute the main contribution of this master thesis. Chapter 3 is focused on the SMEPP API primitives. Each of them will be introduced and formalized. Thanks to our model, we will see how primitives interact with the SMEPP network. Chapter 4 describes the design and the implementation of the SMoL translator developed during our internship in Pisa. Finally, Chapter 5 looks critically at the SMEPP model created in Chapter 3 and also at the SMoL translator. We conclude by opening new work perspectives.

Chapter 2

Context

This chapter, on context, defines the SMEPP project, the support of our thesis. As said in the introduction, there is nowadays a growing need of peer-to-peer applications. Firstly, this chapter underlines the opportunity of a new embedded peer-to-peer middleware like SMEPP.

In a second time, a short presentation of the SMEPP European project will be followed by an introduction of the key concepts of middleware developed by this project. We will see how SMEPP fulfils the requirements described in the beginning of this chapter.

2.1 State of the Art

This section is subdivided into two parts. The first part is based on an analysis [Coa07b] made by the SMEPP project which tries to design the most abstract architecture needed to design Peer-to-Peer networks and especially for embedded Peer-to-Peer networks. The second part mentions what is currently not satisfied and requires further development.

Several years before, due to a poor computer power, bandwidth and storage space, the main software architecture was based on a *client-server model*. In this model *a client software* initiates a communication over a computer network while the *server software* waits for client requests and replies to them. Nowadays, this lack of resource is less and less present. New kinds of networking models can take advantage of this evolution. **Self-organizing networks** also called **distributed transient networks**, are decentralized networks. Network elements can join or leave the network as they want and where they want. This paradigm is composed of two well known networks: **Peer-to-Peer network** (P2P) and **ad-hoc network**.

In a *P2P network*, the notion of *client* and *server* has disappeared ; each application interacting inside this network is called a *peer*. Each peer node acts simultaneously as a *client* and a *server* on the network. The application intelligence and storage is spread into the entire network. We define a Peer-to-Peer network as a network where network elements communicate in a bidirectional and symmetric way with each other. This definition doesn't imply any condition on the underlying network, it can be a virtual network set up on the existing overlay network, or the network can have a fixed infrastructure.

At the same time, the recent technological advances in short distance wireless communications and embedded systems have opened up new areas of application where small, low-powered, low-cost systems collaborate in the processing and management of information using wireless channels.

"*An ad-hoc network consists of a collection of wireless mobile nodes dynamically forming a temporary network without the use of any existing network infrastructure or centralized administration*"[Wu05]. Most of the current ad-hoc networks can be classified into two main types: Mobile Ad-hoc Network (MANET) and Wireless Sensor Network (WSN).

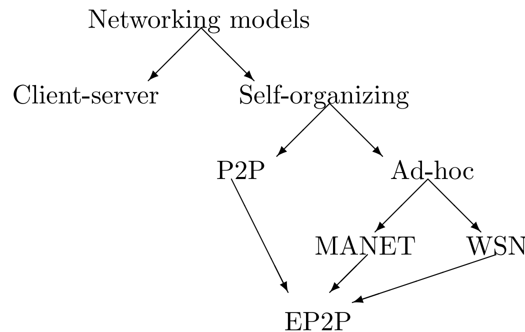

Figure 2.1: Networking Taxonomy

Mobile Ad-hoc Network (MANET) are self configuring networks where each of the network nodes act as a mobile router. The union of these nodes forms and arbitrary changing topology (non existing infrastructure) in which the nodes/routers are free to move randomly and organize themselves arbitrarily.

The second type, **Wireless Sensor Network (WSN)** is composed of hundreds or thousands *sensors* which are tiny, low-power and deployed in the environment in which they interact (e.g. environmental monitoring, military surveillance, etc.). A WSN may exhibit a dynamic topology due to communication or node failures and, sometimes, mobility. But, this mobility is *controlled*. On the contrary, MANETs assume no control over mobility. These non-safely components permit to get a lot of information from their environments, but applications focus on what data is desired rather than on individual sensor nodes. So, WSN applications eliminate redundancy and minimize the number of transmissions in order to save energy and resources.

These networks can be combined: a Peer-to-Peer network over an ad-hoc network. Elements of this hybrid network, called **Embedded Peer-to-Peer network**, interact together through temporary network connections. But, this new network requires additional techniques (routing, service discovery, etc.) on the nodes at the middleware level in order to achieve efficient communications.

The SMEPP team has analyzed (e.g., [S. 05], [MBJ05], [GP05], [RHR06], [Coa07a] or [LZ04]) current solutions for EP2P networks and extracted requirements necessary to create an abstract and flexible middleware. They notice that there is currently no middleware able to create this hybrid network which implements the following concepts:

- **Main Objective.** Developing a middleware which can be used in many domains. The middleware must be easily customizable and provide high-level functions at the level of the Application Programming Interface (API).
- **A Decentralized Network.** SMEPP must create a decentralized network. The basic entity of this network is the *peer*. A peer is a program interacting with other peers. In a decentralized network, peers can join and leave the network as they want without any constraint.
- **Group Requirement.** A group is a logical (i.e. from the point of the application view) set of peers. A peer can join several groups and create a group by itself, establishing the group security requirements (only authenticated peers can join the group) and its description. The description can be used by peers to search for a group associated to a given description.
- **Application Based on Services.** Peers offer services. Services are composed of a set of operations which can be called like a local operation. These operations can be either procedures (without any returned value), or functions (with returned values). They can be called synchronously or asynchronously by a peer or a service. A peer and its services must belong to a group in order to use services offered by the respective group. What's more, operations can be explicitly invoked or implicitly invoked by providing the description of the operation that we want to call. In that case,

the operation which better fulfils the given description is called. Thus, when a service is published the service must be associated to a contract which describes the service and its operations.

- **Events.** Another way of communicating among peer is on events based. An event is a message which can be spread through the entire group in which it's raised. Receiving an event implies a previous subscription to this event and entities must explicitly be listening to that event.

- **Architecture Constraints.** The middleware can be executed on embedded systems. Thus it has to be able to provide latency guarantees to get real-time applications on non powerful devices. Applications can specify their own requirements in terms of QoS parameters when they call a service operation. In that way the middleware has to provide the operation which satisfies these requirements.

- **Scalability.** The middleware has to be efficient in light applications (with few peers) and also in very large systems (with many peers spread in different areas).

- **Abstract Application and Proof.** EP2P application behavior can be described abstractly. This abstraction permits application simulations and behavior checkings.

- **Heterogeneity.** SMEPP applications will be spanning over a very heterogeneous terminal (PCs, laptops, mobile phones, PDAs, etc.), gateways, sensors and device ecosystems.

- **Networking Adaptability.** SMEPP will support infrastructure and infrastructure less networks (ad-hoc networks). In all the cases, the networking and routing protocols used by the middleware will have to support important changes of the network topology (network elements come into the system and go out in an independent way, involving frequent reorganization of the system). But, the middleware should hide the complexity of the underlying infrastructure.

The purpose of SMEPP is to create a middleware which satisfies these above requirements.

2.2 Overview of SMEPP

2.2.1 The Project

The purpose of the Secure Middleware for Embedded Peer-to-peer Systems (SMEPP) project is the development of a middleware which satisfies the requirements described in the previous section.

SMEPP is a project funded by the Sixth Framework programme (FP6) financed by the European Union. "*The main objective of FP6 was to contribute to the creation of the European Research Area by improving integration and co-ordination of research in Europe, which is so far largely fragmented*" [FP6].

Actors involved in the SMEPP project are:

- *Telefónica's International Corporate*: "*Telefónica is one of the world leaders integrated operator in the telecommunication sector, providing communication, information and entertainment solutions, with presence in Europe, Africa and Latin America*" [TEL31]. This corporation is in charge of the SMEPP middleware validation and its requirements.

- *Tecnatom*: "*The Mission of Tecnatom is to contribute through technology development and the application of its services to improving the safety, availability and economic efficiency of power and industrial facilities on the national and international markets*" [TEC31]. *Tecnatom* is going to develop industrial applications using the SMEPP middleware.

- *Universidad de Málaga* is composed of three Faculties: Political, Economic and Business Sciences. One of the main objectives is to achieve the "*strengthening of research and technological development*" [MAL31]. This University works on technology components for distributed and

real-time embedded systems. They also have a wide experience in security issues.

- *Technische Universitaet Graz*: "*The University research fields are the engineering sciences and the technical-natural sciences. These researches are carried by the strength of its knowledge-oriented and applied research*" [GRA31]. They handle all aspects related to security, specially those related to specific hardware support for security.

- *Siemens - IT Consulting and Systems Integeration*: "*One of the business solutions offerred by Siemens is the business processes security, and consequently of the company itself. This security depends on the availability of the information technology and on the security of the "information" assets of a company*" [SIE31]. Their contribution to the SMEPP project is the development of cryptographic algorithms.

- *Valtion Teknillinen Tutkimuskeskus*: "*VTT Technical Research Centre of Finland the biggest multitechnological applied research organisation in Northern Europe. VTT provides high-end technology solutions and innovation services*" [VTT31]. They helped to design the SMEPP system architecture.

- *Università di Pisa (Italy)*: "*The computing department have different research groups and laboratories such as : algorithms and data Structures, parallel and distributed architectures, Wireless and Mobile, ...*" [UPI31]. They are in charge of the SMEPP API. They also work on formal methods and abstract models of computing.

- *Institute for Infocomm Research*: "*Our research capabilities are in information technology, wireless and optical communication networks, interactive and digital media; signal processing and computing*" [IIR31]. Thanks with their experiences they can help to build a secured SMEPP network by creating models and their validations.

2.2.2 Overview of the SMEPP Middleware Issues

The networking model proposed by the SMEPP project is based on a combination between Peer-to-Peer networks and Ad-hoc networks (WSN & MANET) in order to have an Embedded Peer-to-Peer (EP2P) network. This model is thus extremely flexible. As stated before, SMEPP must be abstract thus these networking concerns are out of interest of applications which use the middleware. Whatever the underlying network, applications use the middleware in the same way. The deployment of a SMEPP network on an ad-hoc network, or on an Ethernet network is in charge of the middleware.

The SMEPP middleware can be used by programmers via the **SMEPP API** which provides a set of abstract primitives interacting with the middleware. These primitives are the only way to interact with it. Before introducing each of them in the next chapter, a global introduction of the SMEPP service model is necessary.

A SMEPP network can be conceived as a set of elements called peers. Each SMEPP network is associated to **credentials**. These credentials are used by peers to identify a given SMEPP network and also to grant them to join it. Without the right credentials, they cannot be member of the network. The network credentials are set by the peer creator of the network. The **peer** notion is the basic entity of this model. A peer (see Figure 2.2) is a computing unit which provides **services** callable by other peers. Once created, it receives an unique, stable identifier which can be used by other peers to identify this peer.

In order to distinguish some peers from other peers, the notion of **group** has been imagined. A peer can create a group by itself, when it wants, establishing the security requirements for the created group (a group cannot be a member of another group). A group can accept or reject a new peer in the group if

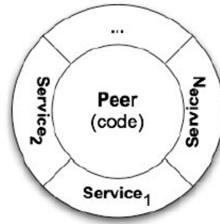

Figure 2.2: Peer Example

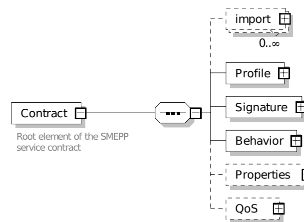

Figure 2.3: Contract.xsd Structure

it doesn't provide the right credentials. Moreover, a peer can search for existing groups, join them if it's authorized to do so and can leave them later when it wants (note that a group must contains at least one peer). A group is thus a logical collection of peers and can be conceived as a secured environment where peers can interact safely together. For further details on groups, please see section 3.1.2.

A SMEPP service consists of three parts:

- A **contract** is specified using the eXtensible Markup Language (XML). "*A contract must contain all the information that any client (human or software) of the service may need to discover, to instantiate and to interact with the service*" [Coa08a]. Figure 2.3 illustrates the structure of this XML file (diagram generated by Altova XMLSpy [XML07]).

  A contract has five parts:

  - A **profile** is a short semantic description of the service: a service name and its category (e.g. a service called "takeTemperature" which is a "Health care service").
  - A **signature** provides an abstract description of operations offered by the service to other peers, a description of events raised by the service and describes "What the service does ?". The signature part based on based on a simplification of WSDL 2.0 [WSD15], is a mandatory part of the contract.
  - A **behavior** is optional and written in SMoL (SMoL will be defined in the next chapter). It serves to better match service descriptions, to analyse the service and simulate it. This part replies at "How the service does it ?".
  - **Properties** is a set of tuples composed of a name, a type, a value and a category (optional links to ontologies).
  - **QoS** describes aspects related to the "Quality of Service" offered by the service (based on Amigo's QoS parameter ontology [Pro15]).

- An **implementation** is the executable service code (e.g. a Java application).

- A **grounding** describes how clients can invoke the service, i.e. it provides information on the way clients have to communicate with it (e.g. port numbers, protocols, . . .).

When a peer wants to publish a service, the peer has to provide the service grounding and contract. *A service offered by a peer is an instance of a service contract which defines the service type.* Each

instance can be identified by an element belonging to `PeerServiceId`. Each service published in the same group with the same service contract receives the same service identifier called `GroupServiceId`. This identifier allows to blindly call a service, if an entity wants to interact with a service by specifying a `GroupServiceId`, the middleware is in charge of searching a specific, connected and currently reachable peer offering this service. This abstraction allows to hide the fact that there are several providers of the same service in the same group.

As said before, a SMEPP service is composed of a set of **operations**. Operation invocations can be conceived as an exchange of messages which can be achieved by other peers or services to retrieve information or to execute a task. SMEPP proposes two kinds of operations:

- **One-Way.** When an application calls a one-way operation, only optional input messages can be specified. A one-way operation starts processing as soon as the service provider is ready to accept the invocation and receives the input message. After that the invoker doesn't stay blocked.
- **Request-Response.** If an application calls this operation, it can provide optional input messages and receive an output message. SMEPP provides two types of request-response operations:
  - **Synchronous Operations**[1]. An application calling a synchronous operation, stays blocked until the provider is not ready to accept invocations and until the called operation has not provided an answer to this call. Once finished, the operation can return an output message and the caller can continue its execution.
  - **Deferred Synchronous Operations**[1]. An application calling a deferred synchronous operation, stays also blocked until the provider is not ready to accept invocations, but it doesn't stay blocked until the operation execution. When the application want to retrieve the operation result, it calls a primitive which waits the operation ending and returns the operation result.

SMEPP defines different service categories based on the notion of **sessions**. A session can be conceived as *a virtual communication channel* between clients and a running service instance. SMEPP intends to be so flexible as possible and sub-divides services into two categories:

- **State-Less.** These services don't keep track of their interactions with clients. Clients can invoke operations of *state-less* services one or more times and in any order. Furthermore, a *session* is created at each invocation and remains until the operation ending. State-less services can be called either thanks to a `GroupServiceId`, or thanks to a `PeerServiceId`.
- **State-Full.** These services keep track of their interactions with clients. State-full services can be divided into two other categories according to the notion of session.

  These new subcategories are:

  - **State-Full Session-Less services.** These services have only one communication channel by service instance shared by all service clients. For example, if a peer publishes the file sharing service in the first and second group, there are two service instances and thus two sessions. This channel remains active from the service publishing to the service unpublishing. State-full session-less services can only be called thanks to their `PeerServiceId`. Indeed, it's necessary to exactly know the service provider (so a `GroupServiceId` cannot be used) since this kind of service keeps track of interactions with clients.
  - **State-Full Session-Full services.** These services support multiple communication channels. Clients can open new sessions and share them with other clients. In this case, each session receives a unique identifier. Invokers of such services must provide a `SessionId` ; `GroupServiceId` and `PeerServiceId` are prohibited. Once published, session-full services have no associated sessions, so there is no running service instance and clients are in charge of creating sessions (see section 3.1.3) before interacting with the service.

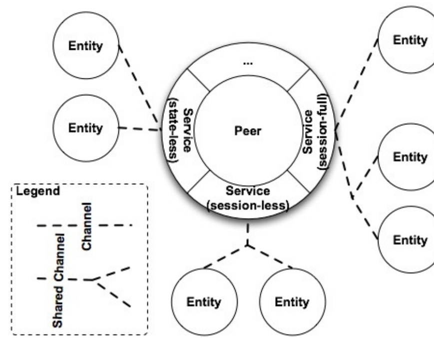

Figure 2.4: State-less, Session-Less and Session-Full Services

On the left-hand side of the Figure 2.4 extracted from [Coa08a], a state-less service is interacting with two clients. The bottom part presents a state-full session-less service where two clients share one communication channel. Finally, the right-hand side of the picture represents a session-full service communicating with three clients over two channels; one client has its own channel while the other two clients share the last channel.

Faults also called **exceptions** are the mechanism to communicate failures. Faults can be thrown by the middleware or by applications. If a failure occurs during the execution of a SMEPP API primitive or if the application code calls this primitives in a wrong way then the primitive throws a fault. Faults can also be thrown by entities as an operation result (only in the case of a request-response operation).

The last communication mechanism is on **event** based which is an *one-to-many* and non-blocking communication way. An event has a *name* and an input (the payload). Entities wanting to receive events of a given name, have to subscribe to this event name and then *listen* to this event. Listening is a blocking operation, entities continue processing as soon as they receive one event having that name.

We can redefine the **group** as "*a set of peers that share or provide services to other services or peers. They are the basic abstraction for identification, security, communication broadcasting and service provisioning*".

2.2.3 An Health-Care Application

The two following Subsections introduce typical distributed applications illustrating the contribution of SMEPP. We will see the usefulness of such kind of middlewares in terms of functions, development processes and applications architectures.

The first example extracted from the thesis of Jean-Louis Buchholz and Julien Lange [BL08], is dedicated to an existing application called SequiTel which SMEPP is intended to be used. We will see that using the SMEPP middleware in order to implement this application permits to develop easier this application.

SequiTel intends to gather information and medical technologies to better meet needs of patients and health-care professionals. Its architecture is a traditional client-server where users can subscribe to different services depending on their needs (elders, pregnant women, diabetics, etc.). They can also make video-conferences (tele-consulting), handle their own agenda, raise alarms, follow their human vital parameters (temperature, electrocardiogram, blood pressure, . . .).

The main drawback of this system is related to its architecture. If servers are not available then users cannot run most of the services. It's for this reason that a P2P architecture can resolve this disadvantage.

[1]Note that this notion is not mentioned in the SMEPP documentation

A patient remains connected with health-care professionals, his family, or neighbors in spite of a server disconnection. Moreover, each user become a service provider and can share services. Another feature concerns the creation of personal groups where users share experiences, help other people of the group, gather people undergoing the same problems or the same situation (pregnant women, renal patient, etc.). Patients are equipped with medical sensors able to send measures or alarms transmitted through the network. Health care applications require to be secured: confidentiality, integrity and authenticity are needed in the data transmission.

As we will see using SMEPP for the development of SequiTel can provide a solution for its conception and its implementation. We can imagine these features:

Browser. The future SequiTel browser provides information related to the user's network environment. This browser can be run in two modes. In the first mode, the user can navigate through the network. The user interface shows connected peers and services they are offering. The second mode provides navigation by showing groups, their members and services they offered.

Alarms Notifications. Patient can raise alarms spread to his family, friends or neighbors. Raising alarms through to groups is a very simple task thanks to SMEPP because it is a built-in functionality.

Emergency Alarms from a Mobile Device. New mobile phones embeds emergency features. In this way, users can send a S.O.S message wherever they are (they keep the phone on them) and this message can possibly automatically include users's GPS coordinates. Applications developed with SMEPP can be easily deployed on a large variety of devices and take advantage of devices features.

File Sharing. In the current system, users can only download videos from the central server. With SMEPP, a peer is able to download a file (videos, documentation, etc.) from several sources (as in systems like BitTorrent) using the bandwidth more efficiently and providing a wider range of information.

Tele-consulting. The current SequiTel system allows tele-consulting which minimizes business travel for non-critical situations. The new system based on SMEPP can improve the quality required by tele-cared users and doctors. We can integrate embedded systems like mobile phones, laptops, autonomous webcams, etc.

Agenda. Each user can create his own agenda which contains reminders or appointments. This planning can be shared with other people: family, doctors, etc. We can imagine a reminder sent by a doctor to his patient. If the patient is not connected, it can be sent to the family, or friends.

Workflow. Health-care applications require measures like blood pressure, temperature, heartbeat, etc. which can be automatically done thanks to electronic sensors. Indeed, SMEPP is able to provide a powerful and easy mechanism for embedded sensors integration. Then, all of these periodic measures can be sent to tele-cared operators which make a periodic report to doctors.

Chat. SequiTel must provide a basic mechanism allowing any user to communicate with people in the same situation. Moreover, it could decrease the number of queries to specialists by avoiding redundant questions. Other similar applications such as a forum could also be studied. The current system is a fully-centralized system in opposition to SMEPP which can create chats directly between users. A fully-decentralized system minimizes request to SequiTel and avoid the dependence on their systems.

Figure 2.5 illustrates a basic scenario. The tele-cared user John Smith is sitting at home. He is equipped with an embedded device which can raise an emergency alarm or interact with other people (video conferences, chat, forum, etc.). This device is connected to his personal network connected to Internet. An autonomous camera is also connected on him network. This camera allows John to have video conferences without any computer. Assistant users (friends, doctors, etc.) are on the web through their mobile operators. The first assistant uses his PDA on a cellphone network. The second uses his laptop on a wireless network like Wi-Fi or Wi-max. SequiTel operators are displayed on top of the

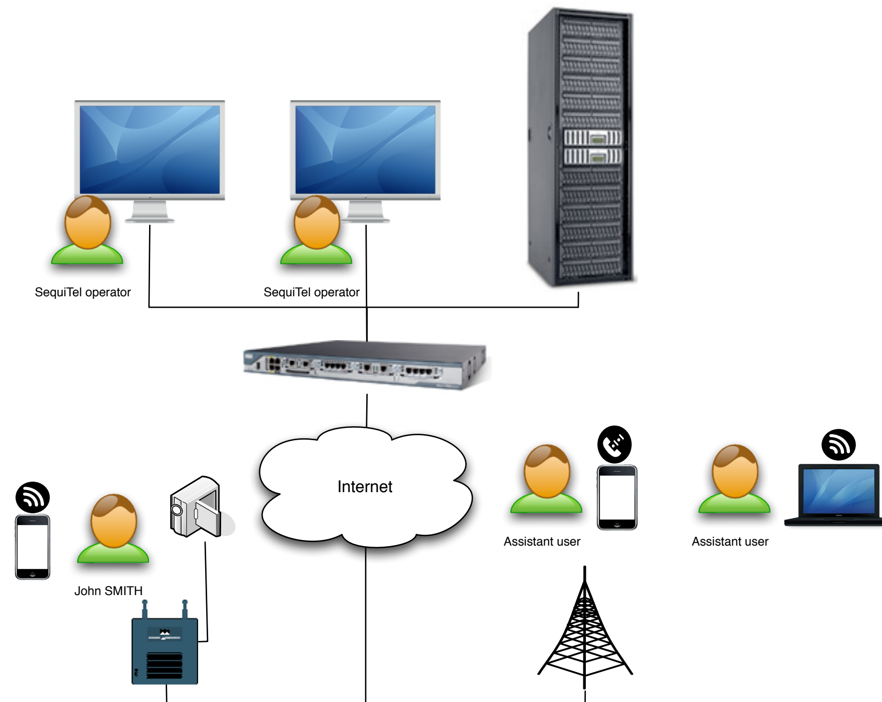

Figure 2.5: SequiTel Example

figure. They use terminal computers connected to the SequiTel servers (connected on the We).

This typical situation illustrates three users categories: tele-cared users, assistant users and operators. These actors interact with different types of devices on different kinds of networks. Without a suitable middleware, the development of such a system would be tedious. Why ? Firstly, programmers have to take into account the communication between devices on different supports. Secondly, since this application requires confidentiality, integrity and authenticity in data transmission, programmers have to choose or create security protocols. Finally, they have to face the transient connection of peers. Peers can join and leave the application whenever they want.

Thanks to SMEPP, all of these concerns are no more in programmers' mind. They have just to design an abstract SMEPP architecture. In this architecture, each users category is associated to a program application: an assistant application, tele-cared user application and operation application. What's more, embedded equipments can run their own SMEPP applications. Each instance of these applications creates and becomes a peer: an assistant peer, a tele-cared user peer, an operator peer and an equipment peer. Services such as file sharing, tele-consulting, chat could be modelled by creating SMEPP services provided by these peers. Moreover, we can imagine that groups can be created for each tele-cared user, gathering all his/her family members, friends and neighbors. In addition, alarms and other notifications could be modelled using SMEPP events. `Emergency` and `notification` are the names of these events.

Figure 2.6 uses the syntax of a UML class diagram and illustrates a possible SequiTel implementation in terms of peers and SMEPP services. This diagram is composed of four boxes (UML package notation). These boxes called *Assistant*, *TelecaredUser*, *Operator* and *Equipment* represent peers described above. Each box contains UML classes which represent services provided by the associated peer. Services operations are listed in each service. This diagram is a simple *example* of the real application.

The assistant peer:

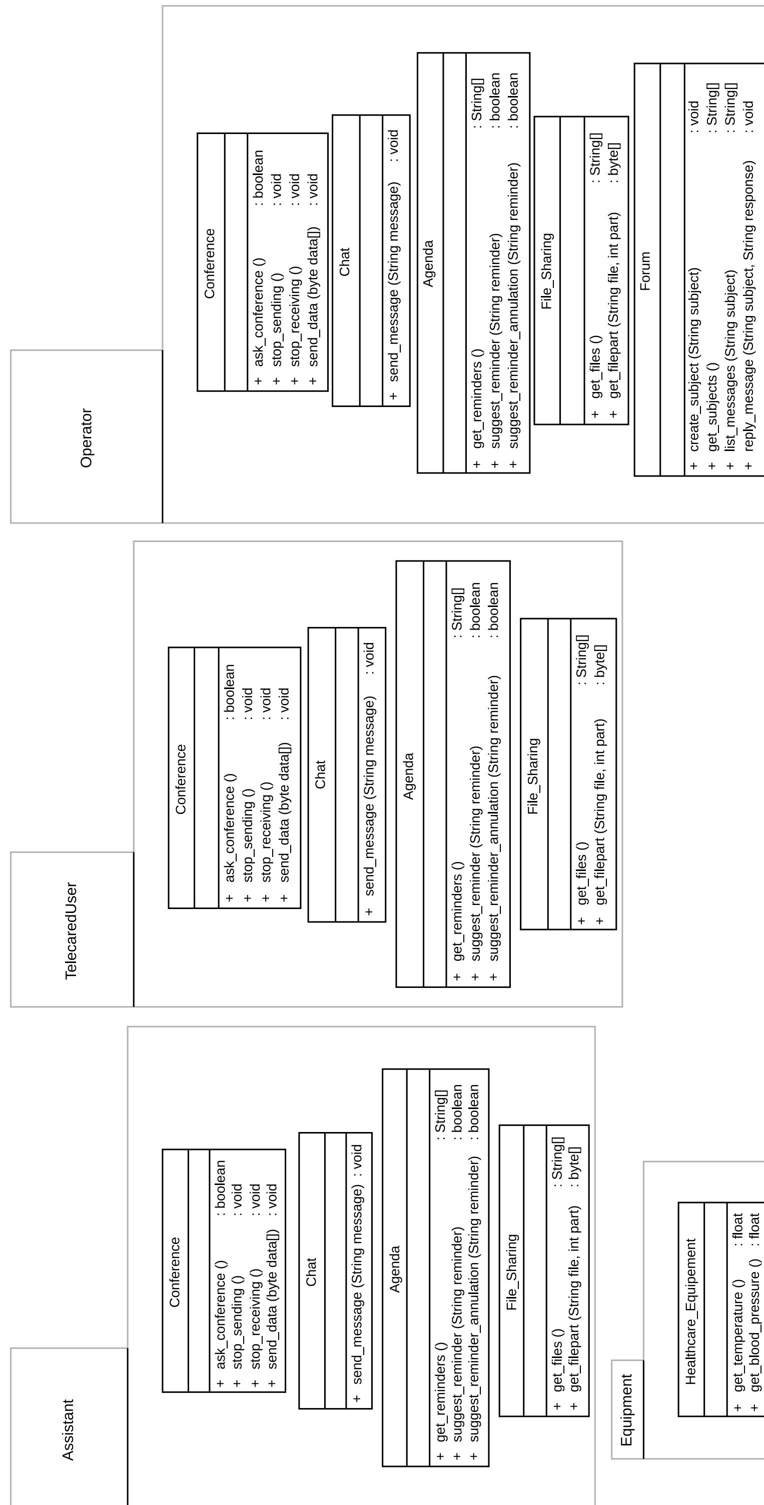

Figure 2.6: SequiTel Peer Services

- `Conference` service:
  - `ask_conference(): boolean`
    The caller asks to the callee an unidirectional video conference. The orientation of this conference is from the callee to the caller. If the caller accepts the invitation it returns true and is ready to receive conference data, else the callee returns false.
  - `stop_sending()`
    The caller notifies that it stops shooting video. The unidirectional link caller-callee is stopped.
  - `stop_receiving()`
    The caller notifies that it stops receiving video. The unidirectional link callee-caller is stopped.
  - `send_data(byte data[])`
    The caller sends video captured by its video camera. We assume that the callee has previously asked to the caller a video conference (see the first method).
- `Chat` service:
  - `send_message(String message)`
    This operation allows to send a textual message to a remote user.
- `Agenda` service:
  - `get_reminders(): String[]`
    A remote peer can ask to this service reminders associated to the remote peer. This peer can filter the returned values (the reminders) according to the caller or the group in which the service is published.
  - `suggest_reminder(String reminder): boolean`
    This operation returns true if the invoked service accepts to add the given reminder to its own agenda. This entry in his calendar is shared between the caller and the callee. If the service doesn't accept this proposition it returns false.
  - `suggest_reminder_annulation(String reminder): boolean`
    The caller wants to cancel one of its previous reminder. If the remote service is agree it returns true, else it returns false.
- `File_Sharing` service:
  - `get_files(): String[]`
    This operation returns file references shared by the service. These references contain the file identifiers and the number of parts associated to each file.
    For example, the returned values: [<myReport.pdf,3>, . . . , <myCare.doc,1>]
  - `get_filepart(String file, int part): byte[]`
    The caller retrieves the part which have the number identified by the "part" parameter of the file "file".
    For example as mentioned above, `myReport.pdf` is composed of three parts: part 1, 2 and 3. The entire file can be downloaded by calling three times the upper method: `get_filepart(myReport.pdf,1)`, `get_filepart(myReport.pdf,2)` and `get_filepart(myReport.pdf,3)`.

The telecared-user Peer:

- `Conference` service: This service has been previously declared in the description of the Assistant peer.
- `Chat` service: This service has been previously declared in the description of the Assistant peer.
- `Agenda` service: This service has been previously declared in the description of the Assistant peer.
- `File_Sharing` service: This service has been previously declared in the description of the Assistant peer.

Equipment:

- `Monitoring` service:
  - `get_temperature(): float`
    This operation returns the temperature of John's body measured by the embedded equipment providing the service.
  - `get_blood_pressure(): float`
    This methods returns another measure provided by the health-care equipment. This measure concerns the current blood pressure of John.

The Operator Peer:

- `Chat` service: This service has been previously declared in the description of the Assistant peer.
- `Forum` service:
  - `create_subject(String subject)`
    A new subject (also called "topic") is created. A subject has a title and it's composed of user replies. The topic title is given by the sole parameter of this operation.
  - `get_subjects(): String[]`
    This primitive returns all created subject (or "topic") titles. We consider that all subjects have a unique title.
  - `list_messages(String subject): String[]`
    Thanks to this primitive, we can retrieve all replies associated to a given subject title (subjects have a unique title).
  - `reply_message(String subject, String reply)`
    A user can post a reply with this primitive. The first argument identifies the subject. The second argument is the user's reply.
- `Conference` service: This service has been previously declared in the description of the Assistant peer.
- `Agenda` service: This service has been previously declared in the description of the Assistant peer.
- `File_Sharing` service: This service has been previously declared in the description of the Assistant peer.

Now we have described all kinds of peers, we can bring them together and we can obtain concrete solutions.

Browser. The SMEPP API provides all the functions required to implement the SequiTel browser: it provides functions which allow to get applications connected on the SMEPP network and functions returning published services. No any further function is necessary.

Alarm Notifications. Tele-cared-users can easily create their own groups by creating SMEPP groups and choosing their members. They can bring together tele-cared-users, assistants, or operators. After that, each group member can raise an alarm to the entire group, or to all groups in which the user is a member by calling a built-in function of the middleware. Thus, this functionality doesn't require any further development, the SMEPP middleware is so powerful that it can provide this feature by itself.

Emergency Alarms from a Mobile Device. As described before, applications developed with SMEPP can be run on different architectures interacting through different networks. If we consider that emergency alarms can be conceived as alarms notifications, they can be quickly implemented into the SequiTel application.

File Sharing. In this example, we consider that each disease is associated to a SMEPP group which brings together people which are concerned by it: patients, specialists, etc. Each member of that groups can share documentation files related to the group topic. For example, John Smith suffers from diabetes and he wants to obtain documents concerning the new insurance facilities for diabetes. John's application can list all peers of the `diabetes group`; for each peer, his application calls `get_files()` and displays the result on the screen (files shared by different peers are displayed only once). John can download a file among this list. Then, his application downloads each file part from different peers sharing the chosen file.

For example, if the document file is composed of three parts and there are three peers sharing this file, SequiTel calls `get_filepart(chosenFile,1)` on the first peer, `get_filepart(chosenFile,2)` on the second peer and `get_filepart(chosenFile,3)` on the last one. Finally, SequiTel merges downloaded parts into a single file.

Tele-consulting. A SMEEP service containing four operations has been created. In the following situation, John Smith wants to begin a video-conference with his doctor Sarah Williams.

Firstly, John's application retrieves the peer associated to Sarah's application then it can call the operation offered by Sarah's conference service called `ask_conference()`. If Sarah agrees to transmit video shooted by her video camera, she can also ask John to begin the video transmission from his video camera to her application.

Once the video-conference has begun, each peer (i.e. Sarah's and John's application) can call the operation `send_data(byte data[])` of the other peer. The given parameter contains video and sound which has just been captured.

If Sarah wants to stop the video-conference, she must call two operations `stop_receiving()` (notifies to John that she stops watching him) and `stop_sending()` (notifies to John that she stops video-transmission) offered by John's video conference service.

Agenda. In this situation, we consider a patient, John Smith, dr. Sarah Williams and John's parents Emily and Jacob Smith. Each of them have their own application: John has his own telecared-user application and Sarah, Emily and Jacob have their assistant applications.

John goes to the Sarah's cabinet. After the consultation, Sarah schedules another appointment with her application which tries to call the operation `suggest_reminder(newAppointmentDate)` of the John's agenda. If his application is not reachable, Sarah's application will retry later. If John's application remains unreachable, Sarah's application calls the same operation `suggest_reminder(newAppointmentDate)`, but now, of the John's parents agenda.

Few hours before the scheduled appointment, if John is not connected, Sarah's application sends an alarm notification to his parents. Few time after the beginning of the appointment, if John is not present, Sarah can cancel the appointment by calling `suggest_reminder_annulation(newAppointmentDate)` of John's agenda and also of John's parents.

Workflow. The flow begins with the operator's applications which periodically call `get_temperature()` and `get_blood_pressure()` offered by embedded sensors. A report can be made and sent via the file sharing service.

Chat. Each SequiTel peer offers a chat service composed of a single operation `send_message(String message)` which permits to send a message to another peer.

Forum. Another way of communicating together is provided by the forum service which is the only service offered by the operator application (only by it).

A forum is composed of subjects (i.e. topics). Each subject has a name and a list of replies written by the SequiTel users. A subject can be created by the operation `create_subject(String subject)` where the single argument is the subject name.

The operation `get_subjects(): String[]` returns all the created subjects and `list_messages(chosenSubject): String[]` returns all the replies associated to a chosen subject. The last operation `reply_message(chosenSubject, reply)` creates a reply to a given subject chosen among the returned subject list.

2.2.4 Traffic Management Application

We have seen in the previous example that SMEPP is a powerful middleware helping developers in their applications development. The following example is a less used distributed application example which intends to demonstrate its ability to create applications over different kinds of networks especially ad-hoc and wireless networks.

Figure 2.7 illustrates a distributed application in the field of *traffic information* which takes place in the Washington town (non realistic example). Two roads are displayed: "*US 66*" (from West to East) and "*Washington 75*" (from South to North). A camera placed on each road segment, counts vehicles on each road segment lane. These cameras, four antennas and two car parks are connected (by wire) to the road information center. Three cars equipped with our system are driving on the two roads: the first car on the road Washington 75, the second and the third on US 66. Cars are able to communicate together and also directly with car parks.

The first purpose of this system is to provide contextual information to drivers: traffic jams, incidents, road constructions or closures, etc. Contextual information also means that these indications are only transmitted to drivers who need this information. The source of this information can be the road information center, or drivers themselves. The second goal is to provide information about available car parks in a given area according to the current car position. Cars must also be able to pay directly through our system; reservations are also permitted.

The biggest challenge of this application development is to allow communication between cars which allows to minimize the number of required antennas. But, thanks to SMEPP, this challenge is out of concern. Indeed, SMEPP can be deployed on different kinds of networks and these networks becomes transparent from the developers' perspective. The diagram displayed in the Figure 2.8 models our system. This diagram is composed of three peer types: `Car`, `Carpark` and `Road_Information_Center`. Each of them has only one service respectively, `Car_Information`, `Carpark_Information` and `Road_Information`.

Car:

- `Car_Information` service:
  - `get_position(): String`
    This operation returns the current GPS coordinates of the car.
  - `get_velocity(): float`
    This getter permits to know the current velocity of the car.

CarPark:

- `CarPark_Information` service:
  - `get_capacity(): int`
    The returned value represents the car park capacity.
  - `get_free_places(): int`
    Return the current number of available car places.

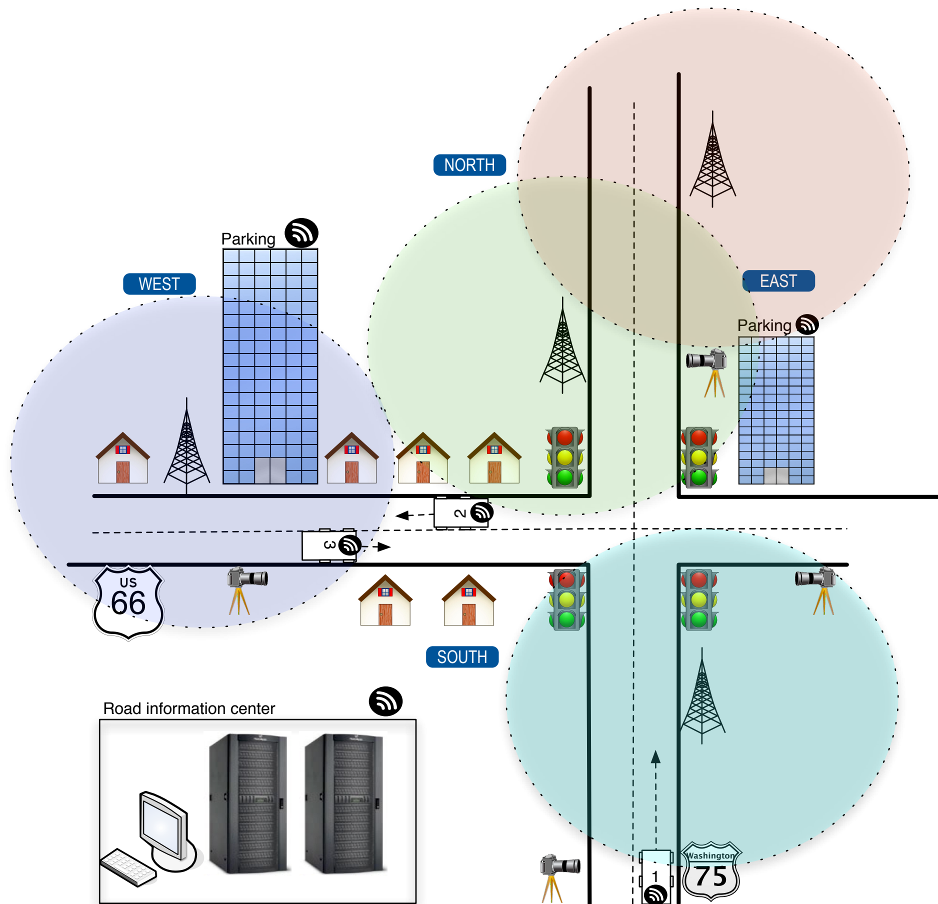

Figure 2.7: Traffic Management Application

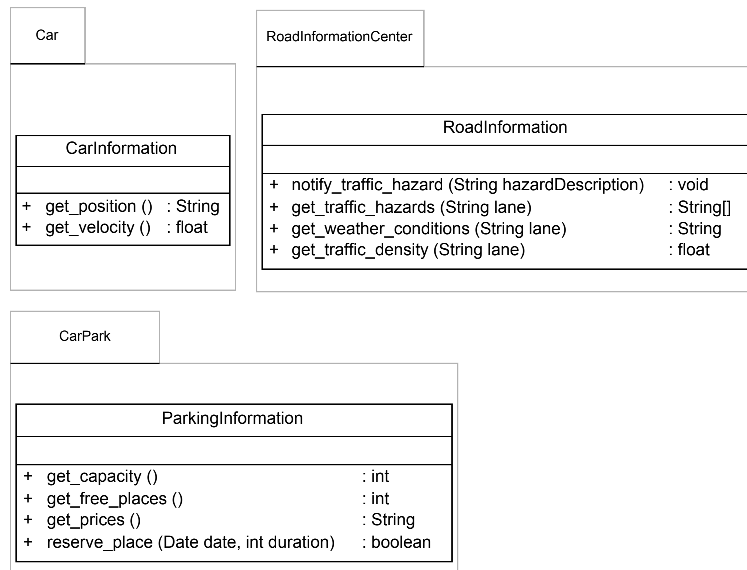

Figure 2.8: Traffic Application Diagram

  - `get_prices(): String`
    Carparks have their own prices list which depend of the duration and the entry time. This operation returns the list associated to the car parking.
  - `reserve_place(Date date, int duration): boolean`
    As mentioned above, drivers can reserve and pay their places directly through the system without drawing any physical money. A reservation requires a date, the beginning of the reservation and its duration. The operations returns true if the reservation succeeds, else it returns false.

Road_Information_Center:

- `Road_Information` service:
  - `notify_traffic_hazard(String hazardDescription): void`
    Drivers can notify to the road information center that they have just encountered a traffic hazard (the position of this hazard is the current car position) described by the single argument of this operation.
  - `get_traffic_hazards (String lane): String[]`
    Get all traffic hazards mentioned by drivers, or by the road information center, associated to a given lane direction. For example: Washington 75 North - South, or US 66 West - East.
  - `get_weather_conditions (String lane): String`
    This operation allows drivers to know the current weather and also the forecast related to a given lane direction.
  - `get_traffic_density (String lane): float`
    Another functionality of this system is to provide information related to traffic congestion on a given lane direction allowing drivers to plan another itinerary.

In Figure 2.7, three cars are displayed. So, three peer instances of `Car` are created. Only once center is displayed so, one instance of the `Road_Information_Center` peer is available. Finally, two car parkings instantiate the `CarPark` peer.

Parking Reservation. Once a car arrives in a city, it can join the associated group. In our example, once the first car arrives in Washington, it can join the Washington group. After that, the car application retrieves all `Road_Information` services of that group. Then, it can call operations `get_free_places()` and `get_prices()` on each of these services. Both operations permit to know the cheapest available car parking.

Once the driver has chosen the best car parking which fits to his requirements, he can pay, or reserve his car place by calling `reserve_place(Date date, int duration)`. In the case of a reservation, the first argument is a future date. Otherwise, it's the current date.

When the car arrives in front of the car parking door, the car can be identified thanks to its position, the car parking can check its reservation and opens the door.

Traffic Information Transmission. The road information center obtains information thanks to road video recorders, call-centers, police, etc. But, knowing the car positions and their velocities (see: `get_position()` and `get_velocity()`) also allows to know if there are slow traffic at some road portions.

The last source of information is cars themselves. They can notify an accident, traffic slow, or any danger by calling `notify_traffic_hazard(String hazardDescription)`. For example, we can imagine that the embedded car device has a button for each of these hazard categories. When the driver pushes a button, the application automatically calls `notify_traffic_hazard(String hazardDescription)` and provides the position of the car and the category associated to the pushed button.

Getting Traffic Information. When an itinerary has been entered in the car device, the application can check on the current road conditions. For example, the first car has planned Washington from South to North then US 66 from East to West. Our application can call these methods:

- `get_traffic_hazards (''Washington 75 South - North''): String[]`,
- `get_traffic_hazards (''US 66 East - West''): String[]`,
- `get_weather_conditions(''Washington 75 South - North''),: String[]`,
- `get_weather_conditions(''US 66 East - West''): String[]`,
- `get_traffic_density (''Washington 75 South - North''): String[]`,
- `get_traffic_density (''US 66 East - West''): String[]`.

These functions allow to have a good knowledge of the *current* situation and allow to schedule another itinerary according to this returned information.

But this situation can evolve so, we can imagine real-time notifications. In order to achieve that, each road lane must be associated to a group. Car devices can join groups associated to their itineraries. If the situation evolves, the road information center can sent a notification event inside the concerned group. This conceptualization permits to minimize communication between cars and the road information center.

Chapter 3

SMEPP service model

The previous section (Section 2.2.2) introduced the SMEPP service model. This chapter focuses on the SMEPP API primitives. Each of them will be introduced and formalized. Before formalizing them, we will see how the abstract language SMoL handles these API functions.

3.1 Primitives of the SMEPP API

An application which uses the SMEPP middleware must be composed of a main code (the peer code) and several service codes. These codes access the SMEPP network via the SMEPP API primitives. Some of these primitives are only callable by peers and some others only by services. The following subsection introduces each of these primitives and specifiy if they can be called by peers or services. As stated before, exceptions are out of concern in our model. They will be simply mentioned in this introduction but not modeled.

SMEPP primitives have the following form:

$$\texttt{output name(input}_1\texttt{, ..., input}_n\texttt{) throws exception}_1\texttt{, ..., exception}_n$$

where:

- **name** is the primitive name,
- **output** is the type of the output parameter,
- $\textbf{input}_i$ is the type of the ith mandatory parameter,
- $\textbf{exception}_i$ is an exception throwable by the primitive.

Some other constructs are also mentioned:

- $< \textbf{part}_1, \ldots, \textbf{part}_n >$ denotes a type composed of several other types
- **param?** refers to an optional parameter
- **par**[] denotes an array

Note that in the following definitions, some exceptions won't be explained. This is the purpose of Table 3.1 (page 33), giving a general meaning and a summarize of exceptions thrown by the primitives.

3.1.1 Peer Management Primitives

newPeer

Primitive signature.

```
peerId newPeer(credentials) throws InvalidCall
```

Primitive description. Peers call this primitive to become peers. The required `credentials`, possibly empty, serve to search, join and authenticate the peer in an existing SMEPP network associated with these `credentials`. If the search failed i.e., no current SMEPP network is associated with these `credentials`, a new SMEPP network associated with them is created and the peer becomes a node of the new created network. `newPeer` returns a new, unique `peerId` peer identifier associated with the new created peer. This identifier is immutable and is retained by the middleware until the peer application terminates. Once the peer is created, calling again this primitive raises an `InvalidCall` exception.

Primitive constraints. Primitive reserved to peers

getPeerId

Primitive signature.

```
peerId getPeerId(serviceId?)  throws InvalidId, InvalidCall
```

Primitive description. Calling `getPeerId` without specifying any parameter returns the `peerId` of the current peer; if the caller is a peer, its identifier is returned. If the caller is a service, the returned peer identifier is the identifier of the peer that has published the service.

If a `serviceId` is specified, the parameter is either a `PeerServiceId`, or a `SessionId`. In the first case, the returned identifier is associated with the peer that has published the service identified by the given argument `serviceId`. In the second case, the primitive returns the `peerId` of the peer that has published a *state-full session-full* service which has one session identified by `serviceId`. An `InvalidId` exception is thrown is `serviceId` is not a valid service identifier: `serviceId` doesn't refer to an existing service, or the referred service is not a state-full session-full service, or the service is not visible to the caller).

3.1.2 Group Management Primitives

createGroup

Primitive signature.

```
groupId createGroup(groupDescription, credentials) throws AccessDenied,
                                                          InvalidCall
```

Primitive description. The first argument `groupDescription` contains the group name, security levels and an optional textual group description. In this thesis, we **only consider** the security parameter called "**Admission security**" which defines how peers can join groups and a SMEPP network, **set to the the default security level**. For further details on the security, please refer [Coa08b].

Three security levels are defined:

- In the first level, level 0, access to the SMEPP network and groups are granted to all peers.
- The second level, level 1, the default security level is based on the assumption that all peers share a set of pre-shared keys. There are two types of keys: one is the key granting access to the SMEPP network, the other gathers the keys granting access to groups. Credentials are included statically in the SMEPP application. That implies that the set of possible groups is known in advance. The security process is simplified, but they bear a higher potential damage to the security in the case of their exposure.
- Level 2 provides the greatest level of security. It is based on asymmetric keys: each peer has a set of private/public keys granting access to groups. This security level requires more complex cryptographic primitives and protocols, but they limit the damage of key exposure.

The second argument, `credentials`, allows to accept or deny a peer inside the group. Only peers providing credentials given at the group creation time can join groups. The provided credentials have to satisfy restrictions `securityInformation`, otherwise an `AccessDenied` is thrown. As said above, we only consider the default level for group admission. So, the provided credentials must contain pre-shared keys of the group and cannot be empty (as it must be in level 0).

Creating groups with the same name doesn't raise an exception. The SMEPP service model assumes that entities can search for a group thanks to the group description and/or on the services they offer. Groups are identified by a unique `GroupId` returned by `createGroup`. Once created, the creator belongs to the group and can leave it even if there are other members. Furthermore, a group lasts as long as it contains at least one member.

Note that `credentials` also serve in the `joinGroup` primitive to restrict the access to a group only to peers that have the right credentials. `securityInformation` and `credentials` can also be exploited in the `getGroups` primitive to restrict the visibility of a group only to peers that have the right credentials according to security information.

Primitive constraints. Primitive reserved to peers

getGroups

Primitive signature.

```
groupId[] getGroups(groupDescription?,credentials) throws InvalidCall
```

Primitive description. Entities call `getGroups` to discover available groups corresponding to certain criteria. The primitive returns identifiers of discovered groups. If no group has matched then the returned value is `null`. This optional parameter `groupDescription` acts as a filter on groups returned by the primitive, only groups having certain characteristics are returned. All fields of the group description are optional. For example, the description can contain the name of the wanted group, or a specific description of the required security level. In all the cases, groups with an admission-security level 1 or 2 (groups set to an admission-security level 0 are all visible) require that the given `credentials` match the credentials associated with the group. Otherwise, unmatched groups are not visible and cannot be returned by the primitive.

Primitive constraints. Primitive reserved to peers

getGroupDescription

Primitive signature.

```
groupDescription getGroupDescription(groupId,credentials) throws InvalidGroupId,
                                                                 InvalidCall
```

Primitive description. Peers call `getGroupDescription` to retrieve the group description associated with the group identified by `groupId`. This description is provided by the creator during the group creation time. It contains the group name, security levels of the group and a short textual description. `getGroupDescription` raises an `InvalidGroupId` exception when `groupId` doesn't refer to an existing group, or when the provided `credentials` don't match the security information of the referred group. For example this exception is thrown in the case of a group `groupId` which has an admission-security level set to level 1 or 0 with `credentials` containing public/private keys, or pre-shared keys unmatching the pre-shared key of `groupId`.

Primitive constraints. Primitive reserved to peers

joinGroup

Primitive signature.

```
void joinGroup(groupId,credentials) throws AccessDenied, InvalidGroupId,
                                           InvalidCall
```

Primitive description. This primitive allows to join an existing group. The first argument `groupId` specifies the group to join. The second argument `credentials` authenticates the peer; only peers providing the right credentials (credentials provided at the group creation) can join the group. A peer may join several groups. Moreover, joining the same group several times does not raise an exception. If a peer doesn't provide the right credentials, `AccessDenied` is raised. Indeed, credentials have to be of the same type as described by the security information (see admission-security level) of the referred group and they have to match credentials associated with that group. If the provided `groupId` is not associated with an existing group, the primitive throws `InvalidGroupId`.

Primitive constraints. Primitive reserved to peers

leaveGroup

Primitive signature.

```
void leaveGroup(groupId) throws InvalidGroupId, CallerNotInGroup,
                                InvalidCall
```

Primitive description. Peers use `leaveGroup` to exit a SMEPP group identified by `groupId`. Of course, the caller has previously joined the group or created the group. Otherwise, `CallerNotInGroup` is thrown. If no current group is associated with this identifier, `leaveGroup` raises `InvalidGroupId`. Once left, all services published by the caller inside the left group are unpublished. Subscriptions, unsubscriptions, invocations made by the peer and its services are also removed. An exception can also be thrown if the peer has still sessions in that group. If the caller was the last group member then the group is removed.

Primitive constraints. Primitive reserved to peers

getIncludingGroups

Primitive signature.

```
groupId[] getIncludingGroups() throws InvalidCall
```

Primitive description. Peers call this primitive to get identifiers of the group in which the caller belongs to.

Primitive constraints. Primitive reserved to peers

getPublishingGroup

Primitive signature.

```
groupId getPublishingGroup(serviceTypeId) throws InvalidCall, InvalidId
```

Primitive description. getPublishingGroup returns the identifier of the group in which the service identified by serviceTypeId has been published. This identifier is either a PeerServiceId, a SessionId, or a GroupServiceId. All services published with the same contract in the same group have the same GroupServiceId. So, in the last case, the returned group identifier is associated with all services identified by serviceTypeId. In the first case, the primitive returns the group where the sole service identified by serviceTypeId has been published. In the second case, the given parameter is a SessionId, the returned group identifier is associated with the service which contains a session identified by this parameter. If serviceTypeId is not a valid service identifier (non existing service, service not visible to the caller, ...), the primitive raises an InvalidId exception.

Primitive constraints. Primitive reserved to peers

getPeers

Primitive signature.

```
peerId[] getPeers(groupId?,credentials) throws InvalidGroupId, InvalidCall
```

Primitive description. getPeers returns the list of peers member of the group identified by groupId. Service calls to getPeers cannot specify a groupId while peer calls must specify it. Otherwise, an InvalidCall exception is raised.

InvalidGroupId is raised when groupId doesn't refer to an existing group, or when the provided credentials don't match the security information of the referred group. For example this exception is thrown, in the case of a group groupId which has an admission-security level set to level 1 or 0 where credentials contain public/private keys, or pre-shared keys unmatching the pre-shared key of groupId.

3.1.3 Service Management Primitives

publish

Primitive signature.

```
<groupServiceId, peerServiceId>
          publish (groupId, contract, grounding) throws InvalidServiceSpecification,
                                                        InvalidGrounding,
                                                        InvalidGroupId,
                                                        CallerNotInGroup,
                                                        InvalidCall
```

Primitive description. This primitive allows peers to publish a service defined by the service contract contract inside a group identified by groupId. The specific service grounding is contained in the grounding argument. For further details on service contracts and groundings, please see the Subsection 3.4.1. The primitive returns a GroupServiceId which identifies all services defined by the same contract contract and published in the same group groupId. The second identifier, peerServiceId identifies the service published by the current peer among all other published services (regardless of the service type and the group in which the service is published).

Furthermore, republishing a service-in-use doesn't raise an exception. This is equivalent to calling unpublish followed by publish. Of course, the re-published service possibly receives new identifiers. Consequently, an invoker still interacting with outdated service identifiers will receive an exception by the middleware.

InvalidGroupId is raised when groupId doesn't identify a current group; CallerNotInGroup is raised when the caller is not a member of the group identified by groupId. If the given argument contract is not well-formed, an InvalidServiceSpecification is thrown. InvalidGrounding is raised in case of an invalid service grounding.

Primitive constraints. Primitive reserved to peers

unpublish

Primitive signature.

```
void unpublish(peerServiceId) throws InvalidServiceId, CallerNotServiceOwner,
                                     InvalidCall
```

Primitive description. A peer stops to offer a service identified by `peerServiceId` in a group by unpublishing it. This primitive is automatically called when a peer leaves a group where it has published services. All these services published in the unjoined group are automatically unpublished. The primitive is also automatically called when a peer exits the network, all these services are unpublished. If the caller of `unpublish` has not published the service that it attempts to unpublish, `CallerNotServiceOwner` is raised. `InvalidServiceId` is thrown if the referred service doesn't exist.

Primitive constraints. Primitive reserved to peers

getServices

Primitive signature.

```
<groupId?,groupServiceId,peerServiceId>[]
        getServices(groupId?, peerId?,servicecontract?,
                    maxResults?,credentials) throws InvalidGroupId
                                                    InvalidPeerId
                                                    InvalidServiceSpecification
                                                    InvalidCall
```

Primitive description. `getServices` retrieves group identifiers, group service identifiers and peer service identifiers of published services that match the specified arguments. Arguments act as filters and are cumulative. Group identifiers returned by the primitive are not present if the caller is a service. Indeed, they are implicit seeing that services can only retrieve services of the group in which theyare published. It is for this reason that services calls where `groupId` is specified, leads to an `InvalidCall` exception.

`groupId` (only peers are allowed to specify it) restricts returned services to services published inside the group `groupId`. Note that, if the caller is a peer, it is not necessary that the caller belongs to the group `groupId`. The last argument, `credentials` allows the caller to get services of different groups. Indeed, each group is protected thanks to credentials. Only credentials matching groups' credentials, authorize the caller to retrieve services of matched groups. For further details, see `createGroup`. The argument `peerId` limits the visibility to services provided by the peer `peerId`. If no peer can be identified thanks to `peerId`, `InvalidPeerId` is raised.

The optional contract `contract` serves as a template for the matching process. The core of the matching process compares the given contract and contract of network services in a syntactic (services share the same operations), behavioural (services have a similar behaviour), computational (QoS requirements are satisfied) and semantic (ontologies) point of view. If the given contract is not well-formed, `InvalidServiceSpecification` is raised. Finally, maximum `maxResults` results are returned by the matching process and so, by the primitive.

getServiceContract

Primitive signature.

```
contract getServiceContract(serviceTypeId) throws InvalidServiceId,
                                                  InvalidCall
```

Primitive description. Entities call `getServiceContract` to retrieve the service contract of a service corresponding to `serviceTypeId`. This parameter is either a `GroupServiceId`, a `PeerServiceId`, or a `SessionId`. In the former case, all services published with the same contract in the same group have the same `GroupServiceId` so, the returned contract is associated with services identified by `serviceTypeId`. In the second case, the primitive returns the group where the only service identified by `serviceTypeId` has been published. In the last case, the given parameter is a `SessionId`, the returned contract is associated with the service which contains a session identified by this parameter. If `serviceTypeId` is not a valid service identifier (no service identified by `serviceTypeId` or a service not visible to the caller), the primitive raises `InvalidServiceId`.

startSession

Primitive signature.

```
sessionId startSession(serviceTypeId) throws InvalidServiceId, AccessDenied,
                                             CannotStartSession, InvalidCall
```

Primitive description. Once published a state-full session-full service has no associated session. So, there is no running service instance provided by the peer in the group in which the service has been published. A session is created every time `startSession` is called; as explained in the Section 2.2.2, a **session** can be conceived as a virtual communication channel between clients and a running service instance. All sessions of the same service instance are executed in a parallel way.

The argument `serviceTypeId` can be either a `GroupServiceId`, or a `PeerServiceId`. In the last case, `startSession` looks for one service instance among service instances of the same group sharing the group service identifier `serviceTypeId`. The chosen service instance is identified by a peer service identifier. Once this identifier retrieved, the behaviour of the primitive is the same as calling `startSession` with a `serviceTypeId` belonging to `PeerServiceId`. In the second case, `serviceTypeId` is a member of `PeerServiceId`, the primitive creates a new session linked with the service instance identified by `serviceTypeId` and returns its unique session identifier. The returned `sessionId` can be used by the caller to communicate with the associated running service instance. Sessions can be shared among peers and their services at the application level. Moreover, a peer or a service can interact with the same service using several sessions.

If the input `serviceTypeId` does not refer to a valid state-full session-full service, the middleware throws an `InvalidServiceId` exception. `AccessDenied` is raised when the caller (peer or service) doesn't belong to the same group as the service specified by `serviceTypeId`. Furthermore, if for any reason, the middleware cannot start a session, it raises an `CannotStartSession` exception.

3.1.4 Message Management Primitives

invoke

Primitive signature.

```
output?  invoke(serviceTypeId, operationName,
                input?, doReturnResult?, timeout?)  throws InvalidServiceId,
                                                           InvalidSessionId,
                                                           InvalidOperation,
                                                           AccessDenied,
                                                           ConcurrentRequest,
                                                           InvalidInputParameter,
                                                           InvalidOutputParameter,
                                                           InvalidCall, ExpiredTimeout
```

Primitive description. `invoke` serves to call a one-way or request-response operation called `operationName` of a service instance identified by `serviceTypeId` which can be either a `GroupServiceId`, a `PeerServiceId`, or a `SessionId`. The second type allows a peer to directly invoke a service instance in a given group. Other identifiers types are explained in the following items. Given identifiers depend on the service type:

- If the invoked service is state-less, then `serviceTypeId` must be either a `GroupServiceId`, or a `PeerServiceId`. So, it *cannot be* a `SessionId` (this service type doesn't support sessions). If a `GroupServiceId` is provided, the middleware is in charge of selecting a service instance identified by a `PeerServiceId` and by `serviceTypeId`. Indeed, all services published with the same contract in the same group receive the same `GroupServiceId` allowing entities to blindly call this service type in that group.

- The invoked service is state-full session-full, the `serviceTypeId` must belong to `PeerServiceId`. So, it cannot be a member of `SessionId` (the service doesn't handle sessions) neither a `GroupServiceId`. Seeing that this kind of applications keeps track of interactions with clients, it is important to know exactly which is the invoked service by providing a `PeerServiceId`.

- If the invoked service is state-full session-full, the given identifier must be a `SessionId`. Indeed, this kind of services can only be used through sessions.

`input` is the input (can be empty) of an operation invocation while `output` (can be empty) is the result. The argument `doReturnResult` (false by default) can only be specified in the case of a request-response operation. If its value is false, `invoke` returns void after a service instance call a corresponding `receiveMessage`. Otherwise, `doReturnResult` is set to true (in the case of a `request-response` operation), the caller of `invoke` remains blocked until a service instance calls `receiveMessage`, processes the invocation, transmits the result to the caller thanks to the `reply` primitives and until the caller receives the result which is then returned by `invoke`.

SMEPP does not allow a running service instance to process (see `receiveMessage` and `reply`) concurrently the same request-response operation called by the same entity. Otherwise, the model would have to implement additional mechanisms for correlating message replies (see `reply`) with message requests. Two scenarios are possible:

- The invoker uses a group service identifier to call a request-response operation (the service must be state-less). Services identified by this group service identifier processing a request from this invoker are considered *not available*. So, if all services identified by the given group service identifier are not available, the invoker will not receive a `ConcurrentRequest`, but an `InvalidServiceId` exception.

- In other cases (`serviceTypeId` belonging to `PeerServiceId`, or to `SessionId`), if a client attempts to invoke concurrently (before retrieving the previous invocation) the same request-response operation associated with the `serviceTypeId`, it will get a `ConcurrentRequest` exception.

receiveMessage

Primitive signature.

```
<callerId,input?> receiveMessage(operationName,timeout) throws InvalidOperation,
                                                              InvalidInputParameter,
                                                              InvalidCall
                                                              ExpiredTimeout
```

Primitive description. Services call this primitive to retrieve an invocation of the operation called `operationName`. Of course, the operation name must have been declared in the service contract of the primitive caller, otherwise `InvalidOperation` is raised. The primitive returns the identifier of the caller and the input provided at the invocation time. `callerId` belongs either to `PeerId` (the invoker is a peer), or to `PeerServiceId` (in case of state-less or state-full session-less service), or to `SessionId` (a state-full session-full invoker). This identifier is used to reply to the invoker (see the next primitive), or to send an event to the invoker, or to call one of its operations (if the invoker is also a service). If no message is received during the `timeout` interval, `ExpiredTimeout` is raised.

`receiveMessage` marks the start of an operation execution i.e., the request processing; in the case of a request-response operation, `reply` marks the end of its execution. For more details on the relation between `invoke`, `receiveMessage` and `reply`, please see the previous primitive.

Note that the SMEPP model does not allow service instances to process concurrent requests from the same client. However, `receiveMessage` will not notify such failures. Further details in the `invoke` primitive description.

Primitive constraints. Primitive reserved to services

reply

Primitive signature.

```
void reply(callerId,operationName,output?,faultName?)  throws InvalidPeerId,
                                                              InvalidServiceId,
                                                              InvalidSessionId,
                                                              InvalidOperation,
                                                              InvalidOutputParamter,
                                                              MissingReceiveMessage,
                                                              InvalidCall
```

Primitive description. A service uses `reply` to give the result of a request-response operation invocation to the caller identified by `callerId`. The concerned operation is `operationName`. So, `reply` marks the end of a request-response operation execution started when `receiveMessage` received an invocation.

For each `reply` the caller must have executed one corresponding `receiveMessage` previously. A corresponding received message means that an invocation has been made by an entity identified by `callerId` (identifier returned by `receiveMessage`) on the operation called `operationName` provided by the caller of the `reply` primitive. Otherwise, `MissingReceiveMessage` is raised. This exception is also thrown if a service tries to reply several times to the same invocation. For example, a peer identified by `pid1` has invoked (`doReturnResult` set to true) a request-response operation called `getFiles` of a state-less service identified by a peer service identifier `psid1`. If `psid1` calls `reply(pid1,getFiles,myOutput)`, it must have previously called `receiveMessage(getFiles)` and received the message from `pid1` (`callerId = pid1`). For further details on the relation between `invoke`, `receiveMessage` and `reply`, please see the explanation of the `invoke` primitive.

The optional `output` is the result of an invocation (can be empty). The primitive can also signal to the caller an erroneous behaviour of an operation. In such cases, operation callers will get a **user exception** called `fault` (see `invoke` and `receiveResponse`) associated with some data contained in the

`output` parameter (can be empty).

Note that between an invocation and the retrieving of its result, an operation caller cannot call the same-request response operation again provided by the same provider i.e.,the invocation target is the same service identifier. But, a misuse of such operations *doesn't lead* to a `ConcurrentRequest` exception raised by `reply` so, service providers will not be notified of such failures. `callerId` is either a `PeerId` (associated with `InvalidPeerId`), a `PeerServiceId` (a non-existing identifier raises an `InvalidServiceId`), or a `SessionId` (if no session can be identified, `InvalidSessionId` is thrown). If `operationName` is not specified in the service contract of the caller, or the operation is not a request-response operation, `InvalidOperation` is raised. A type mismatch between the specified output parameter and its type declared in the service contract leads to an `InvalidOutputParameter`.

Primitive constraints. Primitive reserved to services

receiveResponse

Primitive signature.

```
output receiveResponse(serviceTypeId,operationName,timeout?)  throws InvalidServiceId,
                                                                     InvalidSessionId,
                                                                     InvalidOperation,
                                                                     ConcurrentRequest,
                                                                     InvalidOutputParamter,
                                                                     MissingInvokeMessage,
                                                                     InvalidCall,
                                                                     ExpiredTimeout
```

Primitive description. `receiveResponse` is *used by operation callers* to retrieve the result of an operation call. But, operation execution can also encounter an error. In such cases, an exception is thrown by `receiveResponse` and can be associated with data explaining the reason for this miss behaviour. The caller of `receiveResponse` must have successfully performed a corresponding invoke on the operation `operationName` of the service instance identified by `serviceTypeId`. `operationName` must be a request-response operation. Moreover, at the invocation time, the caller must have set `doReturnResult` to `false` (which is the default value if this parameter is not specified). Further, the invoker cannot have already received the result of this invocation. Failing one of these conditions causes a `MissingInvokeMessage` fault. For details, please refer to the `invoke` primitive.

`receiveResponse` remains blocked until the corresponding `reply` has not been called by the service instance `serviceTypeId` yet. The optional argument `timeout` allows callers to set a limit to the execution time of this primitive. If no response is received during the `timeout` interval, `ExpiredTimeout` is raised. In the case of a request-response operation, if the caller attempts to invoke again the same operation provided by `serviceTypeId` before getting the previous result, `receiveResponse` throws a `ConcurrentRequest` exception. Further details in the `invoke` primitive description.

If `serviceTypeId` is a `GroupServiceId` or `PeerServiceId`. If it refers to an non-existing service, `InvalidServiceId` is raised. Otherwise, `serviceTypeId` is a `SessionId` and must identify an existing session, else `InvalidSessionId` is thrown. Calling an operation not declared in the service contract of the callee leads to an `InvalidOperation`. `InvalidOutputParameter` means a type mismatch between the specified output parameter and its type declared in the service contract.

3.1.5 Event Management Primitives

event

Primitive signature.

```
void event(groupId?,eventName,input?)  throws InvalidGroupId, CallerNotInGroup,
                                              InvalidEvent, InvalidCall
```

Primitive description. Entities (service or peer) call this primitive in order to raise an event named `eventName`. The optional `input` serves to specify the data associated with the raised event. The optional `groupId` restricts the visibility of the event to the group identified by this argument. Otherwise, in the case of a not specified `groupId`, the event called `eventName` is raised inside all groups in which the caller is a member. `groupId` can only be specified by peers else, an `InvalidCall` is thrown.

If the sender is a service, this service must have declared this event name in its service contract (see the service contract model in Section 3.4.1). Otherwise, an `InvalidEvent` is raised notifying that `eventName` has not been declared in the service contract of the sender. An event sent by a service is raised inside its publishing group.

Moreover, `event` is a non-blocking operation. The middleware is in charge of forwarding event to subscribers that are currently listening (see `subscriber` and `receiveEvent`). `InvalidGroupId` is raised if the specified argument `groupId` is not associated with an existing group. If the group exists, but the sender, a peer, doesn't belong to that group, `CallerNotInGroup` is raised.

receiveEvent

Primitive signature.

```
<callerId,input?> receiveEvent(groupId?,eventName,timeout?)  throws InvalidGroupId,
                                                                    CallerNotInGroup,
                                                                    InvalidCall,
                                                                    ExpiredTimeout
```

Primitive description. This primitive listens to events raised inside groups in which the caller belongs to. In the case of a service, only events raised in its publishing group can be retrieved. In the case of a peer, retrievable events are events raised inside groups in which the peer is a member. Only events called `eventName` are retrieved by this primitive. The optional `groupId` can only be specified by peers, else an `InvalidCall` exception is thrown. This argument restricts the reception of events called `eventName` raised inside the group identified by `groupId`. Of course, the identifier must be valid, else an `InvalidGroupId` is thrown. If the peer doesn't belong to that group an `CallerNotInGroup` is thrown.

Receiving an event E in a group G implies a previous subscription allowing the caller to retrieve E raised inside G. For further details, please see the `subscribe` primitive. `receiveEvent` returns the identifier of the event sender, a `CallerId` which is either a `PeerId` (the sender is a peer), a `PeerServiceId` (the sender is either state-less service or state-full session-less service), or a `SessionId` (the sender is state-full session-full service). The output of `receiveEvent` contains the payload of the received event. The last argument specified the maximum execution time of the primitive call. Once this limit is reached, an `ExpiredTimeout` is thrown notifying to the caller that no event has been received during the specified interval of time.

subscribe

Primitive signature.

```
void subscribe(eventName?,groupId?)  throws InvalidGroupId, CallerNotInGroup,
                                            InvalidCall
```

Primitive description. To register as event listeners, entities call `subscribe`. They can subscribe to:

- the event called `eventName` raised in a group identified by `groupId`,
- the event called `eventName`, no constraint on the group in which it is raised (no `groupId` specified),
- all events raised in a group identified by `groupId` (no given `eventName`),
- all events, no matter event names and their associated groups (no argument specified).

After a subscription, an entity can listen to events (see `receiveEvent`) corresponding to its subscription. The second argument can only be specified by peers, otherwise an `InvalidCall` exception is raised. Indeed, services can only listen to events raised in the group in which theyare published. So, services can only make subscriptions of the second and fourth type. If the group identifier is not associated with an existing group, `InvalidGroupId` is thrown. But, if the subscriber, a peer, doesn't belong to this existing group, the middleware raises `CallerNotInGroup`. For further details, please see the mathematical definition of `SubscriptionsState` and its constraints.

unsubscribe

Primitive signature.

```
void unsubscribe(eventName?,groupId?)  throws InvalidGroupId, CallerNotInGroup,
                                              NotSubscribed, InvalidCall
```

Primitive description. This primitive is the dual of `subscribe`. The caller of `unsubscribe` cancels or partially cancels a previous subscription. Entities can unsubscribe to:

- the event called `eventName`, no constraint on the group in which they are raised (no `groupId` specified),
- the event called `eventName` raised in a group identified by `groupId`,
- all events raised in a group identified by `groupId` (no given `eventName`),
- all events, no matter the event name and the associated group (no argument specified).

The second argument can only be specified by peers, else an `InvalidCall` exception is raised. If the identifier is not associated with an existing group, `InvalidGroupId` is thrown. So, services can only make unsubscriptions of the first and fourth type. If the unsubscriber, a peer, doesn't belong to this existing group, the middleware raises `CallerNotInGroup`. `NotSubscribed` is thrown if the unsubscription doesn't match a previous subscription. For further details, please see the mathematical definition of `UnsubscriptionsState` and its constraints.

3.2 SMEPP Modelling Language

As mentioned in the requirements analysis (see Section 2.1), an abstract language is needed. Indeed, this project takes place in the field of web services which bring interoperability between heterogeneous applications through web standards. This kind of services involves complex message exchanges between several parties. In order to formally analyze such applications, an abstract language is required. Such a language can describe observable behaviour of all parties involved in the interaction without assuming any requirement about underlying languages, protocols, etc. But, it also allows applications not to reveal all their interval decision making and data management. Finally, it provides the freedom to change private aspects of the implementation without affecting the observable behaviour[BL08].

Table 3.1: Summarize of Existing Exceptions

Exception name	Meaning
AccessDenied	The caller does not have the credentials required to perform the requested action, or does not belong to the group in which it asks to communicate.
CallerNotInGroup	The peer does not belong to the specified group, although it has the credentials to join it.
CallerNotServiceOwner	A peer tries to administrate a service which has not been published by it.
CannotStartSession	The middleware is unable to instantiate new sessions for the specified service (e.g., when exceeding the maximum number of supported sessions).
ConcurrentRequest	The caller is notified that it invokes the same request-response operation of the same target (same service instance, or session) before retrieving the result of the first invocation.
ExpiredTimeout	The caller can specify its own limit in the execution time of `invoke`, `receiveMessage`, `receiveResponse` and `receiveEvent`. Once this limit is reached, their executions are stopped and this exception is raised.
InvalidCall	Primitives, expect `newPeer` throw an `InvalidCall` exception if they are called before `newPeer` (the caller is a peer), or before being a service (the caller is a unpublished service). This exception is also thrown when an entity is not allowed to call a primitive, because it is to be used by another kind of entity, either for a peer or for a service.
InvalidEvent	Before raising an event, a service must declare the event name in its service contract.
InvalidGrounding	The specified grounding is invalid (only used by `publish`).
InvalidGroupId	The specified `groupId` does not refer to an existing group, or the group referred by `groupId` is not visible to the caller.
InvalidId	The specified Id does not refer to an existing entity, or refers to an entity which is not visible to the caller.
InvalidInputParameter	The specified input parameter (invokes) does not respect the signature of the operation specified in the service contract.
InvalidOperation	The target entity does not support the specified operation (cfr `invoke` and `receiveResponse`), or the operation is not specified in the service contract of the caller (see `receiveMessage` and `reply`).
InvalidOutput-Parameter	There is a type mismatch between the specified output parameter and its type declared in the service contract, or the caller is expecting an output not consistent with the signature of the contract of the target service.
InvalidPeerId	The specified `peerId` does not exist or is not visible to the caller.
InvalidService-Specification	The specified contract (used by `publish`), or contract template (see `getServices`) is not valid.
InvalidServiceId	The specified `serviceTypeId` is not valid, or refers to a service not visible to the caller.
InvalidSessionId	The specified `sessionId` is not valid, or refers to a service not visible to the caller.
MissingInvokeMessage	The caller (of an operation) is trying to receive a response for an invocation which has not been successfully performed (only thrown by `receiveResponse`).
MissingReceiveMessage	The caller is trying to reply to an operation which has not a pending invocation for which a response has still to be replied (only raised by `reply`).
NotSubscribed	The caller is trying to unsubscribe from an event for which it has not subscribed (only used by `unsubscribe`).

An abstract language has been developed by the SMEPP project under the name of SMoL, "*SMoL Modeling Language*". Many SMoL constructions are taken from BPEL[1]. Both languages are written by using XML. SMoL permits to describe a first specification of a peer or service, describing how the application behaviour interacts with the middleware via its primitives. A peer/service specification can be translated into executable code (e.g. Java). In the case of a service, this specification called behaviour is placed in the behaviour part of a service contract. In the case of a peer, the SMoL specification is contained in a file called behaviour file containing only a peer behaviour. Furthermore, the application specification can be used to simulate this application and to check properties[Coa08a, BP08b].

The following definitions intend to introduce SMoL commands informally. These commands are written in a **pseudo-SMoL** avoiding painful reading. In these definitions, arguments followed by "?" means that these arguments are optional. Boolean conditions are assumed to be defined with XPath[W3C]. For further details about SMoL, please see [Coa08a].

3.2.1 Basic Commands

A basic command is either a SMEPP primitive, or a SMoL command listed in this subsection. We define a local command, inspired by BPEL, as a SMoL basic command or as a structured command.

empty

Command signature.

```
void empty ( )
```

Command description. A call to `empty` is equivalent to a no-op. This command can be used for example as a command contained in a catch (see below). Used in that case, `empty` allows to create a catch that doesn't deal with the associated fault, it is ignored since the catch does nothing once a fault is caught.

wait

Command signature.

```
void wait (for?, until?, repeatEvery ?)
```

Command description. Programs call this command to delay their execution either for a certain time (specified by the `for` parameter), or until a certain moment (defined by the `until` parameter). Both parameters cannot be used simultaneously. The last parameter, called `repeatEvery` can only be used in the context of an `InformationHandler` to define an alarm branch (see below). All parameters are specified thanks to XPath[W3C]. In the case of the `for` and `repeatEvery` parameter, the value is a duration. For example, `wait(''P3DT10H'')` stands for "wait for three days and ten hours". In the case of `until`, the paramater format is a date (at least a year, a month and a day must be specified), for example: `wait(''2007-06-30'')` means "wait until June 30, 2007 at 00:00. Note that, `wait` finishes its execution sooner than expected if:

- `wait` must stop since an exception has been raised in another `Flow` branch,
- `wait` in the context of `Pick` alarm branch where another `Pick` branch is activated,
- `wait` as an alarm branch in an `InformationHandler` where the main command finishes.

[1]Business Process Execution Language (BPEL)[OAS] is a standard executable language for specifying interactions with web services.

throw

Command signature.

```
void throw (faultName, faultData?)
```

Command description. `throw` is used to raise an exception called `faultName` inside a program. The optional `faultData` defines a payload associated with the thrown exception. Exceptions can be caught by using a `faultHandler` (see below). Exceptions can also be thrown by the middleware. In order to distinguish them, an exception raised by the basic command `throw` is a **user exception**; an exception thrown by the middleware is a **middleware exception**.

catch / catchAll

`catch`/`catchAll` serve to process faults[2] raised inside the program calling the command. They can only be used in the context of a `FaultHandler`.

Command signature.

```
faultData?  catch (faultName)
```

Command description. The `FaultName` parameter is the name of the fault to be caught. A raised exception matches a `catch`, if its name is syntactically the same as the `FaultName` parameter. This command can return an optional `FaultData`, the payload of the matched fault. This variable may contain an explanation of the fault and can only be used inside the command associated with the `catch` (see below the `FaultHandler`).

Command signature.

```
<faultName,faultData?> catchAll()
```

Command description. This command matches all faults regardless of their names and returns the name and the optional payload of the matched fault.

exit

Command signature.

```
void exit ( )
```

Command description. `exit` can be used in a service code, or in a peer behaviour. In the first case, calling this command ends the service. In the second case, the peer finishes its execution and all of its services are stopped.

[2]The notion of “fault” is exactly the same as “exception”.

Assign

Command signature.

```
Assign
       Copy
            from
            to
       End Copy
       ...
       Copy
            from
            to
       End Copy
End Assign
```

Command description. This command can be used to assign values to variables. It is similar to any assignment in a classic programming language except that, in this case, one `Assign` can do several assignments at once. Each assignment is defined inside a `copy` clause. An assignment copies the value of the source `from` into the target `to`.

The source `from` can be:

- a variable,
- a variable part,
- a literal (a value which is its own translation)
- an opaque value (programmers hide the value which must be assigned).

The target `to` can be:

- a variable,
- a variable part.

Variables and their parts are defined thanks to XPath queries[W3C].

3.2.2 Structured Commands

Structured commands enclose one or more commands (basic or structured) which are orchestrated by these commands. In the case of an *exception*, commands *stop their executions* and the exception thrown by an inner command is thrown outside the command. In the following description, only normal executions are considered.

Sequence

Command signature.

```
Sequence
         command
         ...
         command
End Sequence
```

Command description. A `Sequence` provides a sequential control-flow. Its commands are executed sequentially in lexical order. The `Sequence` terminates when the last child command ends. If a command throws an exception, following commands are not executed.

Flow

Command signature.

```
Flow
     command
     ...
     command
End Flow
```

Command description. `Flow` introduces concurrent executions. `Flow` commands are executed in a parallel way. This command finishes when all of its commands have finished their execution. If a `Flow` branch (i.e. an inner command) throws an exception, other children commands are stopped and the exception is thrown out of the `Flow`.

While

Command signature.

```
While booleanCondition
      command
End While
```

Command description. `While` has the same semantics as in other programming languages. It provides looping in control-flow. Its inner command is executed as long as the boolean condition (see `booleanCondition`) guard is evaluated to `true`. The condition is evaluated before each cycle. `While` terminates when its guard is evaluated to `false`.

RepeatUntil

Command signature.

```
RepeatUntil booleanCondition
            command
End RepeatUntil
```

Command description. This looping control-flow executes again and again `command` as long as the boolean condition (`booleanCondition`) is evaluated to `false`. Once evaluated to `true`, `RepeatUntil` terminates its execution. So, `command` is always executed at least once.

If-Then-Else

Command signature.

```
If booleanCondition
   command
Else
   command
End If
```

Command description. This command is the well-known if-then-else, a conditional, determinist control-flow. It executes either the first command if the boolean condition (see `booleanCondition`) is evaluated to `true`, or the second one if its evaluated to `false`. `If-Then-Else` ends when the selected command terminates its execution. The then branch is optional.

Pick

Command signature.

```
Pick
     <callerId, input?> = receiveMessage (operationName, timeout)
          command
     ...
     output = receiveResponse (id, operationName, timeout?)
          command
     ...
     <callerId, input?> = receiveEvent (groupId?, eventName, timeout?)
          command
     ...
     wait (for?, until?)
          command
     ...
End Pick
```

Command description. This command introduces non-deterministic choices in SMoL. Indeed, the command to be executed depends on external events, messages and alarms (i.e., a branch with a `wait`) whatever the programmer's choice. `Pick` is composed of a set of **handlers** also called **branches**. These handlers can be associated either with a `receiveMessage`, a `receiveResponse`, a `receiveEvent` or finally with an alarm (i.e. a `wait`). In our example, four handlers are displayed. Each handler is also associated with a command. `Pick` waits for a **branch execution** (see below). The first branch which can be executed, starts its execution and other branches are stopped. If several branches become executable at the same time, `Pick` chooses a branch nondeterministicly. In all the cases, maximum **one command** is executed. A branch is said to be **disabled** if this branch cannot be executed, now and in the future. Otherwise, the branch is said to be **enabled** and it is waiting for starting its execution. If all branches are disabled, `Pick` throws a `NoPickBranchActivable` exception. This exception is also thrown if a `Pick` is composed of any branch. A `Pick` finishes once the sole running inner command finishes its execution.

The branch type defines the condition whether a branch can be executed or not:

- `receiveMessage(groupId, operationName, timeout?)`. A command associated with this handler can start its execution only if a message is received. If no message arrives during the `timeout` period, the branch is disabled.

- `receiveEvent(groupId, operationName, timeout?)`. The command of this kind of handler can start its execution if an event is received before the end of the `timeout` period. After this period, the branch becomes disabled.

- `receiveResponse(groupId, operationName, timeout?)`. A `receiveResponse` branch command is allowed to begin its execution once a reply message is available before the end of `timeout` unit of time. Once this limit is reached, the branch becomes disabled.

- `wait(for?, until?)`. The branch command can begin its execution at the end of the interval (see the `for` parameter), or once the date specified by the `until` parameter is reached. In other words, an alarm branch i.e., is always enabled and cannot be disabled.

InformationHandler

Command signature.

```
InformationHandler
     command

     <callerId, input?> = receiveMessage (operationName)
          command
     ...
     output = receiveResponse (id, operationName, timeout?)
          command
     ...
     <callerId, input?> = receiveEvent (groupId?, eventName)
          command
     ...
     wait (for?, until?, repeatEvery?)
          command
     ...
End InformationHandler
```

Command description. `InformationHandler` receives and processes invocations, replies and events as long as the **main command**, lexically placed first inside the `InformationHandler` is running.

An `InformationHandler` is made of a main command and a set of **handlers** also called **branches**. These handlers can be either a `receiveMessage`, a `receiveResponse`, a `receiveEvent`, or an alarm (i.e. a `wait`). In the above definition, four handlers are specified. Each handler is associated with a command. In contrast to a `Pick`, an `InformationHandler` a branch command can be executed several times (possibly in a parallel way) every time the branch is allowed to be executed.

The branch type defines the condition in which a branch can be executed or not:

- `receiveMessage(groupId, operationName, timeout?)`. A command associated with this handler can start its execution once a message is received and if the main command has not finished its execution. If no message arrives during the `timeout` period, the branch becomes disabled.

- `receiveEvent(groupId, operationName, timeout?)`. The command of this kind of handler can start its execution if an event is received before the end of the `timeout` period and if the main command is still running. After this period, the branch becomes disabled.

- `receiveResponse(groupId, operationName, timeout?)`. A `receiveResponse` branch command starts its execution once a reply message is available before the end of the `timeout` duration and if the main command has not finished its execution. After that, the branch is disabled.

- `wait(for?, until?, repeatEvery?)`. The branch command starts at the end of the interval (see the `for` parameter), or once the date specified by the `until` parameter is reached. If the parameter `repeatEvery` is not specified, the branch becomes disabled so, the branch command is executed only once. If this parameter is specified, the branch command starts after every `repeatEvery` unit of time. So the first or the second parameter defines the first execution date of the handler command, other execution dates depend on the third parameter. Note that, the handler command can be executed in a parallel way if the execution time of the handler command exceeds the time specified by `repeatEvery`.

Once the main command finishes its execution, no handler starts its execution. `InformationHandler` finishes its execution once the main command and all running handlers' command have finished. If an exception occurred in the main command or in one handler command, the entire `InformationHandler` finishes and the exception is propagated.

FaultHandler

Command signature.

```
FaultHandler
                command

                faultData1?  = catch (faultVariable1)
                     command
                ...
                faultDataN? = catch (faultNameN)
                     command
                <faultName, faultData?> = catchAll ( )
                      command
End FaultHandler
```

Command description. A `FaultHandler` encapsulates a command, placed first inside this structured command. If an exception occurred during the main command execution, this command is interrupted, and `catches` are evaluated in lexical order. The first `catch` or `catchAll` which maps the thrown exception, gives the command to be executed. In contrast to other structured commands, the execution continues normally with the command following the `FaultHanlder` in the SMoL program expect if a new exception is thrown outside the command associated with the matched `catch`/`catchAll`, or if no `catch`/`catchAll` has matched the exception. A `FaultHandler` contains maximum one `catchAll` (i.e. it is an optional clause) which must be placed *at the end* of this command. If it is not present, exceptions that don't match any `catch` are propagated outside the `FaultHandler`. Otherwise, if a `catchAll` is mentioned, it matches all faults, as described above. This structured command can be written inside another `FaultHandler` and forms a hierarchy of `FaultHandler`. If an exception is propagated outside the outermost `FaultHandler`, it is forwarded to the program environment.

3.2.3 Examples

In the Subsection 2.2.3 we have introduced a concrete application example called SequiTel. The application displayed in this Appendix C aims at illustrating some features of SequiTel. Thanks to this example, we can see how programmers can specify their programs by showing how they manipulate these primitives.

In this example, the `Equipment` peer and `Operator` peer have been partially implemented. The first peer has been specifically developed for John's equipment since the peer behaviour creates a private group called `SequiTel-JohnSMITH`. But our example can be simply extended to other users' equipment by renaming the private user group. The peer code begins by joining, or creating if they don't exist, two groups called `SequiTel-Operators` and `SequiTel-JohnSMITH`. The same service is published in both groups. This service allows his friends and SequiTel Operators to follow the John's current health care status. The execution time of that peer is limited to one year, after that the program stops.

`Monitoring` offers two operations: `get_temperature` and `get_blood_pressure` and emits two kinds of events: `temperatureMonitoring` and `bloodPressureMonitoring`. The service behaviour executes in a parallel way an `InformationHandler` and a `Flow`. The last structured command checks every 30 seconds if John's health-care status is normal i.e., a temperature between 37°C and 38°C and a blood pressure between 9 and 12. If John's health deteriorates, a corresponding event (a `temperatureMonitoring` or a `bloodPressureMonitoring`) is raised. Since the monitoring service is published in two groups, these alerts are raised inside the friends group and operators group. `InformationHandler` receives request during one year after that, it raises an exception used to stop the entire monitoring service. During this period of time, users can invoke both operations provided by the `monitoring` service. These operations allow users to follow the John's current status.

The `Operator` peer can be used by every SequiTel operators; the peer behaviour starts by joining, or creating (if it doesn't exist) the `SequiTel-Operators` group gathering users and operators. The rest of

Table 3.2: Sets Operators

Operators	ASCII	Meaning
$\emptyset$	{}	empty set
$\{e\}$	{e}	singleton set
$\{e_0, \ldots, e_n\}$	{e0,...,en}	set enumeration
$S \cup T$	S \/ T	union
$S \cap T$	S /\ T	intersection
$e \in S$	e : S	member of
$e \notin$	e /: S	not member of
$S \subseteq T$	S <: T	subset
$S \backslash T$	S - T	set substraction
$\lvert S \rvert$	card(S)	size
$\mathbb{N}$	NAT	natural set
$\mathbb{N}_0$	NAT1	positive natural set
$\mathbb{P}\ S$	POW(S)	powerset
$S \times T$	S * T	cartesian product
$\{z \mid P\}$	{ z | P }	set comprehension

Table 3.3: Relation Operators

Operators	Meaning
<>	empty sequence
seq S	set of finite sequences i.e., each element is a list of ordered S elements

the code listens to alerts during a period of one year. Once an alert is received from a patient, if the peer is not already following the patient's state (else the event is ignored), its starts to continuously retrieve the patient's temperature and blood pressure. Once the patient's state becomes normal, the peer finishes to follow his health-care status.

3.3 Formalization Languages

This section introduces the B-method used in the rest of this chapter in order to formalize the SMEPP service model. This introduction is based on [Leu09] and [Abr06].

The purpose of the B-method is to support all stages of the software lifecycle in a uniform and formal way, from the specification to the executable code generation by successive refinement steps. Each step involves writing mathematical proofs in order to justify its results leading to a correct software. In our model, only one abstract machine will be conceived so, we won't get the refinement process in our introduction.

A software system modelled in B is composed of several abstract machines. An **abstract machine** is very close to certain well-known concepts in programming under the names of modules, classes, or abstract data types. A machine contains a *name* (e.g. domotic in Figure 3.1), a local *state* and a collection of *operations*. The machine state cannot be reached directly; they are always reached through machine operations. They are said to be encapsulated in the machine. Operations will be written in terms of how the user should think about the machine. In other words, they describe *what* the machine will provide, but not *how* it will provide them. This highly conceptual specification allows the expression of both abstract specifications and implementation level code.

Table 3.4: Predicate Operators

Logic	ASCII	Meaning
$P \vee Q$	P or Q	logical or
$P \wedge Q$	P & Q	logical and
$\neg P$	not(P)	negation
$P = Q$	P = Q	equality
$P \neq Q$	P /= Q	inequality
$P \Rightarrow Q$	P =>Q	logical implication
$P \Leftrightarrow Q$	P <=>Q	equivalence
$\forall x : T \bullet P$	!(x).(x:T =>P)	for all
$\exists x \bullet P$	# (x).(x:T & P)	there exists

Table 3.5: Substitution Operators

Operators	ASCII
IF P1 THEN G ELSE G2 END	according to the value of the predicate P1, the substitution G or G2 is applied
ANY z WHERE P THEN G END	this substitution choose any value contained in z where this values satisfies the predicate P then z can be used in the substitution G

3.3.1 Mathematical Theory Based

In order to specify the machine in a formal way, the B-method is based on high-level mathematical theories. Data structures are based on sets (see Table 3.2), relations, functions, sequences (see Table 3.3) and trees. Second-order predicates, see Table 3.4[3], are used to write constraints, requirements, variable typing, constant typing, etc. B-machines can be updated so, their state must be changed. Only substitution operators are allowed to change a machine state. These operators are described in the Table 3.5. These mathematical concepts will be detailed in the rest of this introduction.

3.3.2 Data Specification

We begin by defining the data part of a machine. Figure 3.1 illustrates a sample B-machine. Each part of this machine will be explained in the rest of this section; other possible parts will be ignored in order to limit our introduction to B-constructions used in this thesis.

The state of a machine should correspond to how the machine has to be understood. The `SETS` clause defines mathematics sets not built in the B-method. A set is introduced by its *name*. Then we may find an **enumeration** of its distinct elements. Alternatively, the set is left unspecified, it is said to be **deferred** i.e., its elements will be provided at a later stage during implementation. B-sets are *finite*, *non-empty* and *independent*. This latter characteristic implies that any B-set cannot be restricted by a predicate, equality or inclusion between them, predefined sets (i.e., sets build in the B-method) and other B-elements (such as `CONSTANTS` and `VARIABLES`). In Figure 3.1, sets are enumerated, the `DOOR_STATE` set contains two values: `opened` and `closed`. The second set called `DOOR` has three values: `mainDoor`, `backDoor` and `garage`.

The `CONSTANTS` clause lists all machine constants. Constants can be used to list values of deferred sets. Constants can also be used to define independent sets or sets based on other sets (see below). In our example, only one constant has been declared: `defaultDoor`.

[3] In this thesis, we consider that predicates are evaluated in a lazy way.

```
MODEL
        domotic

SETS
        DOOR_STATE = {opened, closed};
        DOOR = {mainDoor, backDoor, garage}

CONSTANTS
        defaultDoor

PROPERTIES
        (defaultDoor : DOOR)

VARIABLES
        doorState

INVARIANT
        (doorState : DOOR --> DOOR_STATE)  & ( ran(doorState) /= {closed} )

INITIALISATION
        doorState := ( (DOOR * {closed}) <+ {defaultDoor |-> opened} )

OPERATIONS
        currentState <-- getCurrentState(door) =
        PRE
                door : dom(doorState)
        THEN
                currentState := doorState(door)
        END;

        closeDoor(door) =
        PRE
                (door : dom(doorState)) & (doorState(door) /= closed)
        THEN
                IF
                        ran( doorState <+ {door |-> closed} ) = {closed}
                THEN
                        doorState := (doorState <+ {door |-> closed})
                                     <+ {defaultDoor |-> opened}
                ELSE
                        doorState := doorState <+ {door |-> closed}
                END
        END;

        openDoor(door) =
        PRE
                (door : dom(doorState)) & (doorState(door) /= opened)
        THEN
                doorState := doorState <+ {door |-> opened}
        END
END
```

Figure 3.1: B-machine Example

All constants and variables (subsequently introduced) must be typed. The PROPERTIES section contains second-order predicates involving constants and sets of the B-machine. Variables are typed in a further clause called INVARIANT. A constant can be an element of a predefined set, or an element of a set declared in the SETS clause of the B-machine. Another possibility is to declare a constant as a new set gathering values of sets predefined, or as a subset of a set declared in the SETS clause (since these sets are independent). The B-language defines four constructs which can be combined allowing user to define his own constants: **cartesian product**, **power-set** and **set comprehension** operator. These constructs are displayed at the end of Table 3.2. In this Figure, S and T can be listed in the SETS or CONSTANTS clause. The only constant of Figure 3.1 belongs to the DOOR set.

The VARIABLES clause lists all variables of the machine. These variables can be updated by operations and represent the current machine state. Our example only declares one variable called doorState. The next clause defines its type.

All variables must be typed in the same way as constants. Variables are typed in the INVARIANT section. In our example, doorState is a total function from DOOR to DOOR_STATE. In other words, this function associates any door present in the DOOR set to a state. So, the purpose of this function is to keep track of each door state. The second part of the invariant avoids that all doors are closed at the same time. So, there is at least one opened door at any moment. This is translated into a logical predicate which avoids that the function range is limited to a closed. But, the range can be opened,closed (at least one door are opened/closed) or simply opened (all doors are opened).

All variables must be initialized. This requirement is achieved in the INITIALIZATION part. Of course, initializations must respect the machine invariant. Up to now, this clause is the only part containing *substitutions*. Other previous parts contain *predicates*. In our machine, the state function is initialized to a function associating all doors with closed denoted by DOOR * {closed}. But the relation "defaultDoor associated with closed" (written: defaultDoor |−> closed) is overridden to "defaultDoor associated with opened" translated in B to: <+ {defaultDoor |−> opened}. So, once our initialization finished, all doors are closed expect the default door which is opened.

3.3.3 Machine Operations

An abstract machine only allows the usage of operations to access the machine state. An operation has a *name*, *input parameters* (optional), *output parameters*, optional *requires* (containing restrictions on parameters), optional *modifies* (inputs and global variables that may be modified) and *effects* (the operation behaviour).

Operations requires are written in terms of **pre-conditions**: PRE P then S END where P is the pre-condition on the operation. It describes restrictions on parameters and on machine state. Operations can only be called when their pre-conditions are true. Pre-conditions can also include other constraints on the current machine state. Thus, pre-conditions are *more general* than requires (limited to parameters constraints), but include them. Moreover, declaring operation requires implies writing of a pre-condition. A pre-condition P is a second-order *predicate*. S is said to be the body of the operation i.e., it is the effects operation part. It must describe how the machine state is updated and the output to be provided, in terms of an abstract assignment. Abstract assignments are called "substitution". Note that, if an output parameter is declared in the operation signature, this output must be assigned.

Figure 3.1 contains three operations called:

- getCurrentState has one output currentState and one input door. This last parameter is restricted. It must belong to DOOR which is the domain of the function contained in the variable doorState. This operation can be conceived as a *getter*. So, there is no modifies operation part, only an effects part which instantiates the output parameter to the state of the given door.

- closeDoor has only an input parameter called door belonging to the domain of the doorState function. Thus, the parameter value, a door, is associated with a state through the doorState function. This door cannot already be closed (operation requires). The function effects consist of

closing the given door. But, if all doors are now closed, `closeDoor` opens the default door. Openings and closings are achieved by the overriding of the `doorState` function (see the `INITIALIZATION` clause). Moreover, `doorState` represents the operation modifies part.

- `openDoor` acts to the contrary. The given door cannot already be opened (requires part). The operation effects is the opening of the given door.

3.4 SMEPP Primitives Formalization

The main purpose of this section is to see how each primitive interacts with the SMEPP network. All primitives will be modeled as functions from a given network state to an updated network state. In this conception, the *current SMEPP network state is contained in a variable which is updated by the primitive and represents the network state knowledge of* ***a*** *peer*. The goal of this section is *not* to model several peers interacting together inside a SMEPP network, but to highlight interactions of an API primitive with the network in the point of a single peer view. Our model will be as simple as possible, exceptions and failure will not be treated. Moreover, primitives are correctly called, there is always somebody to answer requests, there is no network failure, ...

The first step in the API modeling will be the definition, in the B-language, of the data structures needed to model all the valid network states. Once we have these data structures, we can see how the SMEPP primitives interact with the network thanks to its data representation.

3.4.1 Data Structures Definitions

Data structures can be modeled in the B-language thanks to the usage of mathematics sets declared in the `SETS` and `CONSTANTS` clauses. We begin our model with the notion of **credentials**. All peers have their own rights called credentials. We can define the set containing all the possible credentials used by a SMEPP network as: `Credentials`.

A SMEPP network is composed of peers, services and groups. As mentioned in the previous chapter, SMEPP defines different identifiers. The first identifier allows to retrieve a peer among other peers. We define these identifiers as elements belonging to `PeerId`. So, all peers are identified by a unique element of this set.

The second identifier concerns groups. When a peer wants to join/leave or simply operate functions on groups, the peer needs to identify the concerned group. All groups receive an identifier member of the `GroupId` set.

As described previously, service instances are identified by two or three identifiers:

- all services published in the same group with the same service contract receive the same identifier belonging to the `GroupServiceId` set. Thanks to this identifier, an entity can blindly call the service: no matter the service instance inside a given group (see `invoke` and `startSession`),
- all service instances of a SMEPP network receive a unique identifier member of `PeerServiceId`,
- if the service is *state-full session-full*, each of its session are identified by a unique `SessionId`.

Diagram of Figure 3.2 illustrates the hierarchy among identifiers sets. Diagram elements denoted with a key are super-types and are not directly usable in SMEPP. Each subset of this diagram represents a *partition* of the super-set. We can learn thanks to this Figure that `SessionId` and the `PeerServiceId` set are subsets of `ServiceId`. There is a *one-to-many* relation between each element of `PeerServiceId` and elements of `GroupServiceId` and also between elements of `SessionId` and `PeerServiceId`. `ServiceId` and `GroupServiceId` are subsets of `ServiceTypeId`[4] which represents all identifiers associated with

[4]This notion is not mentioned in the SMEPP documentation

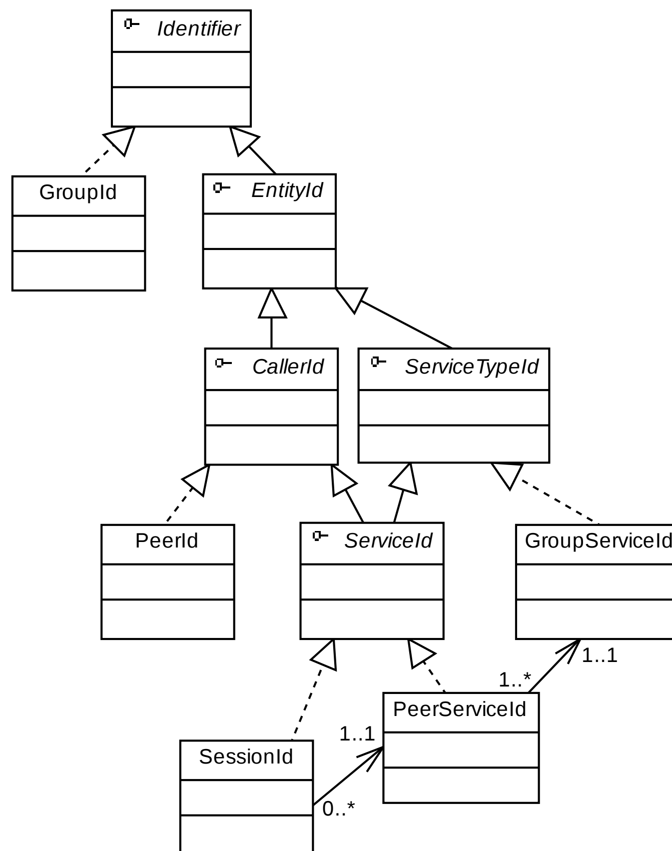

Figure 3.2: Identifiers Sets Hierarchy

services (types, instances and virtual communication channels). `CallerId` elements allow to retrieve the caller of an operation, or the event sender. This set is composed of `PeerId` and `ServiceId`. `CallerId` and `ServiceTypeId` are included in `EntityId` which contains peer identifiers, service instance identifiers and group service identifiers. In the top of the diagram, we can see that all identifiers are members of `Identifier`. That hierarchy among identifiers can be represented by a mathematical representation (Figure 3.3). This graphic permits to easily understand the B-model using mathematics sets to represents each identifier types.

Before introducing complex elements of the data structure, we have to define enumarated types. The first type is related to the service type: `Tservice = {stateLess, sessionLess, sessionFull}` . As said in the SMEPP service description a `state-less` service doesn't keep track of its interactions with clients. So, clients can invoke operations of this service one or more times and in any order without influencing other operations calls. If a service is not state-less then it is a state-full service which is either:

- a state-full session-less service has only one virtual communication channel which is shared by all clients,
- or, a state-full session-full service supporting multiple channels. All of them can be shared among peers and their services. Each operation of a state-full session-full service must be called by providing the session identifier member of `SessionId`.

The second enumerated type concerns the operation type: `Toperation = {oneWay,requestResponse}`. A one-way operation call doesn't stay blocked during the operation execution; no result or fault can be returned by this operation. A request-response operation can throw a fault or return a value. In this case, it is possible to choose: wait for the operation ending and get the returned value, or continue its

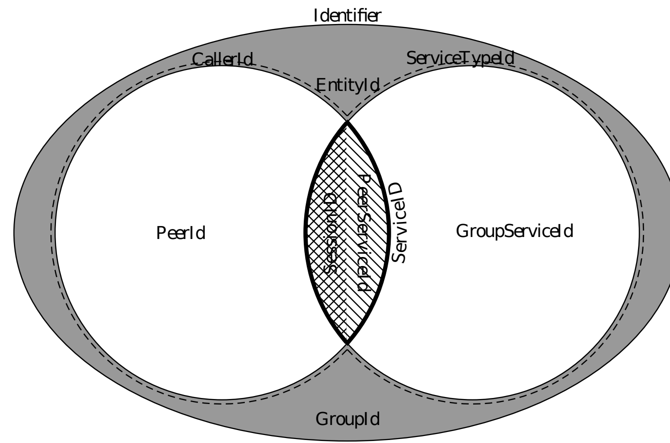

Figure 3.3: Identifiers Sets Hierarchy, Mathematical Representation

own execution and retrieve the returned value or fault latter.

Once we have defined SMEPP identifiers and enumerated types, we can model complex elements of the SMEPP service model. We proceed by steps. Firstly, we begin by defining a *SMEPP service instance*. Note that, *services types won't be taken into account* in the data structure. So, the data structure is composed of services instances. But the service type can be found in its instances through their associated service contract. This model choice suits with the current conception of the implemented SMEPP middleware. We continue with the definition of a *peer* and a *group*. Finally, we conclude by defining interactions inside the network and the definition of a *SMEPP network states*.

Definition of a Service Contract

Services are defined through their contracts which include *information* related to the service (e.g. the service name), *operations* provided by the service and *events raised* by the service code). A service contract can be defined as:

$$\texttt{Scontract = ServiceInfo} \times \texttt{Tservice} \times \mathcal{P}\texttt{(Operation)} \times \texttt{EventsRaised}$$

where:

- `ServiceInfo` is a set containing all the possible service information,
- $\mathcal{P}$`(Operation)` $= \{o \mid o \subseteq$ `Operation`$\}$ is a set containing operation sets.

For example, if we consider:

```
ServiceInfo = {info1,info2},
Operation = {op1,op2}
EventsRaised = {EventsA,EventsB}
```

The service contract set becomes:

```
Scontract = {info1,info2} × {stateLess, sessionLess, sessionFull}
            × P({op1,op2})× {EventsA,EventsB
          = {info1,info2}× {stateLess, sessionLess, sessionFull}
            × { {}, {op1}, {op2}, {op1,op2} } × {EventsA,EventsB}
          = { (info1,stateLess,{},EventsA), ...,
              (info2,sessionFull,{op1,op2},EventsB) }
```

The following sections define the `Operation` set and the `EventsRaised` set. We will see how `op1`, `op2`, `EventsA` and `EventsB` are defined.

Service Operations Definition

Firstly, we define service operations described in the signature part. Operations have a *name* (i.e. a string name), a *type* belonging to `Toperation`, *argument types* and finally *output types*:

$$\texttt{Operation = OperationName} \times \texttt{Toperation} \times SEQ\texttt{(AbstractDataType)} \times SEQ\texttt{(AbstractDataType)}$$

where:

- `AbstractDataType` contains all possible argument types of services operations,
- SEQ`(AbstractDataType)` $= \{(s_1, \ldots, s_n) \mid \forall i, (i \in \mathbb{N}) \wedge (s_i \in AbstractDataType)\}$ each element of this set is a list of abstract data types.

If we consider:

```
AbstractDataType = {String,Int,String[],byte[]}
SEQ(AbstractDataType) = {(), (String), (Int), (String[]), (byte[]), ...,
                         (String,byte[]), ..., (Int,String,byte[]), ...}
OperationName = {''get_filepart'', ''get_files''}
```

According to these values, the operation set becomes:

```
Operation = {(''get_filepart'',oneWay,(),()), ...,
             (''get_filepart'',oneWay,(...),(...)), ...,
             (''get_filepart'',requestResponse,(),()), ...,
             (''get_filepart'',requestResponse,(...),(...)),
             (''get_filepart'',requestResponse,(String,Int),(byte[])),
             (''get_files'',oneWay,(),()), ...,
             (''get_files'',oneWay,(...),(...)),
             (''get_files'',requestResponse,(),()), ...,
             (''get_files'',requestResponse,(...),(...)), ...,
             (''get_files'',requestResponse,(),(String[]))
            }
```

where:

- `(...)` means “a list of abstract data types”.

Among all these possible operations, only two are used in our application (see Figure 2.6). These operations are provided by the `FileSharing` services published by the assistant peer, telecared-user peer and operation peer. We can rewrite the definition of `op1` and `op2` used in the previous section by:

```
op1 ≡ (''get_files'',requestResponse,(),(byte[]))
op2 ≡ (''get_filepart'',requestResponse,(String,Int),(byte[]))
```

Definition of Events Raised by a Service

SMEPP services must define in their contract the name of events they raise. In the following definition, we need to model all possible sets of *event names*:

$$\texttt{EventsRaised} = \mathcal{P}\texttt{(EventName)}$$

where:

- $\mathcal{P}$(EventName $= \{e \mid e \subseteq$ EventName$\}$ each element of this set is a event name set

The example explained in the Section 2.2.3 is composed of two events which have the name `Emmergency` and `Notification`. In the case of this example, this set becomes:

```
EventsRaised = { {}, {Emergency}, {Notification}, {Emergency,Notification} }
```

`EventsA` and `EventsB` used in the service contract definition can be rewritten as:

```
EventsA ≡ {}
EventsB ≡ {Emergency,Notification}
```

SMEPP Service Definition

Now, we can define a service. A service is published inside a *group* (identified by `GroupId`) with a given *contract* which belongs to `Scontract`, a specific *grounding* which is a member of `Sgrounding` and a set of *operations*. Once published, a service instance can be invoked by means of two or three (according to the service type, see `Tservice` in its contract) *identifiers*: `PeerServiceId`, `GroupServiceId` and a set of `SessionId`.

```
Services = GroupId × Scontract × Sgrounding × PeerServiceId
           × GroupServiceId × P(SessionId)
```

where:

- $\mathcal{P}$(SessionId) $= \{s \mid s \subseteq \mathcal{P}$(SessionId) $\}$ is a set of session identifiers sets (i.e. each s is a set of sessions identifiers).

We continue the previous example by defining these values:

```
GroupId = {gid1,gid2}
Sgrounding = {gnd1,gnd2}
PeerServiceId = {psid1,psid2}
GroupServiceId = {gsid1,gsid2}
SessionId = {sid1,sid2}
```

In the previous example, `Scontract` contains many useless contracts. In the rest of this chapter, we only consider one contract describing the file sharing service. So, `Scontract` becomes a singleton set.

```
Scontract  =  {
                ( info1, sessionFull,
                 {
                   (''get_files'',requestResponse,(),(byte[])),
                   (''get_filepart'',requestResponse,(String,Int),(byte[]))
                 },
                 {Emmergency,Notification}
                )
              }
           =  {contract1}
```

In order to minimize the service contract definition length, `contract1` will be used instead of the definition above. We have all the elements required to make our service example:

```
Service = GroupId × Scontract × Sgrounding × PeerServiceId × GroupServiceId
            × P(SessionId)
        = {gid1,gid2} × {contract1} × {gnd1,gnd2} × {psid1,psid2}
            × {gsid1,gsid2} × P({sid1,sid2})
        = {gid1,gid2} × {contract1} × {gnd1,gnd2} × {psid1,psid2}
            × {gsid1,gsid2} × { {}, {sid1}, {sid2}, {sid1,sid2} }
        = {
            (gid1,contract1,gnd1,psid1,gsid1,{}), ...,
            (gid2,contract1,gnd2,psid2,gsid2,{sid1,sid2})
          }
```

The cartesian product generates all the possible service instances. If we assume that the file sharing service is a state-full session-full service, two real service instances can be:

- service1 ≡ (gid1,contract1,gnd1,psid1,gsid1,{}). This service instance has been published in the group identified by gid1 with the contract contract1 and grounding gnd1. This instance is identified by psid1 (the peer service instance) and gsid1 (the group service instance). This instance has currently no created session (the session identifier set is empty).
- service2 ≡ (gid2,contract1,gnd2,psid2,gsid2,{sid1,sid2}). This is an instance of the same service (i.e. service contracts (contract1) are the same), but the service has been published in a different group identified by gid2 and the groundings are different (gnd2 ≠ gnd1). Of course the peer service instance identifier is different (psid2 ≠ psid1) and the group service instance is different (gsid2 ≠ gsid1 because groups are different). Two sessions have been created and identified by: sid1 and sid2.

Peers definition

Now we can define the basic abstraction of a SMEPP network: *the peer*. A peer has an *identifier* belonging to PeerId and offers *services*.

Peer = PeerId × $\mathcal{P}$(Service)

where:

- $\mathcal{P}$(Service) = $\{s \mid s \subseteq$ Service$\}$ each element of this set is a set of SMEPP services offered by a peer.

In the following example, PeerId is defined as {pid1, pid2}. The peers set becomes:

```
Peer = {pid1,pid2} × P({service1,service2})
     = {pid1,pid2} × { {}, {service1}, {service2}, {service1,service2} }
     = { (pid1,{}),(pid1,{service1}),(pid1,{service2}),(pid1,{service1,service2}),
         (pid2,{}),(pid2,{service1}),(pid2,{service2}),(pid2,{service1,service2})}
```

According to our previous application (see the Figure 2.6), we can consider, for example, only these both peers: (pid1,{service1}) and (pid2,{service2}) which have published the same service (i.e. the file sharing service) in two different groups identified by gid1 and gid2.

As stated in the beginning of this chapter, our purpose is to define all the possible SMEPP network states. Each state represents the current connected peers. The Peer set contains all the possible peers, but they are not all connected at the same time so, we need to model a new set which contains all the possible connected peers at any moment. According to our previous definition, this set can be simply defined as the powerset of the Peer set:

ConnectedPeers = $\mathcal{P}$(Peer)

where:

- $\mathcal{P}$(Peer) $= \{p \mid p \subseteq$ Peer$\}$ each element of this set is a set of connected peers.

In the previous example, we have considered two realistic peers among all the possible peers: (pid1,{service1}) and (pid1,{service2}). If we consider that the Peer set is only composed of these two realistic peers, we get:

```
ConnectedPeers = P( {(pid1,{service1}), (pid1,{service2})} )
               = { {}, {(pid1,{service1})}, {(pid2,{service2})},
                   { (pid1,{service1}), (pid2,{service2}) } }
```

The first value represents a SMEPP network where there is currently no connected peer. The second and the third value represent a network with only one connected peer. In the last situation, all peers (i.e., our two peers) are connected.

Groups definition

Before introducing interaction between network entities, we have to define the notion of *groups*. A group has an identifier belonging to GroupId, *credentials* which permit to check whether peers can join the group, a *group description* and a *set of peer* identifiers.

Group descriptions contain the *group name*, *security levels* of the group (more details in the Section 3.1.2) and a textual *description*:

$$\texttt{Gdescr} = \texttt{StringName} \times \{\texttt{0,1,2}\} \times \texttt{String}$$

In the rest of this thesis, we simply consider that a group description belongs to Gdescr avoiding to detail the structure of group descriptions. We can define a group as an 4-uplet:

$$\texttt{Group} = \texttt{GroupId} \times \texttt{Cred} \times \texttt{Gdescr} \times \mathcal{P}(\texttt{PeerId})$$

where:

- $\mathcal{P}$(PeerId) $= \{p \mid p \subseteq$ PeerId$\}$ is the powerset of PeerId. In other words, each element of this set is a set of peer identifiers.

We continue the previous example by keeping these definitions:

```
GroupId = {gid1,gid2}
Cred = {cred1,cred2}
Gdescr = {descr1,descr2}
PeerId = {pid1,pid2}
```

The group sample set is:

```
Group = {gid1,gid2} × {cred1,cred2} × {descr1,descr2} × P({pid1,pid2})
      = {gid1,gid2} × {cred1,cred2} × {descr1,descr2}
        × { {},{pid1}, {pid2}, {pid1,pid2} }
      = { (gid1,cred1,descr1,{}), ..., (gid2,cred2,descr2,{pid1,pid2}) }
```

Among all of these possible groups, we consider only two groups:

- group1 ≡ (gid1,cred1,descr1,{pid1,pid2}). This first group, identified by gid1, is described by descr1. The groups is restricted by cred1 and contains two peers identified by pid1 and pid2.
- group2 ≡ (gid2,cred2,descr2,{pid2}). In this group, there is only one peer identified by pid2.

We are in the same situation as in the definition of the peer set: the upper set contains all the possible groups. What we need is a set containing the *current* groups i.e., each element of this set must be a set of the current groups. As previously solved, we use a powerset:

$$\texttt{CurrentGroups} = \mathcal{P}(\texttt{Group})$$

where:

- $\mathcal{P}(\texttt{Group}) = \{g \mid g \subseteq \texttt{Group}\}$ each element of this set is a set of created group.

Again we can keep these two groups and make our example:

$$\begin{aligned}\texttt{CurrentGroups} &= \mathcal{P}(\{(\texttt{gid1,cred1,descr1,}\{\texttt{pid1,pid2}\}),(\texttt{gid2,cred2,descr2,}\{\texttt{pid2}\})\}) \\ &= \{\ \{\},\{(\texttt{gid1,cred1,descr1,}\{\texttt{pid1,pid2}\})\},\{(\texttt{gid2,cred2,descr2,}\{\texttt{pid2}\})\} \\ &\quad \{(\texttt{gid1,cred1,descr1,}\{\texttt{pid1,pid2}\}),(\texttt{gid2,cred2,descr2,}\{\texttt{pid2}\})\}\ \}\end{aligned}$$

The first element is a situation where there is currently no group. In the second and the third member, only one group is available: the first group identified by `gid1`, or the second group identified by `gid2`. In the last situation, our two groups are present.

Events subscription definition

Now that we have defined the basic entities of a SMEPP network, we can model interactions among peers and services. The following definitions are related to *events* and *operations* invocations.

Receiving events implies a previous subscription of the caller (a peer or a service). Four subscriptions types exist, some of them are reserved to peers. Each subscription type leads to a specific definition:

1. A peer wants to receive events having the specified *name* and raised inside a group identified by a given *group identifier*. This subscription type is limited to peers because a group identifier is specified (service can only receive events raised in the group in which they are published). This subscription type can be defined as:

   $$\texttt{SubscriptionEventInAGroup} = \texttt{EventName} \times \texttt{GroupId} \times \texttt{PeerId}$$

   For example:

   $$\begin{aligned}\texttt{SubscriptionEventInAGroup} &= \{\texttt{Emergency,Notification}\} \times \{\texttt{gid1,gid2}\} \times \{\texttt{pid1,pid2}\} \\ &= \{(\texttt{Emergency,gid1,pid1}),\ldots,(\texttt{Notification,gid2,pid2})\}\end{aligned}$$

   The first value of this set illustrates a peer identified by `pid1` which wants to receive emergency events raised inside the group identified by `gid1`. The last element of this set contains a subscription made by a peer identified by `pid2` which wants to receive notification events raised inside the group identified by `gid2`.

2. The subscriber wants to receive events associated with a given *name*. These events can be raised inside any group in which the subscriber is a member.

   $$\texttt{SubscriptionEventAllGroups} = \texttt{EventName} \times (\texttt{PeerId} \cup \texttt{PeerServiceId})$$

   A possible set can be:

   $$\begin{aligned}\texttt{SubscriptionEventAllGroups} &= \{\texttt{Emergency,Notification}\} \times (\{\texttt{pid1,pid2}\}\cup\{\texttt{psid1,psid2}\}) \\ &= \{\texttt{Emergency,Notification}\} \times \{\texttt{pid1,pid2,psid1,psid2}\} \\ &= \{(\texttt{Emergency,pid1}),\ \ldots,\ (\texttt{Notification,psid2})\}\end{aligned}$$

The first set element represents a subscription made by a peer identified by `pid1` to emergency events raised inside all groups in which this peer is a member. The last element contains a subscription made by a service identified by `psid2` to notification events raised inside the group in which it is published.

3. A peer wants to receive all events raised inside a group identified by a given *group identifier*. This subscription type is limited to peers because a group identifier is specified:

   $$\texttt{SubscriptionAllEventsInAGroup} = \texttt{GroupId} \times \texttt{PeerId}$$

   A sample set:

   $$\begin{aligned}\texttt{SubscriptionAllEventsInGroups} &= \{\texttt{gid1,gid2}\} \times \{\texttt{pid1,pid2}\} \\ &= \{\ \texttt{(gid1,pid1), (gid1,pid2),} \\ &\quad\ \texttt{(gid2,pid1), (gid2,pid2)}\ \}\end{aligned}$$

   Thanks to the first subscription, a peer `pid1` wants to receive all events raised inside the group linked to `gid1`.

4. A *peer* or *service* subscribes to all kinds of events (no group and name restriction).

   $$\texttt{SubscriptionAllEventsAllGroups} = (\texttt{PeerId} \cup \texttt{PeerServiceId})$$

   One possible set can be:

   $$\begin{aligned}\texttt{SubscriptionAllEventsAllGroups} &= (\{\texttt{pid1,pid2}\} \cup \{\texttt{psid1,psid2}\}) \\ &= \{\texttt{pid1,pid2,psid1,psid2}\}\end{aligned}$$

   The first subscription is a peer (associated with `pid1`) subscription which wants to receive all events (no matter the event name) raised inside any group containing this peer. The last one concerns a service which subscribes to all events raised inside the group where it is published.

In these examples, we have considered these values:

$$\begin{aligned}&\texttt{PeerServiceId} = \{\texttt{psid1,psid2}\} \\ &\texttt{PeerId} = \{\texttt{pid1,pid2}\} \\ &\texttt{GroupId} = \{\texttt{gid1,gid2}\} \\ &\texttt{EventName} = \{\texttt{Emergency,Notification}\}\end{aligned}$$

These four upper sets contains *a possible subscription.* Like for the `Peer` set or `Group` set, we need to model all the possible current subscriptions. Thus we have to create a set containing these possible subscriptions states (whatever the subscription type) i.e., each element of a subscriptions state is a current possible subscription which has been made by a current peer, or by a current published service. This new set is defined as:

$$\begin{aligned}\texttt{SubscriptionsState} = &\ \mathcal{P}(\texttt{SubscriptionEventInAGroup}) \\ &\times \mathcal{P}(\texttt{SubscriptionEventAllGroups}) \\ &\times \mathcal{P}(\texttt{SubscriptionAllEventsInAGroup}) \\ &\times \mathcal{P}(\texttt{SubscriptionAllEventsAllGroups})\end{aligned}$$

For example, this set can contain:

```
SubscriptionsState = P({(Emergency,gid1,pid1), ..., (Notification,gid2,pid2)})
                     × P({(Emergency,pid1), ..., (Notification,psid2)})
                     × P({(gid1,pid1), (gid1,pid2), (gid2,pid1), (gid2,pid2)})
                     × P({pid1,pid2,psid1,psid2})

                   = {
                       {},{(Emergency,gid1,pid1)},...,{(Notification,gid2,pid2)},
                       {(Emergency,gid1,pid1),...,(Notification,gid2,pid2)}
                     }
                     × {{},{(Emergency,pid1)},...,{(Notification,pid2)},
                     ...,{(Emergency,pid1),...,(Notification,pid2)} }
                     × { {},{gid1,pid1},...,{(gid1,pid1),(gid1,pid2),(gid2,pid1),
                         (gid2,pid2)} }
                     ×{{},{pid1},...,{pid1,pid2,psid1,psid2}}

                   = {
                         ({},{},{},{}), ...,
                         (
                           {(Emergency,gid1,pid1)}, {(Emergency,pid1)},
                           {(gid1,pid1)}, {pid1}
                         ), ...,
                         (
                           {(Emergency,gid1,pid1), ..., (Notification,gid2,pid2)},
                           {(Emergency,pid1), ..., (Notification,pid2)},
                           {(gid1,pid1), (gid1,pid2), (gid2,pid1), (gid2,pid2)},
                           {pid1,pid2,psid1,psid2}
                         )
                     }
```

The above values represent all the possible subscriptions states of a SMEPP network. If we consider the second displayed state (the second element among these three displayed states) of this set, we can see four subscriptions related to the same peer identified by `pid1`. In the first subscription, this peer wants to receive emergency events raised inside the group identified by `gid1`. The second subscription deals with emergency events raised inside any group in which the peer is a member. The next subscription is related to all events (no event name constraint) raised inside the group identified by `gid1`. In the last subscription, the peer wants to receive all events (no event name constraint) raised inside any group in which it is a member.

But some of these states are not realistic. In the sample set bellow, three subscriptions states are considered. The first state represents a SMEPP network where no subscription has been made yet. In the second and third state, only one subscription has been made. In the second one, the peer `pid1` has subscribed to emergency events raised inside the group `gid1`. In the third one, the peer `pid2` is able to receive all events (no event name constraint) whatever the associated group. The last state means that a subscription has been made by a peer identified by `pid1` which wants to receive all emergency events. Another peer, `pid2` subscribes to all events (no event name constraint) whatever the group.

```
SubscriptionsState = {
                       ({},{},{},{}),
                       ({(Emergency,gid1,pid1)},{},{},{}),
                       ({},{},{},{pid2}),
                       ({},{(Emergency,pid1)},{},{pid2})
                     }
```

Events Unsubscription Definition

SMEPP provides high level subscriptions and unsubscriptions functions. A user can subscribe to events and then constrain his previous subscription. Unsubscriptions require a new data structure, because they act as constraints on previous generic subscriptions and we need to keep this knowledge as long as there is still generic subscriptions.

1. In the first unsubscription type, a peer or service unsubscribes to a given event E. From now, this entity will no longer receive any event called E. The unsubscriber can still want to receive any events raised inside a given group (third subscription type), or inside any group in which it is a member (fourth subscription type).

$$\texttt{UnsubscriptionEvent = EventName} \times \texttt{(PeerId} \cup \texttt{PeerServiceId)}$$

   For example, in the following set, the first peer `pid1` doesn't receive emmergency events anymore. But, this peer can still receive any other kind of events.

$$\begin{aligned}\texttt{UnsubscriptionEvent} &= \{\texttt{Emergency,Notification}\} \times (\{\texttt{pid1,pid2}\} \cup \{\texttt{psid1,psid2}\}) \\ &= \{\texttt{Emergency,Notification}\} \times \{\texttt{pid1,pid2,psid1,psid2}\} \\ &= \{\texttt{(Emergency,pid1), ..., (Notification,psid2)}\}\end{aligned}$$

2. The following definition is the most particular unsubscription. In this situation, a peer (a group identifier is specified, so this unsubscription cannot be made by a service) unsubscribe to the event E in a specific group G. The peer must have previously subscribed to one or more of these subscriptions:

   - a subscription of the second type, to the event E (no group constraint),
   - a subscription to events raised in G (third subscription type),
   - a subscription of the fourth type, to any events raised inside any group containing the peer.

$$\texttt{UnsubscriptionEventInAGroup = EventName} \times \texttt{GroupId} \times \texttt{PeerId}$$

   The following sample set contains two displayed unsubscriptions. The first unsubscription has been made by the first peer, `pid1` which doesn't want to receive emergency events from the first group `gid1`. The last element of this set represents a situation where the second peer `pid2` has unsubscribed to notification events raised inside the second group `gid2`.

$$\begin{aligned}\texttt{UnsubscriptionEventInAGroup} &= \{\texttt{Emergency,Notification}\} \times \{\texttt{gid1,gid2}\} \times \{\texttt{pid1,pid2}\} \\ &= \{\texttt{(Emergency,gid1,pid1), ..., (Notification,gid2,pid2)}\}\end{aligned}$$

3. The last unsubscription data structure concerns the situation where a peer unsubscribes to events raised in a given group G. This peer (services are not allowed) can still receive events from other groups (see the second, third and fourth subscription type).

$$\texttt{UnsubscriptionAllEventsInAGroup = GroupId} \times \texttt{PeerId}$$

   In this example set, the first and the second peer don't want to receive any events from the first and second group:

$$\begin{aligned}\texttt{UnsubscriptionAllEventsInAGroup} &= \{\texttt{gid1,gid2}\} \times \{\texttt{pid1,pid2}\} \\ &= \{\texttt{(gid1,pid1),(gid1,pid2),(gid2,pid1),(gid2,pid2)}\}\end{aligned}$$

4. The last unsubscription type doesn't require any data structure. Indeed, it is the most restrictive unsubscription so, unsubscribing to events implies that there is no more subscription or other unsubscription made by the unsubscriber. For example, if a peer, `pid1` unsubscribed to all kinds of events, no subscription or unsubscription associated with `pid1` can still exist; all of its previous subscriptions are simply removed from the data structure.

The following `UnsubscriptionsState` sample set is close to the `SubscriptionsState` set:

```
UnsubscriptionsState = 𝒫 ({(Emergency,pid1), ..., (Notification,psid2)}) ×
                       𝒫({(Emergency,gid1,pid1), ..., (Notification,gid2,pid2)}) ×
                       𝒫 ({(gid1,pid1), (gid1,pid2), (gid2,pid1), (gid2,pid2)})
                     = {
                         ({},{},{}), ...,
                         (
                           {(Emergency,pid1)},
                           {(Emergency,gid1,pid1)},
                           {(gid1,pid1)}
                         ), ...,
                         (
                           {(Emergency,pid1), ..., (Notification,pid2)},
                           {(Emergency,gid1,pid1), ..., (Notification,gid2,pid2)},
                           {(gid1,pid1), (gid1,pid2), (gid2,pid1), (gid2,pid2)}
                         )
                       }
```

In the rest of this example, we only consider these values:

```
UnsubscriptionsState = {
                         ({},{},{}),
                         ({}, {(Emergency,gid1,pid1)}, {(gid2,pid2)})
                       }
```

In the first state, no entity has unsubscribed. The second state contains two unsubscriptions which have been made by the first (`pid1`) and second peer (`pid2`).

Event Sending Definition

Once we have defined event subscriptions, we can model event sending. Each sent event has a *name*, an *input* (the payload) and can be send to a specific *group* or to all groups in which the caller belongs. In order to know the sender, we need a *sender identifier*. A event sender can be a peer identified by a `PeerId`, or a service instance which is identified either by a `PeerServiceId`, or by a `SessionId`. So, an event sender identifier belongs to `CallerId` (reminder: $\texttt{CallerId} = \texttt{PeerId} \cup \texttt{PeerServiceId} \cup \texttt{SessionId}$).

$$\texttt{SentEvent = EventName} \times \texttt{Input} \times (\mathcal{P}\texttt{(GroupId) - }\emptyset) \times \texttt{CallerId}$$

where:

- $(\mathcal{P}\texttt{(GroupId) - }\emptyset) = \{p \mid (p \subseteq \texttt{GroupId}) \wedge (p \neq \emptyset)\}$ this powerset contains a set of non-empty groups identifier i.e., each element of this set is a non-empty set of group identifiers.

In the previous examples, we have considered:

```
EventName = {Emergency, Notification}
GroupId = {gid1, gid2}
PeerId = {pid1,pid2}
PeerServiceId = {psid1, psid2}
SessionId = {sid1, sid2}
```

The input set has not been defined yet. `CallerId` which is defined as the union of these three latter sets, contains these elements:

```
CallerId = {sid1, sid2, pid1, pid2, psid1, psid2}
Input = {input1, input2}
```

According to these values, we get:

```
SentEvent = {Emergency,Notification} × {input1,input2}
            × (P({gid1,gid2}) - ∅)
            × {sid1,sid2,pid1,pid2,psid1,psid2}

          = {Emergency,Notification} × {input1,input2}
            × {{gid1},{gid2},{gid1,gid2}}
            × {sid1,sid2,pid1,pid2,psid1,psid2}

          = { (Emergency,input1,{gid1},sid1), ...,
              (Notification,input2,{gid1,gid2},psid2) }
```

The first sent event of this sample set, an emergency event, has been raised by the session of a state-full session-full service identified by `sid1` to a group identified by `gid1`. This event contains a payload `input1`. The last event of this set is a notification propagated in two groups identified by `gid1` and `gid2`. In this case, the sender is a service instance, `psid2` and the event payload is `input2`. Of course, these values are not realistic, the session identified by `sid1` has raised an event in a group where its associated service is not published. The second event is also not possible. A state-full session-full cannot be identified by its peer service identifier, it has to be identified by its session identifier. We restrict this set to:

```
SentEvent = { (Emergency,input1,{gid2},sid1),
              (Notification,input2,{gid1,gid2},pid2) }
```

In this sample set, the session identified by `sid1` has raised an emergency event in the group `gid2`. Its payload is `input1`. A peer, `pid2` has raised a notification event with an input `input2` in two groups: `gid1` and `gid2`.

The SMEPP middleware conceives an event as a message with a brief lifetime. So, events don't remain in the network, so they are directly removed as soon as they are raised. Therefore, entities must listen when events are raised. This implies that an event can be received by an entity only if it is listening to this event. Our SMEPP service model considers that events raised between a short period of time can be conceived a set containing the current sent events. The following set is defined in order to model all current sent events. We consider that the middleware is in charge of deleting sent events from the current sent events set after a short period of time.

$$\texttt{SentEventsState} = \mathcal{P}(\texttt{SentEvent})$$

where:

- $\mathcal{P}(\texttt{SentEvent}) = \{e \mid e \subseteq \texttt{SentEvent}\}$ each element of this set is a set of sent events.

According to the previous example, we can get the following sample set:

```
SentEventsState = P ( { (Emergency,input1,{gid2},sid1), ...,
                        (Notification,input2,{gid1,gid2},pid2) } )
                = {
                    {},{(Emergency,input1,{gid2},sid1)},
                    {(Notification,input2,{gid1,gid2},pid2)},
                    { (Emergency,input1,{gid2},sid1),
                      (Notification,input2,{gid1,gid2},pid2) }
                  }
```

As we can see, four states are described. In the first state, there is currently no sent event. In the second and third state, only one event has just been raised (an emergency event and a notification). In the last state, these both events have been raised at the same moment.

Operations Invocations

Finally, the last interaction way in a SMEPP network is operation invocations and their replies. As stated before, the operation invocation is a flexible process. The service which provides an operation can be identified thanks to its *peer service identifier*, or thanks to its *group service identifier*, or finally (in the case of a state-full session-full service) thanks to its *service session identifier*. All of these identifiers belong to `ServiceTypeId`. In order to know who the invoker is, each invocation is associated with an invoker identifier which belongs to `CallerId`. An invocation concerns an operation which has a unique *name* among other service operations. An *input* (which can be empty) is given to the operation. Once a service receives an invocation, it must notify to other services and to the invoker that it has received the invocation and that it starts processing the invocation. An operation receiver is identified by its `ServiceId` (union of `PeerServiceId` and `SessionId`). `empty` is used if an invocation is not received yet. We can define all the possible invocations as:

```
Invocation = CallerId × ServiceTypeId × (ServiceId ∪ {empty}) ×
             OperationName × Input
```

In the following example, we keep the previous samples:

```
CallerId ={sid1, sid2, pid1, pid2, psid1, psid2}
Input ={input1, input2}
OperationName ={"get_filepart", "get_files"}
ServiceId = {"sid1,sid2,psid1,psid2}
```

The `ServiceTypeId` set has not been defined in the previous examples yet, but this set is a union of well known sets:

```
ServiceTypeId = PeerServiceId ∪ GroupServiceId ∪ SessionId
              = {psid1,psid2} ∪ {gsid1,gsid2} ∪ {sid1,sid2}
              = {psid1,psid2,gsid1,gsid2,sid1,sid2}
```

The invocations sets becomes:

```
Invocation = {sid1,sid2,pid1,pid2,psid1,psid2} × {psid1,psid2,gsid1,gsid2,sid1,sid2}
             × {psid1,psid2,sid1,sid2,empty} × {"get_filepart","get_files"}
             ×{input1,input2}
           = { (sid1,psid1,psid1,"get_filepart",input1),...,
               (psid2,sid2,empty,"get_files",input2)}
```

Among the values of this set, we can find a state-full session-full service which has a session identified by `sid1`. This service session has invoked an operation called `"get_filepart"` of a service instance identified by `psid1`. The invocation parameter is `input1`. The message has been received by this service instance. At the end of this example, a service instance identified by `psid2` has invoked the operation `"get_files"` of a state-full session-full service. The concerned session is `sid2`. The

invoker has provided the input `input2`. The service session `sid2` has not received the invocation yet. The second displayed invocation is not correct. Indeed, `psid2` is the peer service identifier of a state-full session-full service. In that case, the caller identifier must be a session identifier and not a peer service identifier. In the following set, only some realistic invocations are considered. The first invocation is the same, but the second invocation is different. This invocation concerns an invocation of the first peer which calls its own service identified by `psid1`, the caller has not received the message yet.

```
Invocation = { (sid1,psid1,psid1,"get_filepart",input1),
               (pid1,psid1,empty,"get_files",input2) }
```

The above set contains all the possible invocations states. At any time, the current invocations states of a SMEPP network is contained in this set. Note that, once the invoker has the invocation result, the invocation is removed from the current invocations state. This state set is defined as:

$$\texttt{InvocationsState = } \mathcal{P}\texttt{(Invocation)}$$

where:

- $\mathcal{P}\texttt{(Invocation)} = \{im \mid im \subseteq \texttt{Invocation}\}$ each element of this set is a set of invocations.

A sample set containing all the possible invocations states of a SMEPP network can be:

```
InvocationsState = P({ (sid1,psid1,psid1,"get_filepart",input1),
                       (pid1,psid1,empty,"get_files",input2)})
                 = {
                     {},
                     {(sid1,psid1,psid1,"get_filepart",input1)},
                     {(pid1,psid1,empty,"get_files",input2)},
                     { (sid1,psid1,psid1,"get_filepart",input1),
                       (pid1,psid1,empty,"get_files",input2)}
                   }
```

The first state corresponds to a SMEPP network where there is currently no invocation. The reason for that situation is either that there hasn't been any invocation yet, or because all invokers have got their results. In the second and third states, there is currently only one invocation (see above the description of these invocations). In the last state, two invocations are still present in the SMEPP network. The first invocation has been made by `sid1`. The second invocation has been made by peer `pid1`.

Operations Replies

Only request-response operations can return a result (possibly an error). Once a request-response operation has finished its execution, it must reply to the invoker by exchanging a message where the caller becomes the receiver and the service provider, the sender. An operation invocation can lead to a *normal* termination, or to an *abnormal* termination.

In the first case, the callee sends a message containing the invoked *operation name* and the *result* (belonging to `Output`) to the caller:

$$\texttt{NormalResultMessage = ServiceId} \times \texttt{CallerId} \times \texttt{OperationName} \times \texttt{Output}$$

A sample set can be:

```
NormalResultMessage = ServiceId × CallerId × OperationName × Output
                    = {psid1,psid2,sid1,sid2}
                      × {sid1,sid2,pid1,pid2,psid1,psid2}
                      × {''get_filepart'',''get_files''}
                      × {output1,output2}
                    = {
                        (psid1,sid1,''get_filepart'',output1), ...,
                        (sid2,psid2,''get_files'',output2)
                      }
```

The first reply of this set has been made by the peer service instance `psid1` to the invoker `sid1` which is a session of a state-full session-full service. The result of the invoked operation `''get_filepart''` is `output1`. At the end of this set, the service session `sid2` provides the result `output2` to the peer service instance `psid2`. The invoked operation was `''get_files''`.

If the operation throws a fault, the exchanged message between the callee and the caller contains an associated *fault name* and an *output data* explaining the thrown fault:

```
ExceptionResultMessage = ServiceId × CallerId × OperationName × Output × FaultName
```

The following values can be exception results:

```
ExceptionResultMessage = ServiceTypeId × CallerId × OperationName ×
                         Output × FaultName
                       = {psid1,psid2,sid1,sid2}
                         × {sid1,sid2,pid1,pid2,psid1,psid2}
                         × {``get_filepart'',''get_files''}
                         × {output1,output2} × {fault1,fault2}
                       = {
                           (psid1,sid1,''get_filepart'',output1,fault1), ...,
                           (sid2,psid2,''get_files'',output2,fault2)
                         }
```

The first abnormal reply is for the invoker `sid1` which is a session of a state-full session-full service. Through this reply, the peer service instance `psid1` notifies that a fault `fault1` has occurred during the processing of the operation `''get_filepart''`. More information about this failure can be found in `output1`. At the end of this set, the service session `sid2` replies to the peer service instance `psid2`. The invoked operation `''get_files''` returned an abnormal result `output2` associated with the fault `fault2`.

Now, we can combine both sets and get a set containing states including current normal and abnormal results:

$$\texttt{ReplyMessagesState} = \mathcal{P}(\texttt{NormalResultMessage}) \times \mathcal{P}\ (\texttt{ExceptionResultMessage})$$

By combining the above sample sets, we get:

```
ReplyMessagesState = 𝒫(NormalResultMessage) × 𝒫 (ExceptionResultMessage)
                   = 𝒫({(psid1,sid1,''get_filepart'',output1), ..., }) ×
                     𝒫({ (psid1,sid1,''get_filepart'',output1,fault1), ..., })
                   = { {},
                       {(psid1,sid1,''get_filepart'',output1)},
                       {(sid2,psid2,''get_files'',output2)}, ...,
                       { (psid1,sid1,''get_filepart'',output1), ...,
                         (sid2,psid2,''get_files'',output2) } } ×
                       { {},
                       {(psid1,sid1,''get_filepart'',output1,fault1)},
                       {(sid2,psid2,''get_files'',output2,fault2)}, ...,
                       { (psid1,sid1,''get_filepart'',output1,fault1), ...,
                         (sid2,psid2,''get_files'',output2,fault2) }
                       }
                   = { ({},{}), ...,
                       (
                         { (psid1,sid1,''get_filepart'',output1), ...,
                           (sid2,psid2,''get_files'',output2) },
                         { (psid1,sid1,''get_filepart'',output1,fault1), ...,
                           (sid2,psid2,''get_files'',output2,fault2) }
                       )
                     }
```

The first state of this set represents a SMEPP network where there is currently no reply. The last state describes several replies including the four replies (two normal and two abnormal replies) described above. Of course, this sample set is not possible. Indeed, SMEPP doesn't allow to process concurrent request i.e., the same invokee process at the same time several invocations from the same entity. Moreover, in our example, the service identified by `psid2` is a state-full session full service thus it cannot be invoked thanks to its peer service identifier. Finally, it is not possible to reply several times to the same invocation, there are either `(psid1,sid1,''get_filepart'',output1)`, or `(psid1,sid1,''get_filepart'',output1,fault1)` and either `(sid2,psid2,''get_files'',output2)`, or `(sid2,psid2,''get_files'',output2,fault2)`. The following sample set is a realistic set. Three states are present, in the first state, the result of the only invocation is not present yet. In the second, the only invocation leads to a normal result, but not in the last one, where `fault1` has been thrown:

```
ReplyMessagesState = {
                       ({},{}), ...,
                       ( {(psid1,sid1,''get_filepart'',output1)}, {} )
                       ( {}, {(psid1,sid1,''get_filepart'',output1,fault1)} )
                     }
```

SMEPP Network Definition

Once we have all these basic concepts, we can define the most abstract concept: *a SMEPP network state*. This data structure models all possible states which the *current* SMEPP networks must belong to. A network state is associated with *credentials* which allow to accept or reject peer in the network. A network is also a set of peers, group, messages exchange (invocations & replies) and events.

```
NetworkState = Credentials × ConnectedPeers × CurrentGroups × SubscriptionsState
             × SentEventsState × InvocationsState × ReplyMessagesState
```

The following set contains possible network states of the SequiTel application. This set is a combination of the previous examples.

By combining these values, we get:

```
NetworkState = Credentials × ConnectedPeers × CurrentGroups × SubscriptionsState
               × SentEventsState × InvocationsState × ReplyMessagesState
             = {
                 (cred1, {}, {}, ({},{},{},{}),
                 ({},{},{}), {}, {}, ({},{}) ), ...,
                 (
                   cred2, { (pid1,{service1}), (pid2,{service2}) },
                   {(gid1,cred1,descr1,{pid1,pid2}), (gid2,cred2,descr2,{pid2})} }
                   ({},{(Emergency,pid1)},{},{pid2}),
                   ({}, {(Emergency,gid1,pid1)}, {(gid2,pid2)} ),
                   {
                     (Emergency,input1,{gid2},sid1),
                     (Notification,input2,{gid1,gid2},pid2)
                   },
                   { (sid1,psid1,psid1,"get_filepart",input1),
                     (pid1,psid1,empty,"get_files",input2) },
                   ({}, {(psid1,sid1,''get_filepart'',output1,fault1)} )
                 )
               }
```

The first state of this set, `(cred1, {}, {}, ({},{},{},{}), ({},{},{}), {}, {}, ({}) )`, is an empty network associated with credentials `cred1`. There is no peer so, no group, etc. This example is not permitted because, the modeled SMEPP machine is a peer so, there is always at least one peer, the peer associated with the B-machine. Only the state described below will be considered.

The second displayed state is more interesting. Two peers are currently connected. The first peer is identified by `pid1` and has published the `service1` in the first group identified by `gid1`. The second peer, `pid2`, has published the second service `service2` in the second group identified by `gid2`. Two groups identified by`gid1` and `gid2` are present. The first group contains these two peers. The second group identified by `gid2` only contains the second peer, `pid2`.
Both peers have subscribed to events. The first peer has subscribed to emergency events expected in the first group, `gid1`) (see the first unsubscription). The second peer wants to receive all kind of events excepting events coming from the second group (see the second unsubscription).

The session identified by `sid1` has raised an emergency event (its payload: `input1`) in the group `gid2`. Unfortunately, nobody can receive this event. The second event has been raised by the peer `pid2` in two groups: `gid1` and `gid2`. This notification event is associated with an input `input2`. Only the second peer is able to receive this notification coming from the first group.

Two invocations have been made, but their results have not been retrieved yet. The first invocation have been made by the session identified by `sid1` associated with `service2` published by the peer `pid2`. The result of this invocation is a fault called `fault1` associated with `output1`. The last e invocation is a rather special invocation; the first peer, `pid1` has invoked its own service identified by `psid1`, but it has not yet retrieved the invocation.

Variables Definition

Up to now all of these data structures have been defined as sets. But, now, we need to define the value of the current network state which belongs to the SMEPP network state set. We can create two variables: `network` which belongs to `NetworkState` and `currentPeer` member of `PeerId`. The first variable represents the current network state. The second variable contains the current peer identifier associated with the B-machine which is a peer among other peers of the SMEPP network described by the variable `network`.

3.4.2 Direct Access to Structures

All of these data structures definitions are highly conceptual, based on sets, powersets and cartesian products. So, these definitions are independent from the B-language. Once we implement them in a B-machine, we can use “records”[Cle07, p. 61-62] structures. Records structures are a technical mechanism provided by the B-language, used to facilitate the manipulation of sets based on cartesian products. Indeed, records allow to directly access to an element of a n-uplet, avoiding complex manipulation and data typing.

Three operators associated with records are defined:

Operator	Definition	Syntax
struct	Set of records	”struct” ”(” (Ident ”:” Expression)$^{+”,”}$ ”)”
rec	Record in extension	”rec” ”(” ([Ident ”:”] Expression)$^{+”,”}$ ”)”
’	Access to a record field (quote operator)	Expression ”′”

In the expression $struct(Ident_1 : E_1, \ldots, Ident_n : E_n)$, E_i must be of the type $\mathcal{P}(T_i)$. Then, the type of this expression is $\mathcal{P}(struct(Ident_1 : E_1, \ldots, Ident_n : E_n))$.

The first and the second operator are linked through this definition:

$$
\begin{aligned}
struct(Ident_1 : E_1, \ldots, Ident_n : E_n) \quad = \quad & \{ \\
& \quad rec\ (Ident1 : x_1, \ldots, Ident_n : x_n) \mid \\
& \quad \forall i,\ 1 \le i \le n,\ x_i \text{ satisfies } E_i \\
& \}
\end{aligned}
$$

where E_i must be of the type $\mathcal{P}(T_i)$. If the record has been type previously, or if its context defines its type, we can rewrite it to: $myRec = rec\ (x_1, \ldots, x_n)$ instead of $myRec = rec\ (Ident1 : x_1, \ldots, Ident_n : x_n)$.

Service Contract Example

In this example, we will see how it is possible to redefine a service contract. All cartesian products must be redefined as records. In a contract, operations are also defined as a cartesian product so, we must redefine them before contracts. An operation has a name, the first field of this 4-uplet. The second field defines the operation type. The third and the last field define argument types and operation result types.

```
Operation =  struct ( name :  OperationName, type :  Toperation,
                      argTypes : P(AbstractDataType),
                      resultTypes : P(AbstractDataType) )
```

`Scontract` is defined as a set of service contracts. A contract 4-uplet, the type of its first field, called `info`, is `serviceInfo`. The second field, `type`, is the service type, `Tservice`. The third and the last field of a contract is a little bit more complex, `operations` is a *set* of `Operation`. `eventsRaised` is a *set* of `EventName`.

```
Scontract  =  struct ( info :  ServiceInfo, type :  Tservice,
                       operations :SEQ(Operation), eventsRaised :  SEQ(EventName) )
```

For example, a possible value of this set above can be:

```
contract1  =  rec(info :  info1, type :  sessionFull,
                 operations : {
                    rec(name :  "get_files", type :  requestResponse,
                       argTypes :  (), resultTypes :  (byte[])),
                    rec(name :  "get_file_part", type :  requestResponse,
                       argTypes :  (String,Int), resultTypes :  (byte[]))
                 },
                 eventsRaised : {Emergency, Notification} )
```

If we have already constrained `contract1` type, `contract1` ∈ `Scontract`, we can rewrite it:

```
contract1   =   rec(info1, sessionFull,
                    {
                       rec("get_files", requestResponse, (), (byte[])),
                       rec("get_file_part", requestResponse, (String,Int), (byte[]))
                    },
                    {Emergency, Notification} )
```

Service Redefinition

We begin by redefining services. Sets which are defined as cartesian products, will be redefined. Each element of a cartesian product simply receives a name and the redefined set is now defined as a structure. The three following sets are concerned by these changes. Values of `argsType` and `resultTypes` fields make a list of abstract data types. In these definitions, values of `operations`, `eventsRaised` and `sessions` are a set of values.

```
Operation   =   struct ( name : OperationName, type : Toperation,
                         argTypes : SEQ(AbstractDataType),
                         resultTypes : SEQ(AbstractDataType) )

Scontract   =   struct ( info : ServiceInfo, type : Tservice,
                         operations : 𝒫(Operation),
                         eventsRaised : 𝒫(EventName) )

Service     =   struct ( gid : GroupId, contract : Scontract,
                         grounding : Sgrounding,
                         psid : PeerServiceId, gsid : GroupServiceId,
                         sessions : 𝒫(SessionId) )
```

Peer Redefinition

Once services are defined as structure, we can redefines peers. The second field of a peer called `service` is a set of services. Note that, the second set is *not* redefined.

```
Peer            =   struct ( pid : PeerId, services : 𝒫(Services) )
ConnectedPeer   =   𝒫(Peer)
```

Group Redefinition

A group is a couple composed of a group identifier and a set of peer identifiers. Once again, the second set hasn't changed.

```
Group           =   struct ( gid : GroupId, cred : Credentials,
                             gdescr : Gdescr, members : 𝒫(PeerId) )
CurrentGroups   =   𝒫(Group)
```

Events Sending Redefinition

A sent event is a 4-uplet: an event name, a payload, recipients and a sender identifier. `SentEventsState` is not redefined.

```
SentEvent         =   struct ( name : EventName,
                               in : Input, groups : ( 𝒫(GroupId) - ∅),
                               sender : CallerId )
SentEventsState   =   𝒫(SentEvent)
```

Subscriptions Redefinition

All of the following sets have been changed. `SubscriptionsState` is a little bit more complex, all of its four fields are a set of subscriptions.

```
SubscriptionEventInAGroup        = struct( event:EventName, group :GroupId,
                                     subscriber:PeerId )
SubscriptionEventAllGroups       = struct( event:EventName,
                                     subscriber:(PeerId ∪ PeerServiceId) )
SubscriptionAllEventsInAGroup    = struct( group:GroupId, subscriber:PeerId )
SubscriptionAllEventsAllGroups   = struct( subscriber:  (PeerId ∪ PeerServiceId) )
SubscriptionsState               = struct( firstType:𝒫(SubscriptionEventInAGroup),
                                     secondType:𝒫(SubscriptionEventAllGroups),
                                     thirdType:𝒫(SubscriptionAllEventsInAGroup),
                                     fourthType:𝒫(SubscriptionAllEventsAllGroups) )
```

Unsubscriptions Redefinition

The four next sets define unsubscription sets. The last one is a set of triplets where each field of a triplet is a set of unsubscriptions.

```
UnubscriptionEvent                 =  struct ( event :  EventName,
                                        unsubscriber :  (PeerId ∪ PeerServiceId) )
UnsubscriptionEventInAGroup        =  struct ( event :  EventName, group :  GroupId,
                                        unsubscriber :  PeerId )
UnsubscriptionAllEventsInAGroup    =  struct ( group :  GroupId, unsubscriber :  PeerId )
UnsubscriptionsState               =  struct ( firstType : 𝒫(UnubscriptionEvent),
                                        secondType : 𝒫(UnsubscriptionEventInAGroup),
                                        thirdType : 𝒫(UnsubscriptionAllEventsInAGroup) )
```

Messages Exchange Redefinition

In the following sets, the second set has not changed. The last one merges the third and fourth set. Indeed each element of `ReplyMessagesState` is a couple where the first couple element contains a set of normal replies and the second, a set of abnormal replies.

```
Invocation             =  struct ( caller :  CallerId,
                                   callee :  ServiceTypeId, provider :  ServiceId
                                   op :  OperationName, in :  Input )
InvocationsState       =  𝒫(Invocation)
NormalResultMessage    =  struct ( callee :  ServiceId,
                                   caller :  CallerId, op :  OperationName,
                                   out :  Output )
ExceptionResultMessage =  struct ( callee :  ServiceId,
                                   caller :  CallerId, op :  OperationName,
                                   out :  Output, fault :  FaultName )
ReplyMessagesState     =  struct ( normalResults : 𝒫(NormalResultMessage),
                                   missResults : 𝒫(ExceptionResultMessage) )
```

Network State Redefinition

We will finish by redefining `NetworkState`. Now, elements of a network state can be accessed thanks to the following fields. For example, if we want to access to one of the current peers, we use the `peers` field of the concerned network state,.

```
NetworkState  =  struct ( cred :  Credentials,
                          peers :  ConnectedPeers,
                          groups :  CurrentGroups,
                          subscriptions :  SubscriptionsState,
                          unsubscriptions :  UnsubscriptionsState,
                          sentEvents :  SentEventsState,
                          invocations :  InvocationsState,
                          replies :  ReplyMessagesState )
```

Thanks to these redefinitions, the following constraints become more readable and easier to write.

3.4.3 Data Structures Constraints

In the previous sub-section, we have defined the SMEPP network states, but among all these states, some of them are not **valid**. We need to define constraints which restrict these states to a subset of valid states. Each of the following constraints is identified by a name. For example, the first constraint is identified by *CS01*. These identifiers are also placed in the file containing the B-model. We begin by services constraints followed by peers constraints, groups constraints and finally communication constraints (events and operations calls).

Services Constraints

CS01. For each identifier, we have to create a constraint which restricts its usage. Indeed, only one element can be associated with an identifier value (except for `GroupServiceIdentifier` elements). We begin by defining the constraint restricting the peer service identifier. The `psid` field corresponds to the fourth element of a service n-uplet. Its domain value is `PeerServiceId`. This constraint means that two services belonging to the same SMEPP network are not identified by the same peer service identifiers.

```
∀ networkState ∈ NetworkState,
∀ peer1 ∈ NetworkState'peers, ∀ peer2 ∈ NetworkState'peers,
∀ service1 ∈ peer1'services, ∀ service2 ∈ peer2'services,
(service1 ≠ service2) ⇒ (service1'psid ≠ service2'psid)
```

Figure 3.4: Constraint CS01

CS02. A service can create sessions if-and-only-if this service is state-full session-full. So, we need a constraint that restricts the session set associated with each service n-uplet. This set must be empty if the service is not a state-full session-full service, otherwise its value is not restricted (can be empty or not):

```
∀ networkState ∈ NetworkState,
∀ peer1 ∈ NetworkState'peers,
∀ service ∈ peer1'services,
(service'contract'type ≠ sessionFull) ⇒ (service'sessions = ∅)
```

Figure 3.5: Constraint CS02

CS03. In the following definition, usage of session identifiers is restricted: two sessions have a unique identifier among all sessions of a SMEPP network.

```
∀ networkState ∈ NetworkState,
∀ peer1 ∈ networkState'peers, ∀ peer2 ∈ networkState'peers,
∀ service1 ∈ peer1'services, ∀ service2 ∈ peer2'services,
∀ session1 ∈ service1'sessions, ∀ session2 ∈ service2'sessions,
(service1 ≠ service2) ⇒ (session1 ≠ session2)
```

Figure 3.6: Constraint CS03

CS04. The last constraint related to service identifiers is the restriction of the service group identifier. Two published services have the same *service group identifier* if-and-only-if, they have the same *contract* and they belong to the same group; otherwise they have different service group identifiers. So, this identifier allows to blindly retrieve service instances of the same service in a given group. This constraint is:

```
∀ networkState ∈ NetworkState,
∀ peer1 ∈ networkState'peers, ∀ peer2 ∈ networkState'peers,
∀ service1 ∈ peer1'services, ∀ service2 ∈ peer2'services,
(service1'gsid = service2'gsid) ⇔
( (service1'contract = service2'contract) ∧ (service1'gid = service2'gid) )
```

Figure 3.7: Constraint CS04

CS05. The following constraint prohibits that two operations provided by a service have the same name. In other words, each operation of a service has a unique name among other operations of this service.

```
∀ networkState ∈ NetworkState, ∀ peer ∈ networkState'peers,
∀ service ∈ peer'services,
∀ operation1 ∈ service'contract'operations,
∀ operation2 ∈ service'contract'operations,
(operation1 ≠ operation2) ⇒ (operation1'name ≠ operation2'name)
```

Figure 3.8: Constraint CS05

CS06. Each published service can be called by other members of the group where it is published. But a service cannot be published inside a group where the peer which provides this service, is not a member. Therefore, we have to check that for each service, a group exists and contains the provider of this service.

```
∀ networkState ∈ NetworkState, ∀ peer ∈ networkState'peers, ∀ service ∈ peer'services ⇒
∃ group ∈ networkState'groups, ( (group'gid = service'gid) ∧
                                 (peer'pid ∈ group'members) )
```

Figure 3.9: Constraint CS06

CS07. Republishing the same service i.e., the same service contract, in the same group is the same as unpublishing and re-publishing the service. So, the same peer has maximum one service instance in a group:

```
∀ networkState ∈ NetworkState, ∀ peer ∈ networkState'peers, ∀ service ∈ peer'services ⇒
∄(service2).( (service2 ∈ peer'services) ∧ (service'gid = service2'gid) ∧
             (service'contract = service2'contract) ∧ (service'psid ≠ service2'psid) )
```

Figure 3.10: Constraint CS07

Peers Constraints

The sole constraint of the peer set concerns their identifiers. All peers have different peer identifiers which allow to recognize a peer among other peers. The following constraint can be summarized as "for each couple of peers, these peers have a different identifier belonging to `PeerId`"; it allows to have unique identifier:

```
∀ networkState ∈ NetworkState,
∀ peer1 ∈ networkState'peers, ∀ peer2 ∈ networkState'peers,
(peer1 ≠ peer2) ⇒ (peer1'pid ≠ peer2'pid)
```

Figure 3.11: Constraint CP01

Groups Constraints

CG01. Firstly, we will start by defining the constraint which allows to have a unique group identifier:

```
∀ networkState ∈ NetworkState,
∀ group1 ∈ networkState'groups, ∀ group2 ∈ networkState'groups,
(group1 ≠ group2) ⇒ (group1'pid ≠ group2'pid)
```

Figure 3.12: Constraint CG01

CG02. A group n-uplet has a component called `members` which contains identifiers of the peers which are members of this group. Of course, these identifiers must refer to existing peers. The constraint CG02 means that for each group in a network state, each peer identifier contained in its members set must be linked to a peer n-uplet (i.e. a connected peer) contained in the peers set of the network state.

```
∀ networkState ∈ NetworkState,
∀ group ∈ networkState'groups, ∀ peerId ∈ groups'members,
∃ peer ( (peer ∈ networkState'peers) ∧ (peerId = peer'pid) )
```

Figure 3.13: Constraint CG02

CG03. The last group constraint is related to its lifetime. A group exists until it has at least one member. Once, its last member leaves a group, this group is over. So, all groups have at least one member:

```
∀ networkState ∈ NetworkState, ∀ group ∈ networkState'groups,
#(group'members) ≥ 1
```

Figure 3.14: Constraint CG03

Events Subscriptions Constraints

Previously, we have divided the model of events subscriptions into four parts:

1. a peer identified by `peerId` subscribes to a given event `E` raised inside a group `G`, denoted by `sub1(E,G,peerId)`,
2. a peer or service identified by `callerId` subscribes to an event called `E`, no matter the group in which it is raised: `sub2(E,callerId)`,
3. a subscription made by a peer `peerId` to events (no event name constraint) raised inside a given group identified by `G`: `sub3(G,peerId)`,
4. a peer or service identified by `callerId` subscribes to any kind of events (no event name and no group constraint): `sub4(callerId)`.

Unsubscriptions notations: `usub1(E,caller)`, `usub2(E,G,caller)` and `usub3(G,caller)` will be introduced in the next part related to unsubscriptions constraints. In order to understand how the data structure can evolve, Table B.1 (page 158) and Table B.2 (page 163) list representative scenarios. Each row represents a possible scenario. The second column represents the current subscriptions and unsubscriptions state. Then, the `Action` column contains the (un)subscriptions which will be executed on the current state. The result of this execution is displayed on the third column. Elements have the following syntax:

- ... is used to unconstraint the current state,
- $\emptyset$ there is no current (un)subscription,
- `SKIP` no modification on the current state,
- $\cup\{X\}$ the (un)subscription `X` is added to the current state,
- $-\{X\}$ the (un)subscription `X` is removed from the current state,
- * means that there is no constraint on the argument,
- `x + y` sequential execution, `x` is executed followed by `Y`.

For example, the first line of Table B.1 (page 158) means that an entity which subscribes to all kinds of events implies that its previous subscriptions and unsubscriptions are removed. These four subscription types have lead to four different data types i.e., four different n-uplet structures. Each of the following constraints is related to one of these structures. The first constraint called `CSUB01` is divided into 8 parts.

```
∀ networkState ∈ NetworkState, ∀ subscription ∈ networkState'subscriptions'firstType,
(
  ∃(group).( (group ∈ networkState'groups) ∧ (subscription'subscriber ∈ group'members) ∧
             (group'gid = subscription'group) ) ∧
  /* Constraint CSUB01 - part 1 */
  /* Constraint CSUB01 - part 2 */
  /* Constraint CSUB01 - part 3 */
  /* Constraint CSUB01 - part 4 */
  /* Constraint CSUB01 - part 5 */
  /* Constraint CSUB01 - part 6 */
  /* Constraint CSUB01 - part 7 */
  /* Constraint CSUB01 - part 8 */
)
```

Figure 3.15: Constraint CSUB01

CSUB01 - Part 1. If a peer has initially subscribed to an event called E (second subscription type) and then, subscribed to an event called E in a given group G (first subscription type) then between these subscriptions, it must have unsubscribed to events raised in G (third unsubscription type). This is translated in the following constraint.

```
( /* sub1(E,G,caller) ∧ sub2(E,caller) ⇒ usub3(G,caller) */
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'secondType) ∧
                     (subscription2'subscriber = subscription'subscriber) ∧
                     (subscription2'event = subscription'event)) ⇒
  ∃(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'thirdType) ∧
                     (unsubscription'unsubscriber = subscription'subscriber) ∧
                     (unsubscription'group = subscription'group))
) ∧
```

Figure 3.16: Constraint CSUB01 - Part 1

CSUB01 - Part 2. In the same way, if a peer has initially subscribed to events raised in a group identified by G (third subscription type) and then, subscribed to an event called E raised in G (first subscription type) then between these subscriptions, it must have unsubscribed to the event E (first unsubscription type).

```
( /* sub1(E,G,caller) ∧ sub3(G,caller) ⇒ usub1(E,caller) */
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'thirdType) ∧
                     (subscription2'subscriber = subscription'subscriber) ∧
                     (subscription2'group = subscription'group)) ⇒
  ∃(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'firstType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'event = subscription'event))
) ∧
```

Figure 3.17: Constraint CSUB01 - Part 2

CSUB01 - Part 3. The third part avoids that a specific subscription to a given event E in a given group G, coexists with two more generic subscriptions: a subscription to the event E (second subscription type) and a subscription to events raised in G (third subscription type). This coexistence can never happen.

```
( /* sub1(E,G,caller) ∧ sub2(E,caller) ∧ sub3(G,caller) ⇒ FALSE */
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'firstType) ∧
                     (subscription2'subscriber = subscription'subscriber) ∧
                     (subscription2'event = subscription'event) ) ∨
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'thirdType) ∧
                     (subscription2'subscriber = subscription'subscriber) ∧
                     (subscription2'group = subscription'group) )
) ∧
```

Figure 3.18: Constraint CSUB01 - Part 3

CSUB01 - Part 4. If an entity (a service or a peer) initially subscribed to all kinds of events (fourth subscription type) and after, subscribes to a certain event called E in a specific group G (first subscription type) then it must have unsubscribed to the events E (first unsubscription type) or to events raised in G

(third unsubscription type) before its second subscription.

```
( /* sub1(E,G,caller) ∧ sub4(caller) ⇒ usub1(E,caller) ∨ usub3(G,caller) */
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'fourthType) ∧
                     (subscription2'subscriber = subscription'subscriber) ) ⇒
                     ∃(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'firstType) ∧
                     (unsubscription'unsubscriber = subscription'subscriber) ∧
                     (unsubscription'event = subscription'event)) ∨
  ∃(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'thirdType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'group = subscription'group))
) ∧
```

Figure 3.19: Constraint CSUB01 - Part 4

CSUB01 - Part 5. If a peer has unsubscribed to the event `E` (first unsubscription type) and then subscribed to this event in the group `G` (first subscription type) then it must have initially subscribed to events raised in `G` (third subscription type), or subscribed to all kinds of events (fourth subscription type).

```
( /* sub1(E,G,caller) ∧ usub1(E,caller) ⇒ sub3(G,caller) ∨ sub4(caller) */
  ∃(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'firstType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'event = subscription'event)) ⇒
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'thirdType) ∧
                     (subscription2'subscriber = subscription'subscriber) ∧
                     (subscription2'group = subscription'group)) ∨
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'fourthType) ∧
                     (subscription2'subscriber = subscription'subscriber))
) ∧
```

Figure 3.20: Constraint CSUB01 - Part 5

CSUB01 - Part 6. It is not possible to have a subscription to the event `E` raised in the group identified by `G` (first subscription type) and at the time an unsubscription to `E` in `G` (second unsubscription type).

```
/* sub1(E,G,caller) ∧ usub2(E,G,caller) ⇒ FALSE */
∄(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'secondType) ∧
                    (unsubscription'group = subscription'group) ∧
                    (unsubscription'event = subscription'event) ∧
                    (unsubscription'unsubscriber = subscription'subscriber)
) ∧
```

Figure 3.21: Constraint CSUB01 - Part 6

CSUB01 - Part 7. If a peer unsubscribes to events raised in the group associated with `G` (third unsubscription type) and then subscribes to the event `E` raised in `G` (first subscription type), it must have initially subscribed to all kinds of events (fourth subscription type), or subscribed to the event `E` (second subscription type).

```
( /* sub1(E,G,caller) ∧ usub3(G,caller) ⇒ sub2(E,caller) ∨ sub4(caller) */
  ∃(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'thirdType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'group = subscription'group)) ⇒
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'secondType) ∧
                     (subscription2'subscriber = subscription'subscriber) ∧
                     (subscription2'event = subscription'event)) ∨
  ∃(subscription2).( (subscription2 ∈ networkState'subscriptions'fourthType) ∧
                     (subscription2'subscriber = subscription'subscriber))
) ∧
```

Figure 3.22: Constraint CSUB01 - Part 7

CSUB01 - Part 8. Its not possible to have at the same time an subscription to an event called `E` raised in the group `G` (first subscription type), a subscription to the event `E` (first unsubscription type) and a subscription to events raised in the group `G` (third unsubscription type).

```
( /* sub1(E,G,caller) ∧ usub1(E,caller) ∧ usub3(G,caller) ⇒ FALSE */
  ∄(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'firstType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'event = subscription'event) ) ∨
  ∄(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'thirdType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'group = subscription'group) )
)
```

Figure 3.23: Constraint CSUB01 - Part 8

CSUB02. This constraint concerns subscriptions of the second type i.e., a subscription to a given event (no group constraint). The first constraint part concerns the coexistence of a subscription to a given event `E` (second subscription type) and a subscription to this event `E` in a certain group (first subscription type). "Constraint CSUB01 - part 1" already constrained this co-existence . The next restriction is taken into account in "Constraint CSUB01 - part 3". The third part avoids that the most generic subscription (fourth subscription type) exists at the same time as a subscription of the third type which concerns events raised in a specific group. The last constraint part introduces the prohibition on the coexistence of a subscription of the second type (subscription to the event `E`) and an unsubscription of the first type (unsubscription to the event `E`) which is its counterpart.

CSUB03. The last subscription constraint restricts subscriptions to events raised in a specific group (third subscription type). The first constraint part is related to the coexistence of a subscription of the third type and an associated subscription of the first type: if the peer has initially subscribed to events raised in a group identified by `G` (third subscription type) and after, subscribes to the event `E` raised in that group (first subscription type), it must have unsubscribed to the event `E` before doing the second subscription. The second part avoids that a generic subscription to all kinds of events (fourth subscription) is present at the same time as a subscription to events raised in a specific group (third subscription type). The next part is taken into account in "Constraint CSUB01 - part 3". The last constraint part, avoids that a subscription of the third type and its counterpart (an unsubscription of the third type) exist at the same time.

CSUB04. In the last subscription constraint related to subscription of the fourth type (subscription to all kinds of events), all constraints have been taken into account in the previous constraints:

```
∀ networkState ∈ NetworkState, ∀ subscription ∈ networkState'subscriptions'secondType,
(
  ( /* check subscriber identifier */
    ∃(peer).( (peer ∈ networkState'peers) ∧ (subscription'subscriber = peer'pid) ) ∨
    ∃(peer,service).( (peer ∈ networkState'peers) ∧ (service ∈ peer'services) ∧
                      (subscription'subscriber = service'psid) )
  ) ∧
  /* sub2(E,caller) ∧ sub1(E,G,caller) ⇒ usub3(G,caller) */
    /* already constrained */
  /* sub2(E,caller) ∧ sub3(G,caller) ⇒ TRUE */
    /* not constrained */
  /* sub2(E,caller) ∧ sub4(caller) ⇒ FALSE */
    ∄(subscription2).( (subscription2 ∈ networkState'subscriptions'fourthType) ∧
                       (subscription2'subscriber = subscription'subscriber) ) ∧
  /* sub2(E,caller) ∧ sub1(E,G,caller) ∧ sub3(G,caller) ⇒ FALSE */
    /* already constraint */
  /* sub2(E,caller) ∧ usub1(E,caller) ⇒ FALSE*/
  ∄(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'firstType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'event = subscription'event) ) ∧
  /*sub2(E,caller) ∧ usub2(E,G,caller) ⇒ TRUE */
    /* not constrained */
  /* sub2(E,caller) ∧ usub3(G,caller) ⇒ TRUE */
    /* not constrained */
)
```

Figure 3.24: Constraint CSUB02

```
∀ networkState ∈ NetworkState, ∀ subscription ∈ networkState'subscriptions'thirdType,
(
  ∃(group).( (group ∈ networkState'groups) ∧ (subscription'subscriber ∈ group'members) ∧
             (group'gid = subscription'group) ) ∧
  /*sub3(G,caller) ∧ sub1(E,G,caller) ⇒ usub1(E,caller) */
    /* already constrained */
  /* sub3(G,caller) ∧ sub2(E,caller) ⇒ TRUE */
    /* not constrained */
  /* sub3(G,caller) ∧ sub4(caller) ⇒ FALSE */
  ∄(subscription2).( (subscription2 ∈ networkState'subscriptions'fourthType) ∧
                     (subscription2'subscriber = subscription'subscriber) ) ∧
  /*sub3(G,caller) ∧ sub1(E,G,caller) ∧ sub3(G,caller) ⇒ FALSE */
    /* already constraint */
  /* sub3(G,caller) ∧ usub1(E,caller) ⇒ TRUE */
    /* not constrained */
  /* sub3(G,caller) ∧ usub2(E,G,caller) ⇒ TRUE */
    /* not constrained */
  /* sub3(G,caller) ∧ usub3(G,caller) ⇒ FALSE */
  ∄(unsubscription).( (unsubscription ∈ networkState'unsubscriptions'thirdType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) ∧
                      (unsubscription'group = subscription'group) )
)
```

Figure 3.25: Constraint CSUB03

- the impossibility of having a subscription to all kinds of events and a subscription to a given event (denoted by: `sub2(E,caller)`), or a subscription to events raised in a certain group (i.e., `sub3(G,caller)`),

- having a specific subscription of the first type (denoted by `sub1(E,G,caller)` and the most generic subscription type (subscription of the fourth type, `sub4(caller)`) without having an unsubscription of the first (`sub1(E,caller)`), or of the third type (`sub3(G,caller)`).

```
∀ networkState ∈ NetworkState, ∀ subscription ∈ networkState'subscriptions'fourthType,
(
  ( /* check subscriber identifier */
    ∃(peer).( (peer ∈ networkState'peers) ∧ (subscription'subscriber = peer'pid) ) ∨
    ∃(peer,service).( (peer ∈ networkState'peers) ∧ (service ∈ peer'services) ∧
                      (subscription'subscriber = service'psid) )
  ) ∧
  /* sub4(caller) ∧ sub1(E,G,caller) ⇒ usub1(E,caller) ∨ usub3(G,caller) */
    /* already constrained */
  /* sub4(caller) ∧ sub2(G,caller) ⇒ FALSE */
    /* already constrained */
  /* sub4(caller) ∧ sub2(E,caller) ⇒ FALSE */
    /* already constrained */
  /* sub4(caller) ∧ usub1(E,caller) ⇒ TRUE */
    /* not constrained */
  /* sub4(caller) ∧ usub2(E,G,caller) ⇒ TRUE */
    /* not constrained */
  /* sub4(caller) ∧ usub3(G,caller) ⇒ TRUE */
    /* not constrained */
)
```

Figure 3.26: Constraint CSUB04

Events Unsubscriptions Constraints

Once again, a constraint is needed for each data type. Our model contains four unsubscription types:

1. a peer or a service no longer wants to receive a specific event,
2. a peer unsubscribes to a specific event raised in a given group,
3. a peer no langer wants to receive events from a specific group,
4. a peer or a service unsubscribes to all kinds of events.

The last type doesn't require a data structure, so we only need three constraints, one for each unsubscription type.

CUSUB01. We will begin by constraining the first unsubscription type. Two restrictions are introduced by this constraint. The first, mentioned by: `usub1(E,caller)` ⇒ `sub3(G,caller)` ∨ `sub4(caller)` prohibits the existence of an unsubscription to the event `E` (first unsubscription type) without having a previous subscription to events raised in `G` (third subscription type), or a subscription to all kinds of events (fourth subscription type). The second restriction avoids that a generic unsubscription of the third type exists at the same time as a particular unsubscription of the second type (unsubscription to an event `E` raised in a group `G`). Two other restrictions are also present but they are already included in `CSUB01` and `CSUB02`.

CUSUB02. The coexistence of a second unsubscription type (see `usub2(E,G,caller)`) and a first subscription type (`sub1(E,G,caller)`) is already restricted in `CSUB01`. Other restrictions are introduced by `CUSUB02`. The following restriction has not been taken into account by previous constraints yet. Before

```
∀ networkState ∈ NetworkState, ∀ unsubscription ∈ networkState'unsubscriptions'firstType,
(
  ∃(peer).( (peer ∈ networkState'peers) ∧ (unsubscription'unsubscriber = peer'pid) )
  ∨
  ∃(peer,service).( (peer ∈ networkState'peers) ∧ (service ∈ peer'services) ∧
                    (unsubscription'unsubscriber = service'psid) )
) ∧
  /* usub1(E,caller) ∧ sub1(E,G,caller) ⇒ sub3(G,caller) ∨ sub4(caller) */
     /* already constrained */
  /* usub1(E,caller) ∧ sub2(E,caller) ⇒ FALSE */
     /* already constrained */
  /* usub1(E,caller) ∧ sub3(G,caller) ⇒ TRUE */
     /* not constrained */
  /* usub1(E,caller) ∧ sub4(caller) ⇒ TRUE */
     /* not constrained */
  /* usub1(E,caller) ⇒ sub3(G,caller) ∨ sub4(caller) */
  (
    ∃(subscription).( (subscription ∈ networkState'subscriptions'fourthType) ∧
                      (subscription'subscriber = unsubscription'unsubscriber) ) ∨
    ∃(subscription).( (subscription ∈ networkState'subscriptions'thirdType) ∧
                      (unsubscription'unsubscriber = subscription'subscriber) )
  ) ∧
  /* usub1(E,caller) ∧ usub2(E,G,caller) ⇒ FALSE */
  ∄(unsubscription2).( (unsubscription2 ∈ networkState'unsubscriptions'secondType) ∧
                       (unsubscription2'event = unsubscription'event) ∧
                       (unsubscription2'unsubscriber = unsubscription'unsubscriber)
  )
  /*usub1(E,caller) ∧ usub3(G,caller) ⇒ TRUE */
    /* not constrained */
)
```

Figure 3.27: Constraint CUSUB01

unsubscribing to a given event E raised in G, the peer must subscribe to the event E (second subscription type), or subscribe to events raised in a group G, or subscribe to all kinds of events (fourth subscription type). The last restriction doesn't allow to have at the same time a particular unsubscription to the event E coming from G (second unsubscription type) and a more generic unsubscription of the third type (unsubscription to events coming from G).

CUSUB03. Only one restriction on unsubscriptions of the third type has not been written yet. This restriction concerns the existence of such unsubscriptions. A subscription of the third type where an entity unsubscribes to events raised in a given group G implies that it has previously subscribed to a certain event E (second subscription type), or subscribed to all kinds of events.

The following restrictions are already mentioned. If a peer unsubscribes to events raised in G and then subscribes to the event E raised in G, that implies an initial subscription to E, or a subscription to all kinds of events (see CSUB01). The coexistence between an unsubscription of the third type and its counterpart (a subscription to events raised in G) is impossible (see CSUB03). Finally, an unsubscription to the event E coming from G is impossible (see CUSUB02). Indeed, it is not possible to have a particular unsubscription and a more restrictive unsubscription at the same time.

```
∀ networkState ∈ NetworkState, ∀ unsubscription ∈ networkState'unsubscriptions'secondType,
∃ peer ∈ networkState'peers ( (unsubscription'unsubscriber = peer'pid) ) ∧
(
  /* usub2(E,G,caller) ∧ sub1(E,G,caller) ⇒ FALSE */
     /* already constrained */
  /* usub2(E,G,caller) ∧ sub2(E,caller) ⇒ TRUE */
     /* not constrained */
  /* usub2(E,G,caller) ∧ sub3(G,caller) ⇒ TRUE */
     /* not constrained */
  /* usub2(E,G,caller) ∧ sub4(caller) ⇒ TRUE */
     /* not constrained */
  /* usub2(E,G,caller) ⇒ sub2(E,caller) ∨ sub3(G,caller) ∨ sub4(caller) */
  (
     ∃(subscription).( (subscription ∈ networkState'subscriptions'secondType) ∧
                       (subscription'event = unsubscription'event) ∧
                       (subscription'subscriber = unsubscription'unsubscriber) ) ∨
     ∃(subscription).( (subscription ∈ networkState'subscriptions'thirdType) ∧
                       (subscription'group = unsubscription'group) ∧
                       (subscription'subscriber = unsubscription'unsubscriber) ) ∨
     ∃(subscription).( (subscription ∈ networkState'subscriptions'fourthType) ∧
                       (subscription'subscriber = unsubscription'unsubscriber) )
  ) ∧
  /* usub2(E,G,caller) ∧ usub1(E,caller) ⇒ FALSE */
     /* not constrained */
  /* usub2(E,G,caller) ∧ usub3(G,caller) ⇒ FALSE */
  ∄(unsubscription2).( (unsubscription2 ∈ networkState'unsubscriptions'thirdType) ∧
                       (unsubscription2'group = unsubscription'group) ∧
                       (unsubscription2'unsubscriber = unsubscription'unsubscriber) )
)
```

Figure 3.28: Constraint CUSUB02

```
∀ networkState ∈ NetworkState, ∀ unsubscription ∈ networkState'unsubscriptions'thirdType,
∃ peer ∈ networkState'peers ( (unsubscription'unsubscriber = peer'pid) ) ∧
(
  /* usub3(G,caller) ∧ sub1(E,G,caller) ⇒ sub2(E,caller) ∨ sub4(caller) */
     /* already constrained */
  /* usub3(G,caller) ∧ sub2(E,caller) ⇒ TRUE */
     /* not constrained */
  /* usub3(G,caller) ∧ sub3(G,caller) ⇒ FALSE */
     /* already constrained */
  /* usub3(G,caller) ∧ sub4(caller) ⇒ TRUE */
     /* not constrained */
  /* usub3(G,caller) ⇒ sub2(E,caller) ∨ sub4(caller) */
  (
    ∃(subscription).( (subscription ∈ networkState'subscriptions'secondType) ∧
                      (subscription'subscriber = unsubscription'unsubscriber) ) ∨
    ∃(subscription).( (subscription ∈ networkState'subscriptions'fourthType) ∧
                      (subscription'subscriber = unsubscription'unsubscriber) )
  )
  /* usub3(G,caller) ∧ usub1(E,caller) ⇒ TRUE */
    /* not constrained */
  /* usub3(G,caller) ∧ usub2(E,G,caller) ⇒ FALSE */
    /* already constrained */
)
```

Figure 3.29: Constraint CUSUB03

Events Sendings Constraints

Events are based on a one-to-many asynchronous communication way. They can be sent by peers or services. In the case of a service, each raised event must be listed in its service contract. We have to distinguish both cases:

- **Peer constraint.** In this case, the sent event identifier is a `PeerId` element. So, we have to check if this identifier is associated with a peer belonging to the current network state. Another verification concerns event authorization, a peer can sent an event to a specific group if this peer belongs to that group otherwise it is not authorized. So, all group identifiers associated with a sent event must identify existing groups in which the sender is a member.

- **Service constraint.** In this case, the sender identifier is a service instance identifier which is either a `PeerServiceId`, or a `SessionId`. Several verifications have to be made:
  - If this identifier is a `PeerServiceId`, a non state-full session-full service linked to this identifier must exist. Otherwise, the sender identifier is a `SessionId` and a state-full session-full service must contain this identifier in its session identifier set. So, the identifier refers to a service published by a peer belonging to the peer set of the current network state.
  - The sent event name must be listed in the service contract of the concerned service.
  - The service must be published in the group in which the event is raised.

```
∀ networkState ∈ NetworkState, ∀ sentEvent ∈ networkState'sentEvents,
∀ groupId ∈ sentEvent'groups,
(
  (sentEvent'sender ∈ PeerId) ∧
  ∃ peer ∈ networkState'peers, (sentEvent'sender = peer'pid) ∧
  ∃ group ∈ networkState'groups, (group = group'gid) ∧ (sentEvent'sender ∈ group'members) )
) ∨
(
  (sentEvent'sender ∈ (PeerServiceId ∪ SessionId)) ∧
  ∃ peer∈ networkState'peers, ∃ service ∈ peer'services (
        (
         (
           (sentEvent'sender ∈ PeerServiceId) ∧ (sentEvent'sender = service'psid) ∧
           (service'contract'type ≠ sessionFull)
         ) ∨
         (
           (sentEvent'sender ∈ SessionId) ∧ sentEvent'sender ∈ service'sessions) ∧
           (service'contract'type = sessionFull)
         )
        ) ∧
        (sentEvent'name ∈ service'contract'eventsRaised)
  )
)
```

Figure 3.30: Constraint CESD01

Operations Invocations Constraints

The following constraint is divided into three parts.

```
∀ networkState ∈ NetworkState, ∀ networkState ∈ networkState'invocations,
∃ peer ∈ networkState'peers, ∃ service ∈ peer'services,
∃ operation ∈ service'contract'operations, (
            /* First part */ ∧
            /* Second part */ ∧
            /* Third part */
)
```

Figure 3.31: Constraint CI01

Constraint CI01 - Part 1. This first part checks that each invocation refers to an existing service offering the invoked operation. Invoking a service thanks to its peer identifier implies that the service is state-less or state-full session-less. A `SessionId` is used to invoke state-full session-full services. If a `PeerServiceId`, or a `SessionId` is used to invoke a service, the invocation provider must be this identifier (can be `empty` if the invoked service has not received the message yet). If a `GroupServiceId` is used, the service provider must be one service instance among other service instances identified by this group service identifier (`service'psid`).

```
(invocation'op = operation'name) ∧
(
  (
    (invocation'callee ∈ PeerServiceId) ∧ (invocation'callee = service'psid)
    (service'contract'type ≠ sessionFull) ∧
    ( (invocation'provider ≠ empty) ⇒ (invocation'provider = invocation'callee) )
  ) ∨
  (
    (invocation'callee ∈ GroupServiceId) ∧ (invocation'callee = service'gsid)
    (service'contract'type = stateLess) ∧
    ( (invocation'provider ≠ empty) ⇒ (service'psid = invocation'provider) )
  ) ∨
  (
    (invocation'callee ∈ SessionId) ∧ (invocation'callee ∈ service'sessions) ∧
    (service'contract'type = sessionFull) ∧
    ( (invocation'provider ≠ empty) ⇒ (invocation'provider = invocation'callee)
  )
)
```

Figure 3.32: Constraint CI01 - Part 1

Constraint CI01 - Part 2. SMEPP doesn't allow that the same running instance process simultaneously several invocation of the same request-response operation from the same entity before the caller retrieves its previous invocation. So, there is only one invocation having the same caller identifier, provider identifier and operation name. The second conjunction avoids that the same client invokes the same target i.e., the same callee identifier (otherwise, the invoker cannot distinguish its requests). Note that, in the case of a state-less service, it is possible to invoke a service in a given group thanks to its group service identifier/peer service identifier and then, calling the same service in the same group by using a peer service identifier/group service identifier. For example, there is no restriction if the `Equipment` peer calls the `Monitoring` service (we consider it as a state-less service) in the `SequiTel` group by using its group service identifier and then, before retrieving the previous result, this peer calls `Monitoring` again but now by using its peer service identifier.

Constraint CI01 - Part 3. If a peer invokes a service instance, the peer must belong to the group in which the service has been published. If a service invokes another service, both services must be published in the same group. A service invoker is identified by a peer service identifier only if the service is not a state-full session-full else, the service is identified by a session identifier.

```
        (
          (operation'type = requestResponse) ⇒
          ∄(invocation2).(
                           (invocation2 ∈ networkState'invocations) ∧
                           (invocation'caller = invocation2'caller) ∧
                           (invocation'provider = invocation2'provider) ∧
                           (invocation'op = invocation2'op) ∧
                           (invocation'in ≠ invocation2'in)
          ) ∧
          ∄(invocation2).(
                           (invocation2 ∈ networkState'invocations) ∧
                           (invocation'caller = invocation2'caller) ∧
                           (invocation'callee = invocation2'callee) ∧
                           (invocation'op = invocation2'op) ∧
                           (invocation'in ≠ invocation2'in)
          ) ∧
        ) ∧
```

Figure 3.33: Constraint CI01 - Part 2

```
(
  (
    (invocation'caller ∈ PeerId) ∧
    ∃(peerCaller).( (peerCaller ∈ networkState'peers) ∧ (peerCaller'pid = invocation'caller) ) ∧
    ∃(group).( (group ∈ networkState'groups) ∧ (invocation'caller ∈ group'members) ∧
               (group'gid = service'gid) )
  ) ∨
  (
    ∃(peerCaller,serviceCaller).( (peerCaller ∈ networkState'peers) ∧
                 (serviceCaller ∈ peer'services) ∧
                 (service'gid = serviceCaller'gid) ∧
                 (
                   (
                     (invocation'caller ∈ PeerServiceId) ∧
                     (serviceCaller'psid = invocation'caller) ∧
                     (serviceCaller'contract'type ≠ sessionFull) ∧
                     (invocation'caller = serviceCaller'psid)
                   ) ∨
                   (
                     (invocation'caller ∈ SessionId) ∧
                     (invocation'caller ∈ service'sessions) ∧
                     (service'contract'type = sessionFull)
                   )
                 )
    )
  )
)
```

Figure 3.34: Constraint CI01 - Part 3

Note that, there is no constraint on the input type. So, it is possible to give arguments of a different type than types required by the service contract.

Operations Replies Constraints

Constraints associated with operations replies are divided into three parts. The first part restricts normal results, the second part is shared between the first and the third part. The last part constraints abnormal responses.

Constraint CR01. First, we have to check that there isn't another normal reply associated with the same invocation. Of course, this reply must be different from the considered reply. Secondly, an abnormal result associated with this invocation cannot be present.

```
∀ networkState ∈ NetworkState, ∀ reply ∈ networkState'replies'normalResults,
∄ reply2 ∈ networkState'replies'normalResults, (
  (reply2'callee = reply'callee) ∧ (reply2'caller = reply'caller) ∧
  (reply2'op = reply'op) ∧ (reply2'out ≠ reply'out)
) ∧
∄ reply2 ∈ networkState'replies'missResults, (
  (reply2'callee = reply'callee) ∧ (reply2'caller = reply'caller) ∧
  (reply2'op = reply'op)
) ∧
/* Constraint part shared by CR01 and CR02 */
```

Figure 3.35: Constraint CR01

Constraint CR01-02. The following constraint is a common part of both replies constraints. A reply must be associated with a previous invocation. So, a reply and its associated invocation are present in the network data structure until the invoker has not received the invocation result yet. Moreover, only request-response operations can return a result. In other words, if there is a reply on network, this reply must concern a request-response operation.

```
∃(invocation).( (invocation ∈ networkState'invocations) ∧ (invocation'caller = reply'caller) ∧
                (invocation'provider = reply'callee) ∧ (invocation'op = reply'op) ) ∧
∃(peer,service,operation).( (peer ∈ networkState'peers) ∧ (service ∈ peer'services) ∧
                            (operation ∈ service'contract'operations) ∧
                            (operation'type = requestResponse) )
```

Figure 3.36: Constraint Part Shared by CR01 and CR02

Constraint CR02. This last constraint is related to miss results. A service can only reply once to an invocation. So, it is not possible to have a normal result and a miss result associated with the same invocation. It is also impossible that another miss result (different from the concerned result) associated with this invocation exists.

SMEPP Network Constraint

Our SMEPP model only concerns one SMEPP network. So, our purpose is not the representation of several SMEPP networks interacting together or acting at the same time. So the `NetworkState` set contains all the valid states of *only one* SMEPP network.

```
∀ networkState ∈ NetworkState, ∀ reply ∈ networkState'replies'missResults,
∄ reply2 ∈ networkState'replies'normalResults, (
  (reply2'callee = reply'callee) ∧ (reply2'caller = reply'caller) ∧
  (reply2'op = reply'op)
) ∧
∄ reply2 ∈ networkState'replies'missResults, (
  (reply2'callee = reply'callee) ∧ (reply2'caller = reply'caller) ∧
  (reply2'op = reply'op) ∧
  ( (reply2'out ≠ reply'out) ∨ (reply2'fault ≠ reply'fault) )
∧
/* Constraint part shared by CR01 and CR02 */
```

Figure 3.37: Constraint CR02

Constraint CN01. The first constraint concerns credentials. A SMEPP network is associated with credentials which allow to restrict network access. In this model, we consider that these credentials don't change during the network lifetime. So, all network states of *a* SMEPP network are associated with the same credentials. The following constraint checks this restriction:

```
∃ credentials∈ Credentials, ∀ networkState ∈ NetworkState,
(credentials = networkState'cred)
```

Figure 3.38: Constraint CN01

Constraint CN02. `currentPeer` is a constant referencing the peer associated with the B-machine. This constant must belong to `PeerId`. Of course, the referred peer must belong to the peer set. What's more, `NetworkState` is composed of at least one state.

```
currentPeer ∈ PeerId ∧
∀(networkState ∈ NetworkState).(
     ∃ peer ∈ networkState'peers, (peer'pid = currentPeer) )
) ∧
(card(NetworkState) ≥ 1)
```

Figure 3.39: Constraint CN02

Variables Constraints

Our model contains only one variable. The `network` variable contains the current SMEPP network state. Thus, this variable must belong to `NetworkState` which contains all the *valid* network states.

```
VARIABLES
  network
INVARIANT
  (network ∈ NetworkState)
```

Figure 3.40: Constraint CV01

3.4.4 Peer Management Primitives

Now we have designed the data structure needed to model a SMEPP network state, we can implement each primitive as a B-operation of our B-machine and see how they interact with the data, the machine state. All B-operations are displayed in the Appendix B.

newPeer

Primitive signature.

```
peerId newPeer(credentials) throws InvalidCall
```

Primitive implementation. This primitive is implemented in a different way from other primitives. Indeed `newPeer` is implemented through the `Initialization` clause of the B-machine. The B-code begins by setting the current network structures. `currentPeer` is randomly chosed in `PeerId`, this identifier must refer to a peer belonging to `currentNetworkState` which currently offers no services. This implementation doesn't take into account whether the network already exists, or whether it has just been created by `currentPeer`.

Primitive constraints. Primitive reserved to peers

getPeerId

Primitive signature.

```
peerId getPeerId(serviceId?)  throws InvalidId, InvalidCall
```

Primitive implementation. The B-model of this primitive is divided into two parts. The first part (see Figure B.1.2, page 139) models the primitive without any arguments. This part is quite simple, this operation returns the value of the constant referring to the current peer identifier.

The second part (see Figure B.1.2, page 140), more complex, implements the primitive with only one argument called `serviceId`. But, the implementation has one more argument than the conceptual primitive. Indeed, the SMEPP API is able to know which is the caller of a primitive. But, in our model, the B-code cannot know the context of an operation call. So, we have to explicitly add an argument called `caller`, identifying the operation caller. The caller is either the peer associated with the B-machine, or one of its services.

The first part of the operation pre-condition checks that the given `serviceTypeId` is associated with an existing published service. The second part avoids that the caller gets the identifier of a peer that has published a service in a group to which the caller doesn't belong to. The caller is either a service, or a peer. In the last case, the peer must belong to the group in which the service identified by `serviceTypeId` has been published. In case of a service instance, it must have been published in the same group as the service `serviceTypeId`. The behaviour part initializes the `pid` constant. The constant value belongs to the `PeerId` set. Its value must be associated with an existing peer. The referred peer must have published the service identified by `serviceId`. `serviceId` belongs either to `PeerServiceId` and it must identify a service instance published by the referred peer. Or `serviceId` belongs to `SessionId` and the referred peer must have published a state-full session-full service which has one of its sessions identified by `serviceId`.

3.4.5 Group Management Primitives

createGroup

Primitive signature.

```
groupId createGroup(groupDescription, credentials) throws AccessDenied,
                                                          InvalidCall
```

Primitive implementation. Creating a group requires a new unique group identifier. This is the purpose of the `ANY` clause, searching for a group identifier not currently in use. The second part of this clause initializes the new current network state. The new state equals the previous state (all of their fields are equal) expect for the field called `groups` where a new 4-uplet is added, the new created group.

Primitive constraints. Primitive reserved to peers

getGroups

Primitive signature.

```
groupId[] getGroups(groupDescription?,credentials) throws InvalidCall
```

Primitive implementation. `getGroups` is implemented in a simple way. Indeed, matching processes have been very simplified. The first matching processes is optional and concerns `groupDescription`. Thanks to `groupDescription`, the caller can specify the name of wanted groups and/or a description of the required group security. The matching rule is a simple equality. So, the structure of the description is ignored. The second matching process checks that the given `credentials` allow to access to a group. Once again, this matching process is implemented as an equality, avoiding to decompose the structure of credentials and check that some of its elements match the group credentials.

Primitive constraints. Primitive reserved to peers

getGroupDescription

Primitive signature.

```
groupDescription getGroupDescription(groupId,credentials) throws InvalidGroupId,
                                                                 InvalidCall
```

Primitive implementation. The operation pre-condition checks that the given credentials is rightly associated with an existing group identified by `groupId`. Its body consists of initializing the variable `descr` to the description of the group identified by `groupId`.

Primitive constraints. Primitive reserved to peers

joinGroup

Primitive signature.

```
void joinGroup(groupId,credentials) throws AccessDenied, InvalidGroupId,
                                           InvalidCall
```

Primitive implementation. The operation pre-condition checks that the given credentials are correctly associated with an existing group identified by `groupId`. The operation behaviour part constructs the variable `networkUpdated` which contains the updated network state. It is the same as the previous state expect that the group identified by `groupId` has a new group member, the current peer.

Primitive constraints. Primitive reserved to peers

leaveGroup

Primitive signature.

```
void leaveGroup(groupId) throws InvalidGroupId, CallerNotInGroup,
                                InvalidCall
```

Primitive implementation. The operation pre-condition firstly checks whether the given `groupId` identifies an existing group (see `InvalidGroupId` exception). Secondly, the current peer (identified by `currentPeer`) must be a member of that group i.e., `currentPeer` must belong to `group'members` (cfr `CallerNotInGroup`). The operation behaviour starts by creating a variable containing the new network state which is the same as the previous state expect that the group `group` is removed from the network state and replaced by `group2` which doesn't contain the current peer. If the new updated group `group2` doesn't contain any peer, `group` is removed, but not replaced by `group2`.

Primitive constraints. Primitive reserved to peers

getIncludingGroups

Primitive signature.

```
groupId[] getIncludingGroups() throws InvalidCall
```

Primitive implementation. The operation body simply constructs the set containing all identifiers of groups in which the current peer is a member.

Primitive constraints. Primitive reserved to peers

getPublishingGroup

Primitive signature.

```
groupId getPublishingGroup(serviceTypeId) throws InvalidCall, InvalidId
```

Primitive implementation. The pre-condition checks that the given `serviceTypeId` is rightly associated with an existing published service which belongs to the same group as the primitive caller. The `ANY` clause placed in the body initializes the constant `gid` to the group associated with the group in which the service identified by `serviceTypeId` has been published.

Primitive constraints. Primitive reserved to peers

GetPeers

Primitive signature.

```
peerId[] getPeers(groupId?,credentials) throws InvalidGroupId, InvalidCall
```

Primitive implementation. Normally, services are not allowed to provide a `groupId` and peers must provide it. But in this model, it has been conceived as mandatory. Indeed, `getPeers` called without any `groupId` is the same as first, searching for the group where the primitive caller (a service) has been published and secondly calling `getPeers` with this group identifier.

`getPeers` begins by checking that the specified group identifier really exists. The operation body consists of setting the returned set to the `members` field of the group n-uplet identified by the given `groupId`.

3.4.6 Service Management Primitives

publish

Primitive signature.

```
<groupServiceId, peerServiceId>
                 publish (groupId, contract, grounding) throws InvalidServiceSpecification,
                                                               InvalidGrounding,
                                                               InvalidGroupId,
                                                               CallerNotInGroup,
                                                               InvalidCall
```

Primitive implementation. The service model allows peers to re-publish a service i.e., to publish again a service in the same group with the same service contract, published by the same peer. Re-publishing a service is the same as calling `unpublish` and then `publish`. In our model, in order to remain as simple as possible, the implementation of `publish` doesn't allow peers to re-publish a service. This is the purpose of the constraint following the comment "*Check that this service has not been already published by the same peer in the same group*". This constraint checks that the caller, a peer, has not already published the same service, `contract`, in the same group `groupId`. An implementation suiting the conceptual model must verify that this constraint is false, in this case, it has to call `unpublish` with the identifier of the previous published service and then executes the behaviour part of the operation displayed in Figure B.3.1 (page 146).

The second part of the operation pre-condition confirms that `currentPeer` rightly belongs to the group in which it is publishing a service. The behaviour part begins by searching for a `GroupServiceId`. If a service has been already published in the group `groupId` with the same contract, the group service identifier is set to value of the `GroupServiceId` associated with the found service. Otherwise, the primitive searches for a `GroupServiceId` not currently in use. Afterwards, the primitive searches for a new peer service identifier not currently in use. Indeed, all services instances have a unique peer service identifier. The behaviour ends by setting the new network state, the peer identified by `currentPeer` has one more service.

Primitive constraints. Primitive reserved to peers

unpublish

Primitive signature.

```
void unpublish(peerServiceId) throws InvalidServiceId, CallerNotServiceOwner,
                                     InvalidCall
```

Primitive implementation. The first operation part consists of initializing two constants: `peerOld` and `peerUpdated`. `peerOld` is the peer state before unpublishing. `peerUpdated` is the same state expect that the service identified by `peerServiceId` has been unpublished. The first state is removed from the peer set of the new network state while the second, `peerUpdated` is added.

Once a service is unpublished, its previous subscriptions and unsubscriptions are deleted. If the service has invoked a service, its invocations are also removed. If the service has received a message, but hasn't replied before being unpublished, its associated invocations are removed. Note that, *replies* made by an unpublished service are *not removed*; this is an implementation choice.

Primitive constraints. Primitive reserved to peers

getServices

Primitive signature.

```
<groupId?,groupServiceId,peerServiceId>[]
        getServices(groupId?, peerId?,servicecontract?,
                maxResults?,credentials) throws InvalidGroupId
                                                InvalidPeerId
                                                InvalidServiceSpecification
                                                InvalidCall
```

Primitive implementation. In the B-implementation, all arguments are declared mandatory avoiding several operations, one operation per given arguments. Indeed, the B-language doesn't support optional operation arguments. So, all arguments of a B-operation are mandatory. There are 16 possible signatures, for example, `getServices(credentials)`, `getServices(groupId,credentials)`, `getServices(peerId,maxResults,credentials)`, ...

In order to avoid 16 B-operations implementing `getServices`, only one operation is present in the B-machine. The operation signature contains all the arguments of `getServices`. Each argument acts as a filter in the result set. Indeed, each result must satisfy the constraint associated with the argument value. So, removing an argument is simply removing constraints associated with this argument. Other 15 implementations can be obtained thanks to the B-operation displayed in the Figure B.3.3 (page 148).

First, the pre-condition verifies that the provided peer and group identifier refer to an existing peer and an existing group. The substitution part initializes the value of the constant called `resultsVar` which will be used to assign the returned result of `getServices`. Each element of `resultsVar` is a triplet containing a group identifier, a `GroupServiceId` and a `PeerServiceId`. The *returned group identifier* becomes *mandatory*; in the service model, the group identifier is only returned if the caller is a peer. The line `(result'groupId = service'gid)` is normally not present if the caller is a service.

Another difference between the implementation and the service model also concerns the group identifier. In our implementation, the argument `groupId` has been conceived as mandatory. Indeed, the behaviour of `getServices` called without any `groupId` is the same as first, searching for the group where the caller, a service has been published and secondly calling `getServices` with its group identifier. Consequently, it is not necessary to create a second implementation of `getServices` limited to services.

getServiceContract

Primitive signature.

```
contract getServiceContract(serviceTypeId) throws InvalidServiceId,
                                                  InvalidCall
```

Primitive implementation. The implementation of `getServiceContract` has one more argument than the conceptual primitive. Indeed, the SMEPP API is able to know which is the caller of a primitive. But, in our model, the B-code cannot know the context of an operation call. So, we have to explicitly add an argument, called `caller`, identifying the operation caller. The caller is either the peer associated with the B-machine (identified by `currentPeer`), or one of its services.

The first part of the operation pre-condition checks that the given `serviceTypeId` is associated with an existing published service. The second part avoids that the caller gets the contract of a service instance published in a group in which the caller doesn't belong. The caller is either a service, or a peer. In the last case, the peer must belong to the group in which the service identified by `serviceTypeId` has been

published. In case of a service instance, it must have been published in the same group as the service `serviceTypeId`. The behaviour part initializes the constant `contract` to the contract of the service identified by `serviceTypeId`. Later, `contract` serves to set the value of the returned value.

startSession

Primitive signature.

```
sessionId startSession(serviceTypeId) throws InvalidServiceId, AccessDenied,
                                             CannotStartSession, InvalidCall
```

Primitive implementation. As explained in the previous implementation, the argument called `caller` has been added. First of all, the provided `serviceTypeId` (either a `GroupServiceId` or a `SessionId`) must refer to an existing state-full session-full service. Afterward, the primitive checks that the referred service is visible from the caller. Only services published in a group to which the caller belongs, are visible. The primitive caller is either a service, or a peer. In the last case, the peer must belong to the group in which the service identified by `serviceTypeId` has been published. In case of a service instance, it must have been published in the same group as the service `serviceTypeId`.

The behaviour part initializes three constants: `sessionId`, `serviceUpdated`, `networkUpdated`. The first constant is setted to a `SessionId` not used by the current network i.e., no current session is identified by `sessionId`. The second constant, `serviceUpdated` is a published service identified by `serviceTypeId`. If this identifier belongs to `PeerService`, only one service can satisfy "∃(peer,service).(...)". Otherwise, `serviceTypeId` belongs to `GroupServiceId` and several published services can satisfy this constraint. One of them is randomly chosen to assign `serviceUpdated`. The rest of the behaviour clause updates the `peers` field of the current network. The session set of `serviceUpdated` has one more session identified by `sessionId`.

3.4.7 Message Management Primitives

invoke

Primitive signature.

```
output?  invoke(serviceTypeId, operationName,
                input?, doReturnResult?, timeout?)  throws InvalidServiceId,
                                                           InvalidSessionId,
                                                           InvalidOperation, AccessDenied,
                                                           ConcurrentRequest,
                                                           InvalidInputParameter,
                                                           InvalidOutputParameter,
                                                           InvalidCall, ExpiredTimeout
```

Primitive implementation. `serviceTypeId` must refer to an existing service instance and must correspond to the service type (state-less, state-less session-less and session-full). The referred service instance must provide the operation `operationName`. `invokeCaller` has been added. It identifies the entity which calls the `invoke` primitive. If `doReturnResult` is set to `true`, the concerned operation must be a request-response operation. Even if this parameter is true, the operation behaves as if `doReturnResult` was set to false. That simplifies the operation implementation. Indeed, calling invoke with a return result is simply calling the B-operation `invoke` with `doReturnResult` to true and then, calling `receiveResponse` and returning the response.

In the case of a request-response, the invoker cannot call a second time the same operation of the running service instance (identified either by a `PeerServiceId`, or by a `SessionId`) before retrieving the previous result. Expect, if the entity invokes a service by using a group service identifier, the service instance (which must be state-less) cannot be the sole instance of the service in its group. So, there must be more than one service in that group identified by `GroupServiceId` (`service'gsid = serviceTypeId`). Indeed, a second service instance associated with `serviceTypeId`, different from the service instance processing the previous invocation, must be able to receive the new request.

The new invocation is added to the new network state, the invocation provider is set to `empty` which means that the new invocation has not been received yet.

receiveMessage

Primitive signature.

```
<callerId,input?> receiveMessage(operationName,timeout) throws InvalidOperation,
                                                               InvalidInputParameter,
                                                               InvalidCall
```

Primitive implementation. `receiveMessage` has one extra argument called `calleeId` allowing the B-operation to know which service calls the `receiveMessage` operation. First, we have to check that the service identified by `calleeId` offers the operation called `operationName`. `calleeId` is either a `SessionId` used by state-full session-full services, or a `PeerServiceId` used by session-less or state-less services.

Invocations belong to the network invocation set. This invocation cannot be already received by another service (`provider = empty`). If the invoker used a `SessionId`, or `PeerServiceId`, the service provider (`invocation'callee`) must be `calleeId`. Note that, `NetworkStates` constraints check that there is no concurrent request. But, if a `GroupServiceId` has been used to invoke the receiver (a state-less service), we have to check that the receiver is not currently processing a request from the same invoker i.e., another invocation from the same entity (`callerId` are equal) on the same operation

Once an invocation received, the invocation provider field is set to `calleeId`. Other entities can know that `calleeId` is now processing the request.

Primitive constraints. Primitive reserved to services

reply

Primitive signature.

```
void reply(callerId,operationName,output?,faultName?)  throws InvalidPeerId,
                                                              InvalidServiceId,
                                                              InvalidSessionId,
                                                              InvalidOperation,
                                                              InvalidOutputParamter,
                                                              MissingReceiveMessage,
                                                              InvalidCall
```

Primitive implementation. In order to identify the service trying to reply to an invocation, `calleeId` has been added. First, `calleeId` must rightly identify a service instance providing the operation `operationName`. Secondly, `callerId` must refer to an existing entity on the current network. This entity must have previously invoked the operation `operationName` and `calleeId` must have received this invocation (`invocation'provider = calleeId`). Moreover, we cannot reply twice to the same

invocation thus, there is no existing reply associated with `callerId`, `calleeId` and `operationName`. If `faultName` is empty, the reply is added to `normalResults`. Otherwise, the invocation process leads to an exception and the reply is placed in `missResults`.

Primitive constraints. Primitive reserved to services

receiveResponse

Primitive signature.

```
output receiveResponse(serviceTypeId,operationName,timeout?)  throws InvalidServiceId,
                                                                     InvalidSessionId,
                                                                     InvalidOperation,
                                                                     ConcurrentRequest,
                                                                     InvalidOutputParamter,
                                                                     MissingInvokeMessage,
                                                                     InvalidCall,
                                                                     ExpiredTimeout
```

Primitive implementation. An argument called `callerId` identifies the entity trying to receive an invocation result. This argument is not present in the service model. We have to check that an existing service is rightly identified by `serviceTypeId` and offers the operation called `operationName`. Other checks verify that `callerId` identifies an existing entity. This entity must belong to the same group as the service instance `serviceTypeId`. Receiving a response implies a previous invocation of `callerId` on the operation `operationName` using the same target identifier `serviceTypeId`.

Once checked, the operation builds the new network state contained in `networkUpdated`. Then, it searches for a corresponding reply in the replies set of the current network. The reply is either in the `normalResults` set, or in the `missResults` set. The searched reply must be associated with the current entity (`reply'caller = callerId`), to the operation `operationName` provided by `serviceTypeId` (`invocation'callee = serviceTypeId`). Once retrieved, the reply and its associated invocation are removed from the `replies` and `invocations` set of `networkUpdated`.

If the invocation hasn't lead to an exception (reply belonging to `normalResults`), the returned fault name (`output'fault`) is `empty`. Note that, the real primitive raises an exception is placed of returning its name.

3.4.8 Event Management Primitives

event

Primitive signature.

```
void event(groupId?,eventName,input?)  throws InvalidGroupId, CallerNotInGroup,
                                              InvalidEvent, InvalidCall
```

Primitive implementation. One extra argument has been added: `callerId`. This argument allows the primitive to identify its caller. If the caller is a peer (`callerId` in `PeerId`), it must be the peer associated with the B-machine. Only peers can specify a `groupId` (`groupId` different from `empty`) associated with a group in which the peer belongs. In all cases, if `groupId` is `empty` that means an unspecified group identifier.

If the caller is a service, the service must be published by the current peer. Services cannot specify a `groupId`, it must be `empty`. `eventName` must belong to the set of events raised by the service denoted

by: `eventName` ∈ `serviceCaller'contract'eventsRaised`.

The operation effects consists of initializing a group set called `sentGroups` and the new network state (`networkUpdated`) containing one more sent event. `sentGroups` contains identifiers of group in which the new event will be send. In the case of a service caller, this set is a singleton containing the identifier of the group in which the service has been published. `sentGroups` can also be a singleton set if peers call `event` and specify a `groupId`. Otherwise, it contains group in which the current peer is a member. Once `sentGroups` initialized, `event` can add a new event n-uplet to the sent event set of `networkUpdated`. Note that `input` is mandatory, an unspecified `input` is equivalent to provide the `empty` value.

receiveEvent

Primitive signature.

```
<callerId,input?> receiveEvent(groupId?,eventName,timeout?)  throws InvalidGroupId,
                                                                    CallerNotInGroup,
                                                                    InvalidCall,
                                                                    ExpiredTimeout
```

Primitive implementation. Only one B-operation implements `receiveEvent(groupId,eventName)` and `receiveEvent(eventName)`. One extra argument has been added in the event operation: `callerId` allowing the primitive to identify its caller. If the caller is a peer, its identifier must be the same as `currentPeer`. Only peers can specify a group identifier; a group identifier is specified if its value is different from `empty`. If the caller is a service, the caller must be published by `currentPeer` and `groupId` must be `empty`.

The main part of this operations consists of initializing the constant `gid`. Its possible values are groups concerned by the listening. So, if a `groupId` is specified, `gid` must only contain `groupId` since `receiveEvent` listen only to events `eventName` raised in `groupId`. Otherwise, if the caller is a service, `gid` is set to the identifier of the group in which the caller has been published (a service can only retrieve events raised in the group where it has been published). In the case of a peer which has not specified a `groupId`, `gid` is composed of group identifiers where the peer is a member.

The rest of the operation behaviour consists of randomly choose a sent event called `eventName` raised in a group identified by an element of `gid`. This event must be associated with an existing subscription not restricted by an unsubscription which prohibits the reception of this event.

subscribe

Primitive signature.

```
void subscribe(eventName?,groupId?)  throws InvalidGroupId, CallerNotInGroup,
                                            InvalidCall
```

Primitive implementation. Each kind of subscription has been implemented in a specific B-operation. Some checks are shared between operations: every time a caller is specified, we must check that in case of a peer, `callerId` has the same value as `currentPeer`, in the case of a service, `callerId` identifies an existing service published by the current peer. If a `groupId` is provided, operations must check that the caller, a peer, belongs to an existing group identified by `groupId`. Table B.1 (page 158) contains scenarios involving the `subscribe` primitive is called. The syntax used in that Table has been explained in Subsection 3.4.3, "Events Subscriptions Constraints".

First subscription type. The peer associated with the B-machine (`currentPeer`) subscribing to the event `eventName` raised in the group `groupId`, is denoted by `sub1(eventName,groupId,currentPeer)`. As previously mentioned, once this subscription made, it is not possible to have a corresponding unsubscription of the second type: `usub2(eventName,groupId,currentPeer)` meaning that the subscriber has unsubscribed at the same time from `eventName` raised in `groupId`. Thus, if a previous unsubscription exists, it is removed from the unsubscription set. In the third part of Table B.1 (on page 158), we can see that this new subscription is not added in the subscription set if a previous, more generic subscription exists:

- a subscription to events `eventName` (second type),
- a subscription to events raised in `groupId` (third type),
- a subscription to events (fourth subscription type).

which is *not* restricted by a unsubscription to:

- an event called `eventName`(first type),
- events raised in `groupId` (third type).

These possibilities correspond to lines: 2, 3, 4, 7, 8 and 9 of the third part of Table B.1 (on page 158). In other cases, `sub1(eventName,groupId,currentPeer)` is added to the subscription set.

Second subscription type. This kind of subscription concerns an entity (a peer or a service) which subscribes to an event called `eventName`. The first operation behaviour part updates subscriptions of the first type. If the subscriber has previously unsubscribed to events raised in `groupId`, a subscription to the event `eventName` raised in `groupId` is created, notifying that although the entity doesn't want to receive events raised in `groupId`, now it wants to receive the event `eventName` (no matter groups in which it is raised). If this unsubscription doesn't exist, all less generic subscriptions (subscriptions to a certain event raised in a specific group) are removed.

Afterward, subscriptions to events raised in a given group (second subscription type) are not updated if some conditions are present, see line 2, 5, 7, 9, 10 and 11 (see Table B.1). Otherwise, a new subscription to the event `eventName` is added. Third and fourth subscription type are not changed.

At the end of `subscribeSecondType`, unsubscriptions of the first and second type are changed. Previous unsubscriptions to the event `eventName` are removed, less specific unsubscriptions are also deleted (unsubscription to a certain event raised in a given group).

Third subscription type. Subscriptions of the third type can only be made by peers since a group identifier `groupId` is specified allowing peers to subscribe to all kinds of events raised in `groupId`. The first operation' isbehaviour part updates subscriptions of the first type. If the subscriber has previously unsubscribed to the event `eventName` (first unsubscription type), a subscription to the event `eventName` in `groupId` is created notifying that even if the entity doesn't want to receive the event `eventName`, it now wants to receive event raised in `groupId` (no matter the event name). If this unsubscription doesn't exist, less generic subscriptions (subscriptions to a given event raised in a specific group) are removed.

Afterward, subscriptions to events raised in a given group (third subscription type) are not updated if some conditions are present, see line line 2, 5, 6, 7, 8 and 10 of Table B.2. Otherwise, a new subscription to events raised in `groupId` is added. Second and fourth subscription type are not changed.

At the end of `subscribeThirdType`, unsubscriptions of the second and third type are changed. The previous unsubscriptions to events raised in `groupId` (third type) is removed, less specific unsubscriptions are also deleted (unsubscription to a certain event raised in a given group, the second type).

Fourth subscription type. When an entity (a peer, or a service) subscribes to all kinds of events, its previous subscriptions and unsubscriptions are simply removed.

unsubscribe

Primitive signature.

```
void unsubscribe(eventName?,groupId?)  throws InvalidGroupId, CallerNotInGroup,
                                              NotSubscribed, InvalidCall
```

Primitive implementation. Each kind of unsubscription has been implemented in a specific B-operation. Some checks are shared between operations: every time a caller is specified, we must check that in case of a peer, `callerId` has the same value as `currentPeer` and that in the case of a service, `callerId` identifies an existing service published by the current peer. If a `groupId` is provided, operations must check that the caller, a peer, belongs to an existing group identified by `groupId`. Table B.2 (page 163) contains scenarios where `unsubscribe` is called.

First unsubscription type. In the first unsubscribe operation (Figure B.5.4, page 166), peers and services unsubscribe to a given event `eventName`. The operation pre-condition verifies that the caller has previously made at least one of these subscriptions:

- a subscription to the event `eventName` raised in a certain group (first subscription type),
- a subscription to the event called `eventName` (second type),
- a subscription to events raised in a certain group `groupId` (third type),
- a subscription to all kinds of events (fourth type).

The operation begins by removing previous less generic subscriptions:

- subscriptions to an event `eventName` in a certain group `groupId` (first subscription type),
- subscriptions to the event `eventName` (second subscription atype).

Third and fourth subscription type are unchanged. The first unsubscription type, an unsubscription to the event `eventName` is added to the unsubscription set only if there is still a generic subscription (third or fourth subscription type).

If there are previous unsubscriptions to the event `eventName` raised in a certain group (second unsubscription type) then they are removed because they are less specific than the new unsubscription. Unsubscriptions of the third type are kept if there was a subscription of the second type (see line 8 and 9 of Table B.2, page 163).

Second unsubscription type. The operation displayed in Figure B.5.4 (page 165) corresponds to the implementation of the primitive allowing peers to unsubscribe to a given event `eventName` in a specific group `groupId`. We must first check that it has at least one of this subscription:

- a subscription to the event `eventName` raised in `groupId` (first subscription type),
- a subscription to `eventName` (second type),
- a subscription to events raised in `groupId` (third type),
- a subscription to all kinds of events (fourth type).

If the peer has previously made a subscription to the event `eventName` in `groupId` (first subscription type), it is removed from the network. Other previous subscriptions are unchanged.

The new unsubscription is added to the unsubscription set only if there is no generic unsubscription: a first unsubscription type (unsubscription to `eventName`), or an unsubscription of the third type

(unsubscription to events raised in `groupId`) and if there is a previous generic subscription: a subscription of the second (subscription to the event `eventName`), third (subscription to events in `groupId`), or fourth type (all kinds of events).

Third unsubscription type. A third subscription type (unsubscribing to events raised in `groupId`) requires that the caller has previously made at least one of these subscriptions:

- a subscription to a certain event raised in `groupId` (first subscription type),
- a subscription to a certain event (second type),
- a subscription to events raised in `groupId` (third type),
- a subscription to all kinds of events (fourth type).

Then, the operation removes the following subscriptions:

- subscriptions to a certain event raised in `groupId` (first type),
- subscriptions to events raised in `groupId` (third subscription type).

The new unsubscription is added to the network only if there is still one of these generic subscriptions (see line 2, 4, 7, 8, 10 of Table B.2, page 163):

- a subscription to a certain event (second subscription type),
- a subscription to all kinds of events (fourth type).

Unsubscription of the first type are kept only if there is a subscription of the third type, see line 5 and 6 of Table B.2. Moreover, specific unsubscriptions are removed (unsubscriptions of the second type).

Fourth unsubscription type. When an entity (a peer, or a service) unsubscribes to all kinds of events, it must have made at least one subscription. Its previous subscriptions and unsubscriptions are simply removed.

3.5 Conclusion

This chapter intended to model the SMEPP API primitives. Firstly, we begun by introducing the SMEPP API primitives (this introduction is based on [Coa08a]). Section 3.2 has shown how SMoL is able to manipulate these primitives in order to design complex EP2P applications. In the previous Chapter, Section 2.2.3, we have introduced an health-care application called SequiTel extracted from [BL08]. This example has been completed in our thesis by illustrating how SMoL can be used to easily implement some SequiTel features. In few lines, we have created an EP2P application much more simpler to program and understand than one of its possible translations in Java, or in any other programming language.

Section 3.4 is the main contribution of this thesis. Firstly, we have modeled all the possible SMEPP network states as mathematics sets implemented in a B-machine (written in the B-language). Secondly, we have formalized one by one the API primitives as B-operations acting on a given network state and updating it. Our B-model is able to complete the current documentation of the SMEPP API. Moreover, our it can be simulated by making operations on it and seeing how the network state is updated.

SMoL has not been formalized seeing that the B-language doesn't provide constructions allowing to implement SMoL commands. Indeed, a B-machine cannot specify how its operations can be used. So, it is not possible to define in a B-machine how a SMoL command orchestrate its operations. However, the co-usage of our B-machine and CSP[MB05] can resolve this problem. In order to see how these languages can be combined, we develop the traffic management application introduced in Subsection 2.2.4. Figure 3.41

contains a sample pseudo-SMoL code. This sample code searches for an available car parking (having free places) in the group associated with the current car position (e.g., the Washington group). If an available car parking is found, a car place is reserved during one hour. Figure 3.42 contains the translation of this application. The CSP code is composed of three processes: `MAIN`, `SEARCH` and `RESERVE`. The first process initializes services offered by parkings in the group associated with the current car position. The second process receives a sequence of services, the identifier of the current cheapest service (service offered by the cheapest parking) and the current cheapest price. `SEARCH` is a recursive process on the sequence of services. If the service, placed first in the service sequence, is provided by the current cheapest available parking, the process calls itself and provides this service and its associated price. In all cases, the process continues to search in the tail of the service sequence for a service provided by the cheapest available parking. The last process reserves a car place only if a parking has been found (`cheapestService != null`)). Chapter 5 will provide further suggestions about SMoL translation in CSP.

```
services = getServices(currentGroup, carParkContract, myCredentials)
i = 0
cheapestService = -1
cheapestPrice = opaque
While (i < services.length )
       freePlaces = invoke(services[i], ‘‘get_free_places’’, null, true)
       If (freePlaces > 0 )
           prices = invoke(services[i], ‘‘get_prices’’, null, true)
           currentPrice = opaque
           If ( currentPrice < cheapestPrice )
                cheapestPrice = currentPrice
                cheapestService = i
           Else empty
           End If
       Else empty
       End If
End While
If (cheapestService >= 0)
       invoke(services[i], ‘‘reserve_place’’, (now,1H), true)
Else empty
End If
```

Figure 3.41: A Sample Pseudo-SMoL Code

```
MAIN = getServices!currentGroup, carParkContract, myCredentials?services
       → SEARCH(null,MAXINT,services) → SKIP

SEARCH(cheapestService,cheapestPrice,services) =
   IF (services = <>) THEN RESERVE(cheapestService)
   ELSE
        invoke!head(services), ‘‘get_free_places’’, null, true?freePlaces
        IF (freePlaces > 0 ) THEN
            invoke?head(services), ‘‘get_prices’’, null, true?currentPrice
            IF (currentPrice < cheapestPrice)
                SEARCH(tail(services),currentPrice,tail(services))
            ELSE SEARCH(cheapestService,cheapestPrice,tail(services))
        ELSE SEARCH(cheapestService,cheapestPrice,tail(services))

RESERVE(cheapestService) =
   IF (cheapestService != null) THEN invoke!cheapestService, ‘‘reserve_place’’, (now,1H), true)
   ELSE SKIP
```

Figure 3.42: A Sample CSP Code

Chapter 4

SMoL Translator

The purpose of our internship in the University of Pisa (Italy) was the conception of a translator from SMEPP Modeling Language (SMoL) to Java[1]. This chapter introduces the specification of this project, the goals of a SMoL translator and constraints on it. Then, we will see how SMoL structures are translated in Java. Some of them cannot be directly translated in Java constructions and so, require further development. Finally, we will see how the translator is designed.

4.1 Specifications

As introduced in the Section 3.2, the SMEPP project has developed an abstract language called SMoL (SMEPP Modeling Language). This language describes in an abstract way, observable behaviours of entities communicating on the SMEPP network. As explained in Subsection 2.2.2, services are described through contracts. In these contract, the last part, written in SMoL concerns the service behaviour. Peers are not described through contracts, but their behaviours can be described in an independent (not used by the SMEPP middleware) SMoL file.

A SMoL2Java translator must be able to receive any contract, or behaviour file and translates it to Java. The translator cannot make assumptions on the way in which the SMEPP API is implemented. The translated code must be as close as possible to the final code, requiring few modifications to be compiled. During our internship, a **SMoL parser** has been imagined and implemented. Even if we have participated to this project, the SMoL parser won't be mentioned in this thesis seeing that its conception is quite simple. Indeed, it has been automatically generated from the description of SMoL (i.e., from XSD files describing SMoL). Thanks to the SMoL parser, elements of a contract or behaviour file (in the case of a peer) can be simply retrieved by high level functions. In the rest of this chapter, the way XML elements are retrieved will be out of concern.

Appendix C.2 (page 170) contains SMoL codes of an application imagined in the Appendix C written in a pseudo-SMoL language. This application is divided into three parts: the operator peer, the equipment peer and its monitoring service. The two first peers are described through behaviour files (see Figure C.2.2 and Figure C.2.1). The monitoring service is described by the contract displayed in figure C.1.2 (page 173). These are examples of possible codes which must be automatically translated.

4.2 SMoL Translation

Before seeing how the translator is designed, we must see how SMoL codes can be translated to Java. We begin by defining how we can translate most basic SMoL elements: expressions. Then, we will see how we can conceive SMoL execution flows by introducing exceptions. Finally, we can design commands translation.

[1] Java is the language chosen by the SMEPP project to implement the SMEPP API and other parts of the project.

4.2.1 Expressions Management

Like other programming languages, SMoL uses left and right expressions. A right expression is a value, or a reference to a value. Left expressions refer to a "container" able to receive right expressions.

SMoL Variables

SMoL variables are used by both left and right expressions. A **SMoL variable** has a name, an optional initial value (a right expression) and eventually a type (Figure D.1, page 205). During our internship, we have decided to introduce scopes. Each command can be associated with variables usable by the command and its inner commands (in case of structured commands). All variables must be declared just before the associated command. For example, in lines 4-27 of Figure C.2.1 (page 170), a list of variables associated with the `Sequence` command are declared. Their scopes are the entire peer behaviour since their are associated with the command including every other command. They are said to be the global variables.

Moreover, handlers (`Pick` branches and `InformationHandler` branches) can also declare their own variables. These local variables are placed first inside the handler. In Figure C.1.2 (page 174) in line 43-46, two variables are declared. These variables have a scope limited to the `ReceiveMessage` handler.

Variable types *must* refer to `JavaTypes` declared in the signature part of a contract (see Figure D.1, page 206). This part is only present in service contracts. So, **variables** used in **behaviour files** (files describing peers behaviours) are **untyped**. Variables used in service contracts can be untyped. A `JavaType` has a *unique name* and is associated with a *Java type* (`class` attribute). For example, a `JavaType` named "String" is associated with `java.lang.String` where type "String" can be used to type a variable.

Left Expression

As described in Figure D.1 (page 206), a left expression called "**tTo**" is either a variable and an optional part, or a query. Unfortunately, XSD description file doesn't restrict the usage of a combination of a variable and a query. So, if a variable and a query are specified, we consider only the variable and not the query. In the scope of this translator, only XPath queries are considered (`queryLanguage = ‘‘http://www.w3.org/TR/xpath’’`).

SMoL variables must be translated to Java variables. One SMoL variable is translated to one Java variable having the same name and placed in the same scope as the SMoL variable. A left expression is a reference to a container, if this expression is a variable, the reference is the variable address. In Java, variables used in left expressions are simply referred by their names. Translating a SMoL variable contained in a left expression is simply giving the variable name (without any prefix like "this."). SMoL variable parts are translated to Java as object fields.
For example, in Figure C.1.2 (page 180) line 131, the left expression will be translated to `temperature`. If there was a variable part called "`patient`": `variable=‘‘temperature’’ part=‘‘patient’’`, the translated expression would be `temperature.patient`.

XPath queries can also be used in left expressions. They must only contain a variable and an optional part. An XPath variable is prefixed by $ and followed by the variable name, optionally followed by "." and the variable part. The translation process is the same as explained in the previous paragraph. For example, an XPath query "`$temperature.patient`" is translated to `temperature.patient`. The translator checks that for each XPath variable, a corresponding SMoL variable (having the same name) has been declared in the XPath expression scope. As we can see, the XPath language is simplified because all others XPath constructions are prohibited (e.g., XML nodes referencing, XML nodes manipulating, etc.).

Right Expression

A right expression is a value, or a reference to a value. Right expressions are called "**tFrom**" (see Figure D.1 on page 207) is either:

- a variable and optional part,
- an opaque value,
- a literal value,
- or a query.

Unfortunately, XSD description file doesn't restrict the coexistence of the upper right expression types. The upper lexical order defines the priority of these types (each following itemizes has a lower priority). So, if a variable (placed first in the itemize list) is present and a query (the last one), the translator must consider only the variable (and its optional part). In the scope of this translator, only XPath queries are considered (`queryLanguage = ‘‘http://www.w3.org/TR/xpath’’`).

An `opaque` value serves to hide a value. It allows applications not to reveal how they initialize variable. So, they can hide internal data management (for security reasons, or in order to abstractly describe behaviours). A translated `opaque` value is simply the `null` value. For example, Figure C.2.2 (page 177) line 7 is translated to C.3.2 (page 189) line 27.

A `literal` is a value which must be directly translated without any manipulation on it. The translator must simply make a textual copy of its value to Java. Figure C.2.2 (page 177) in lines 8-10 contains a literal value translated in 25-26 of figure C.3.2 (page 189).

Variables used in a right expression contain references to the variable value. In Java, in the context of a right expression, variables values are simply obtained by mentioning their variable names. As previously said, a SMoL variable is translated to one Java variable having the same name and placed in the same scope of the SMoL variable. So, the translation of a SMoL variable contained in a right expression is simply the variable name (without any prefix like "this."). SMoL variable parts are translated to Java as object fields; their translations are the variable part name prefixed by ".". For example, in Figure C.2.2 line 31, `credentials` is a variable used in a right expression. The value of this expression is the value of the variable called `credentials`. This expression is translated to: `credentials` which is a reference to the value of the Java variable called `credentials` (see Figure C.3.2, line 21). If a there was a variable part called "`newPeer`" (`variable=‘‘credentials’’ part=‘‘newPeer’’`), the translated expression would be `credentials.newPeer`.

Finally, right expressions can be expressed by queries. Currently, the only recognized query language is Xpath 1.0[W3C]. Fortunately Java provides tools to evaluate the value of this kind of expressions by providing the class `javax.xml.xpath.XPath`. Note that, the following usage of XPath is limited. As previously explained, XPath expressions can use variables. An XPath variable is declared by means of "$". For example, `$temperature` is an XPath variable called "`temperature`". A variable part could be also defined with the usage of ".". For instance, `$temperature.zone1` is a variable part called "`zone1`" of the XPath variable called "`temperature`". The translator checks that for each XPath variable, a corresponding SMoL variable (having the same name) has been declared in the XPath expression's scope. Different scenarios exist:

- If an XPath expression only contains a variable and an optional part, this expression is equivalent to the attribute `variable=‘‘’’` and its optional attribute `part=‘‘’’`.
- If a right XPath expression contains several variables, its translation is simply the mapping between XPath variables to their corresponding Java references. For example, the following right XPath expression: `count(($temperature.patient1,$temperature.patient2))` is translated to `count((temperature.patient1,temperature.patient2))`.
- If no variable are used, XPath right expressions are evaluated thanks to `javax.xml.xpath.XPathFactory.newInstance().newXPath().compile(query).evaluate(node,`

javax.xml.xpath.XPathConstants.STRING). XPath evaluations are simply translation results. For example, concat('XPath ','is ','FUN!') is translated to 'XPath is FUN!'.

Assign

Once we have left and right expressions, we can use them in assignments. SMoL provides the Assign command to assign values to variables. It is similar to any assignment in a classic programming language except that, in this case, one Assign can do several assignments at once. Each assignment is defined inside a copy clause. An assignment copies the value of the source from into the target to.

Command signature.

```
Assign
        Copy
              from
              to
        End Copy
        ...
        Copy
              from
              to
        End Copy
```

Translating an Assign is very simple, Copy sections are translated sequentially, in lexical order. A Copy clause is translated to a Java assignment where the left part of this assignment is the translation of to (a left expression) and the right part, the translation of from (a right expression). For example, in Figure C.1.2 (page 178) line 39-44, a single copy clause is present. The generated code will be:
groupSequiTel = groupsSequiTel[0];

Variable Initialization

Initialization can be conceive as an Assign where to is the initialized variable and from is the init variable section. So, a variable initialization is translated in the same way as the Assign SMoL command.

4.2.2 Exceptions Handling

As described in Section 3.2, SMoL allows programmers to raise their own exceptions thanks to the basic command throw. Exceptions can also be thrown by the reply primitive as a result of an operation invocation. These exceptions are called “user exception”. The SMEPP middleware is also able to throw exceptions. These exceptions are called “middleware exceptions”. Table 3.1 (page 33) summarizes each of them. Programmers have to handle thrown exceptions through the usage of FaultHandler.

Java provides a mechanism to handle exceptions called “try-catch” (see Figure D.2, page 208). A Java try-catch is basically composed of two parts: a body and a list of handlers. The body is encapsulated in the try part. The body contains a Java code that might throw exceptions that catches handle. If an exception occurred, the execution of the try part is stopped. Then, that exception is handled by maximum one catch listed in the second part of the try-catch. The first catch matching an exception thrown by the try-block starts the execution of the catch command instead of executing the try code. If an exception occurs, the try-catch terminates its execution once the catch command finishes. If no exception is thrown by the try code, the try-catch terminates when the main command (i.e., the try code) finishes. In that case, no catch handler is executed. A thrown exception matches a catch if it's type is a subtype (or the same type) of the Java type declared by the catch. Once the exception caught, a local variable (the scope of this variable is the catch command), having the name declared after the exception type, is created. For example in Figure D.2 (page 208), if the mainCommand throws ExceptionType2, there are two possibilities. Firstly, if ExceptionType2 is a subtype of ExceptionType, the first catch matches so, command1 is executed instead of terminating the execution of mainCommand.

Exceptions caught by the first catch can be referred by the local variable called `nameObject`. Secondly, if `ExceptionType2` is not a subtype of `ExceptionType`, only the second catch matches this exception.

But there are differences between the Java try-catch and the SMoL `FaultHandler`:

- A SMoL catch can initialize a local variable containing data associated to the thrown exception. This problem could be easily solved if each translated SMoL exception is a Java exception containing a field which is a reference to exception data. Once an exception is caught the first instruction of the catch body is an assignment of a local variable to the caught exception field.

- The SMoL `catchAll` could be easily implemented to a simple Java catch. This catch is associated with the most common type among translated SMoL exception. We can define a super type called `SMEPPException` which will be the super type of each translated SMoL exception. This is an implementation choice, a `catchAll` *handles only SMEPP exceptions* and not all Java exceptions.

- `catchAll` is able to initialize global or local variables containing the current exception name and its data. We can solve this problem as previously. Instead of that, the `catchAll` body can initialize local or global variables initialized thanks to fields of caught exceptions containing the exception name and its data.

- Java catch matching is based on types matching. SMoL matching process is only based on the exception name. The problem could be solved if all translated SMoL exception are Java exception having the same hierarchy type level.

Figure D.6 (page 208) illustrates the hierarchy among translated SMoL exception. All exceptions mentioned in a SMoL behaviour are translated to exceptions inheriting from `SMEPPException`. User exceptions inherit from `ApplicationException`. Exceptions thrown by the middleware are translated to Java exceptions inheriting from `MiddlewareException`. Figure shows `ExpiredTimeout`, a middleware exception. `MeasuredTemperatureError` is a translated SMoL user exception (see below). These exceptions belong to the same level (third level) of the Java type hierarchy.

`throw (faultName, faultData?).` This primitive is simply implemented as a Java throw: `throw new faultName(faultData?)`. Each fault name (i.e., exception name) leads to the creation of a Java exception having the name `faultName`. Note that users are not allowed to throw middleware exceptions. The optional `faultData` is used by the exception constructor.

`faultData? catch (faultName).` A SMoL catch corresponds to a Java catch: `catch(faultName temporaryVariable){ faultData = temporaryVariable.getFaultData()}`. The SMoL catch body is translated inside the Java catch body.

`faultVariable? catchAll().` Note that, there is a difference between the `catchAll` described in the pseudo-SMoL language and its SMoL description. From now, we consider only the SMoL definition. `catchAll` returns a fault variable containing the fault name and its data. This command is translated to a Java catch handling `SMEPPException`. The `faultVariable` is not returned by the Java catch, but initialized in the Java catch body thanks to the reference to the caught exception: `catch(SMEPPException temporaryVariable){ faultVariable = temporaryVariable}`.

`FaultHandler.` A Java try-catch is created where main command of the `FaultHandler` is translated to Java inside the try-block. Each SMoL catch is translated (see above) sequentially, in lexical order.

Figure D.7 (page 209) contains SMoL codes: a user SMoL exception called `measuredTemperature` is raised by `throw` which is the main command of a `FaultHandler`. This exception is associated with data contained in the variable `measuredTemperature`. The first `catch` is able to handle `measuredTemperatureError` exceptions. The `catch` body initializes the variable called `uncorrectMeasure` to data associated to the caught exception. A `catchAll` is also present. Commands of

these handlers ignore the fault i.e., `catch`/`catchAll` commands are simply composed of `empty`. Figure D.8 (page 209) contains the translated code. `exception27940` and `exception21345` are temporary local variable used to referred the caught exception. Each user exception is translated to a Java exception. So, `measuredTemperature` has leaded to a Java class called `measuredTemperature` displayed in Figure D.9 (page 209). Its constructor can be called by providing data which can be retrieve later thanks to `getFaultData()`.

4.2.3 Structured Command Translation

Structured commands enclose one or more commands (basic or structured) which are orchestrated by these commands. Some of them, `Flow`, `InformationHandler` and `Pick` require more complex Java constructions. Once these complex structures are implemented, we can see how basic instructions (SMoL basic command and primitives) can be used inside them.

Flow

Command signature.

```
Flow
    command1
    ...
    commandN
End Flow
```

Command translation. `Flow` introduces concurrent execution, each command of a `Flow` is executed in a parallel way. This command finishes when all its commands have finished their execution. If a `Flow` branch (i.e. an inner command) throws an exception, other children commands are stopped.

We can implement concurrent execution in Java thanks to the Thread class. We have to create a thread per flow branch allowing them to be executed in a parallel way. `Flow` is executed inside an independent thread executing `Flow` command. So, if a `Flow` is composed of three branches, there are three plus one threads. When an exception is thrown in a branch command, other branches must stop and the exception must be thrown out of the `Flow` which is executed by an independent thread. Unfortunately, Java exceptions raised by threads are not propagated to creators. So if a certain thread X creates a thread Y, exception which are not caught inside Y are not raised inside X. It also impossible to raise an exception inside the execution flow of a remote thread. These restrictions imply that we have to find a mechanism to propagate exceptions raised by `Flow` branches.

We can easily solved exception forwarding; each `Flow` branch command can be executed inside a try-catch. When an exception is thrown inside them, they are caught and notified to the `Flow` by a simple method provided by `Flow`. At this time, the `Flow` construction is waiting for its branch endings. Once a notification is received from one of its branch, it can be awakened. Then, the `Flow` re-throws the exception caught by one of its branch. Other branches can be stopped if their periodically check whether they must stop their execution i.e., whether an exception has been caught or not.

Figure D.12 (page 211) contains the `Flow` implementation. Each command requiring concurrent execution implements `IThreadedCommand`. Classes implementing that interface can know whether they must be stopped since their contexts require their endings:

- `mustBeStopped()`. Return true if the class implementing `IThreadedCommand` must stop its execution, else it returns false.
- `stop()` notifies to a command that it must stop its execution.
- `addInnerCommand(InThreadedCommand command)`. An inner command calls this method provided by its father in order to signal that it is an inner threaded command. If the father must stop its execution, its children must also stop.

Note that each translated command is in the scope of a class implementing `IThreadedCommand`. Even the outer most command (the command which encapsulates every other commands) i.e., the main command of a behaviour must be executed inside a class implementing that interface. Indeed, every command must be able to check whether they are allowed to be executed or not. This conception implies that the Java class containing the translation of the main command must implement `IThreadedCommand`.

Figure D.2 (page 211) illustrates a basic `Flow` command composed of two `empty` commands. As we can see, each SMoL `Flow` is translated to an anonymous class extending the Java class `SMoLFlow`. Branches are translated to anonymous classes extending `SMoLFlowBranch`. This class contains a function called `command ()` which contains the translated `Flow` branch command.

SMoLFlow constructor. Firstly, the `SMoLFlow` constructor registers itself as the child of its context: `context.addInnerThreadedCommand (this)` and as the parent of its `SMoLFlowBranch` objects: `branches[i].setAssociatedFlow (this)`. After, branches commands are executed asynchronously (see `branches[i].executeCommandInBackground ()`). If an exception occurs in a branch, the thrown exception is forwarded to the parent `SMoLFlow` thanks to `forwardException`. Then all other branches are stopped. In all cases, `SMoLFlow` periodically (see `SMoL.WAITING_TIME_STEP`[2]) checks whether a branch is finished and terminates when its inner commands finishes. If an exception has been thrown by one of its branches, it is propagated outside the `SMoLFlow`.

Pick

Command signature.

```
Pick
     <callerId, input?> = receiveMessage (operationName, timeout)
          command
     ...
     output = receiveResponse (id, operationName, timeout?)
          command
     ...
     <callerId, input?> = receiveEvent (groupId?, eventName, timeout?)
          command
     ...
     wait (for?, until?)
          command
     ...
End Pick
```

Command translation. This command has a more complex implementation. Indeed, only one branch command of a `Pick` can be executed while handlers must be executed concurrently. Some difficulties appear, if a message or a response is received (we said that the corresponding branch is activated) from the middleware, the reception cannot be cancelled. So, if this kind of branch is activated, the branch command *must be executed*. `receiveEvent` and alarm handlers can be activated, but their commands are not necessarily executed. Furthermore, `receiveEvent` must be currently listening for events otherwise, events are lost. So, they cannot stop their executions. While other handlers are not obliged to be always running. So, `receiveMessage` and `receiveResponse` handlers cannot be executed concurrently; they have to be executed sequentially in a round robin with alarm handlers. We can periodically check in each round of this round robin that alarm handlers can be executed. While `receiveEvent` can be executed in background, concurrently. All handlers must have a limited execution time[3], allowing them to check whether another handler has been activated.

A SMoL `Pick` is translated to an anonymous class extending the Java class `SMoLPick`. Its branches are

[2]See the Java class: org.smepp.extendFramework.SMoL.

[3]The execution time is the minimum between `org.smepp.extendFramework.SMoL.WAITING_TIME_STEP` and the value returned by the `getDefaultTimout` method of the `SMoLPick` class, eventually restricted by the available time (see the `getTimeout` method of handlers).

also translated to anonymous classes. Their types depend on the branch type: `SMoLPickRcvEvtBranch`, `SMoLPickRcvMsgBranch`, `SMoLPickRcvResBranch`, `SMoLPickAlarmBranch`. All of these classes extend `SMoLPickBranch`. Figure D.2 (page 213) illustrates a sample translation.

SMoLPick behaviour. Figure D.16 (page 215) models the `SMolPick` behaviour. `SMoLPick` begins by notifying to its context (another threaded command, or a service or peer class) that it exists and it's an inner threaded command. Then, `SMoLPick` notifies to its branches its own Java object reference. `NoPickBranchActivable` is thrown if no branch can be activated (timeout expired, or if there is no `Pick` branch). Branches having a specified timeout (see `hasTimeout()` and `getTimeout()`), have an expiration date set at the beginning of the `SMoLPick`. Expiration dates are computed like this: now + `getTimeout()`.

In order to minimize the number of threads, `receiveEvent` branches can be grouped. Indeed, `receiveEvent` can be executed concurrently, but for each handler, a thread is required. They can be grouped and executed in a round robin inside a single thread (a thread per group). Groups size depends on the function `numberOfRcvEvtsPerThread()` provided by the `SMoLPick` class customizable by anonymous classes extending this class. For example, if a `Pick` is composed of three `receiveEvent` handlers and `Pick` is configured to create a group per two `receiveEvent` handler (the default value is one), two round-robins will be created and executed concurrently. The first round robin executes the two first `receiveEvent` handlers and the second round robin executes the last one handler. `SMoLPick` is in charge of creating groups and giving a subset of `receiveEvent` handlers to an instance of `SMoLPickRoundRobinEvents`.

Only one instance of `SMoLPickRRAlarmsRcvMsgsRes` handles `receiveMessage`, `receiveResponse` and `wait` handlers. This unique round robin instance is ran under the thread executing `SMoLPick`. But, if there is no handler of these types, the main thread can execute one events round robin (see `groups[i].execute()`) instead of executing it in a new thread (see `groups[i].executeInBackground()`). If at least one handler of these types exists, the round robin `SMoLPickRRAlarmsRcvMsgsRes` is executed.

After the ending of `SMoLPickRRAlarmsRcvMsgsRes`, or `SMoLPickRoundRobinEvents` (if there are only `receiveEvent` handlers), `SMoLPick` checks periodically[4] that the activated branch is finished (there may be currently no activated branch). It waits only if it is not forced to stop, or if there is still an executable branch. If an exception has been thrown by the command, or by an handler, it is thrown outside the `SMoLPick`. If no branch has been activated, it means that all branches are disabled and `NoPickBranchActivable` is thrown.

SMoLPickRRAlarmsRcvMsgsRes behaviour. The round robin for `receiveMessage`, `receiveResponse` and alarm handlers is displayed in Figure D.18 (page 217). Firstly, alarm branches are predictable, we can easily compute when their commands must start. Indeed, the `Until` parameter contains the date of this moment. While the `For` parameter, now + `For` defines the moment of the command execution. By computing activation dates, we are able to know the first alarm handler command (referred by `alarmBranch`) which must start. So, we can ignore other handlers.

The round robin is executed while the `SMoLPick` must not stop (since an exception occurred in its context), while at least one of its handlers is enabled and if there is no `receiveEvent` handler activated. First, the round-robin checks that the alarm branch can start its execution. If the alarm branch has reached its activation date, the lock is asked. If no other branch is activated, the alarm branch command can start its execution. In all the cases, the lock is released.

Secondly, if the alarm branch command cannot start, if there is no `receiveMessage` and `receiveResponse`, the round robin awaits the alarm branch activation date. Otherwise, it tries the next `receiveMessage`, or `receiveResponse` still activated i.e., member of `rcvMsgsResEnabled`. Before executing the handler, the round robin asks for the lock. This avoids that another handler command starts its execution at the same time. Indeed, once a message or response is received, we have to

[4]See constant: org.smepp.extendFramework.SMoL.WAITING_TIME_STEP.

be sure that its associated handler command will be executed. If a message, or response is received (`ExpiredTimeout` not thrown), the branch is activated and its command can start only if no exception has been raised. Otherwise, the exception is forwarded to the `SMoLPick`.

SMoLPickRoundRobinEvents behaviour. Figure D.17 (page 216) illustrates the behaviour of the `receiveEvent` round-robin. A subset of `receiveEvent Pick` branches is given to each instance of `SMoLPickRoundRobinEvents`. The round robin is executed while it must not be stopped: no exception occurred in the `Pick` context, no branch command is running. The purpose of this round-robin is to execute in each round, one after another, a `receiveEvent` contained in the `rcvEvetsEnabled` set. This set contains all the handlers still enabled. Once an handler becomes disabled (`rcvEvtsEnabled.isEnabled()` = false), it is removed from `rcvEvetsEnabled`.

Programmers can specify a `timeout` defining a limit to the `receiveEvent` life time. If a `timeout` is specified (`hasTimeout()` returns true), the corresponding expiration date is computed before starting the round-robin. Once the computed date is reached, the handler becomes disabled and it's removed from `rcvEvtsEnabled` set. In all the cases, `receiveEvent` is executed several times. Each execution has a limited execution time defined by `getDefaultTimeout` (restricted by the expiration date if the `receiveEvent` has a timeout and by the value of `SMoL.WAITING_TIME_STEP`[5]). So, at each loop, a new `receiveEvent` is executed during `getDefaultTimeout` milli-seconds (or less if the expiration date is reached).

If an event has been retrieved by the primitive, the round-robin checks that there is no other `Pick` branch activated before executing the handler command. As illustrated, `receiveEvent` round robin asks the lock only if an event has been received. In contrast with the previous round robin, this round robin asks the lock before trying to receive a message or a result, but in the case of an alarm, the previous round robin asks the lock once the activation date of an alarm branch is reached i.e., before executing the handler command. The `receiveEvent` round robin finishes if a branch is activated, or if there is no additional enabled `receiveEvent` handler, or if the `Pick` must stop. If an handler is the first to be activated, the handler command start is execution and other handlers are stopped. Of course, this command is not executed if an exception (different from `ExpiredTimeout`) has been thrown during the execution of `receiveEvent`. In that case, the exception must be thrown out of the `SMoLPick`.

InformationHandler

Command signature.

```
InformationHandler
      command

      <callerId, input?> = receiveMessage (operationName)
           command
      ...
      output = receiveResponse (id, operationName, timeout?)
           command
      ...
      <callerId, input?> = receiveEvent (groupId?, eventName)
           command
      ...
      wait (for?, until?, repeatEvery?)
           command
      ...
End InformationHandler
```

Command translation. `InformationHandler` receives and processes invocations, replies and events as long as the main command, lexically placed first inside the `InformationHandler`, is running. The main command translation is contained in a function called `command()` of an anonymous class extending

[5]See the Java class: org.smepp.extendFramework.SMoL.

`SMoLIHMainCommand`. Handlers are also translated to anonymous Java classes according to their types: `SMoLIHRcvEvtBranch`, `SMoLIHAlarmBranch`, `SMoLIHRcvMsgBranch`, `SMoLIHRcvResBranch`. A sample translation is displayed in Figure D.2 (page 220). Note that the `empty` command is translated to ";".

SMoLIH constructor. Figure D.23 (page 223) models the `SMoLIH` constructor behaviour. First, handlers don't have the reference of the `SMoLIH` object encapsulating them. So, its reference is given to handlers thanks to: `setAssociatedInformationHandler`. Then, the current context (another threaded command, or a service or peer class) is notified that an inner threaded command exist (see `addInnerThreadedCommand`). As explained in the SMoL `Pick` implementation, `receiveMessage`, `receiveResponse` and alarm handlers are executed inside a single round robin instance contained in the `execute` method of `SMoLIHRRAlarmsRcvMsgsRes`. A round-robin executes sequentially these kinds of handlers. Indeed, once received a message or a response cannot be canceled, so the associated handler command must be executed. So, they cannot be executed in a parallel way, but sequentially. Alarms handlers don't require parallel executions. At each round, we can check whether their activation date are reached. `SMoLIHRRAlarmsRcvMsgsRes` is consider as "the main round robin".

`receiveEvent` can be cancelled. So, once an event is received, it can be ignored, avoiding to start the associated handler command. Moreover, events are not received if the entity is not *currently* listening. `receiveEvent` must be executed in a parallel way. But, in order to minimize the number of threads (one thread per handler), we can group them. `numberOfBranchesPerThread()` defines the number of `receiveEvent` handlers executed sequentially in a round-robin (see `SMoLIHRcvEvtsGroup`). Of course, if groups are composed by more than one handler, application may suffer from events loosing.

Normally, the round robin for `receiveMessage`, `receiveResponse` and alarm handlers contained in the method `execute` of the `SMoLIHRRAlarmsRcvMsgsRes` instance, runs under the thread executing the `SMoLIH` constructor called the "main thread". The main command (placed first in the `InformationHandler`), translated in the `command()` function of `SMoLIHMainCommand`, is executed by another thread (see `mainCommand.executeInBackground`) in background. But, if no handler is present, the main command is executed by the current thread (see `mainCommand.execute()`).

Each instance of `SMoLIHRcvEvtsGroup` handling a subset of `receiveEvent` handlers is executed in background, inside a new thread (see `groups[i].executeInBackground()`) expect the last instance which is executed by the current thread if no `receiveMessage`, `receiveResponse` and alarm handler is present. So, if there is no handler, the main command is executed by the current thread. If there are only `receiveEvent` handlers, the last `receiveEvent` round robin is ran under the current thread.

After that, `SMoLIH` constructor waits[6] for the termination of all handlers command executions. It also waits for the main command only if it is executed by another thread i.e., if there is at least one handler. If an exception has been thrown, it's re-thrown.

SMoLIHRRAlarmsRcvMsgsRes behaviour. The round robin for `receiveMessage`, `receiveResponse` and alarm handlers see Figure D.23, page 223) starts by setting expiration dates. An handler has an expiration date if a `timeout` value is given to the `receiveMessage`, or to the `receiveResponse` primitive (see `hasTimeout()` and `getTimeout()`). An expiration date is computed like this: now + `timeout`. Once an expiration date reached, the handler becomes disabled and it's removed from the set containing enabled handlers.

Alarm handlers have also expiration dates only if they are not repeat every alarms (see `isRepeatEvery()`) i.e., the parameter `RepeatEvery` of `wait` is not specified. If the parameter `For` is specified, the expiration date is: now + `For`. Otherwise, `Until` is specified and the expiration date is: `Until`. If `RepeatEvery` is given, they are infinite activation dates. The first activation date is obtained as previously described (according to `For` and `Until`), others are obtained by adding `RepeatEvery` to the previous activation date. So, we get: (first activation date + `RepeatEvery`), (first activation date + `RepeatEvery` + `RepeatEvery`), ...

[6]See the join method provided by the Java Thread class.

The round robin is executed while the `SMoLIH` is not forced stop (the context requires its ending), while there is still at least one enabled `receiveMessage`, or `receiveResponse`, or alarm handler and while the main command has not finished its execution. The round robin starts by checking whether an alarm branch has reached its activation date (`nextAlarmBranch.canBeExecuted`). If an activation date is reached, the handler command is executed in background expect if the handler command will be not executed again in the future (not a repeat every alarm) and only if there is no other handler executed by the round robin. In these conditions, the alarm command is executed by the current thread (because it has nothing else to do). Alarm branches which have no more activation date, are removed from the set containing enabled alarms.

If no alarm command can start its execution, the round robin searches for a `receiveMessage`, or `receiveResponse` handler to execute. If the next handler member of `rcvMsgsResEnabled` is disabled, it's removed from this set. Once an enabled handler found, the round robin executes the associated `receiveMessage`, or `receiveResponse` during a certain interval of time (see `IH.getDefaultTimeout()`) restricted by the available time (if a `timeout` is specified) and by the value of `SMoL.WAITING_TIME_STEP`[6]. If `ExpiredTimeout` is not thrown, or any other exception, it means that a message or a response has been received and the handler command is executed in background. If no handler command can start, the round robin waits for a certain moment (defined by `SMoL.WAITING_TIME_STEP`[7]).

SMoLIHRcvEvtsGroup behaviour. The following description is related to Figure D.24 (page 224) and concerns the behaviour of the `execute` method provided by instances of `SMoLIHRcvEvtsGroup`. As stated above, each events round robin receives a subset of `receiveEvent` handlers. Expiration dates of these handlers are initialized at the beginning (only if a `timeout` value is specified by `receiveEvent`). A set called `subsetRcvEvtsEnabled` contains handlers still enabled. Before executing the round-robin, all handlers are members of this set. Expiration dates are computed in this way: now + `timeout`. Disabled handlers are removed from `subsetRcvEvtsEnabled`.

The round robin takes the next handler in `subsetRcvEvtsEnabled` and execute its associated `receiveEvent` during a certain interval of time (see `IH.getDefaultTimeout()`) restricted by the available time (if a `timeout` is specified) and by the value of `SMoL.WAITING_TIME_STEP`[6]. If an event has been received (`ExpiredTimeout` is not thrown), the handler command starts. If another kind of exceptions has been raised, it's notified to the associated `SMoLIH` instance (`IH.forwardException`).

While

Command signature.

```
While booleanCondition
      command
End While
```

Command translation. `While` is translated to the Java `while`. The body of the Java `while` contains the translated `command`. The Java `while` condition is simply the translated `booleanCondition`.

RepeatUntil

Command signature.

```
RepeatUntil booleanCondition
      command
End RepeatUntil
```

Command translation. `RepeatUntil` is also easily translated to Java. Its Java counterpart is the `do-while` structure. The body of the `do-while` command contains the translated `command`. The command guard is simply the translated `booleanCondition`.

[7]See the Java class: org.smepp.extendFramework.SMoL.

If-Then-Else

Command signature.

```
If booleanCondition
   command
Else
   command
End If
```

Command translation. The SMoL `if-then-else` is the same as the Java `if-then-else`, where SMoL commands are translated in their counterparts. The command guard is the translation of the boolean condition (see `booleanCondition`).

Sequence

Command signature.

```
Sequence
      command1
      ...
      commandn
End Sequence
```

Command translation. `Sequence` commands are translated sequentially and placed in the same lexical order. After each command translation, a test must be placed checking that the current context must not stop its execution (see `IThreadedCommand`). If the test is successful, the current execution flow must end and following `Sequence` commands are not executed.

4.2.4 Basic Commands

A basic command is either a SMEPP primitive, or a SMoL basic command. Basic commands can be used inside structured commands, or as the main command of a peer or service.

empty

Command signature.

```
void empty ( )
```

Command translation. This command can be simply translated to Java to "`;`".

waitfor

Command signature.

```
void wait (for?, until?, repeatEvery ?)
```

Command translation. `wait` can be used in two different contexts: as an alarm branch (see `Pick` and `InformationHandler`) or used as a simple command like other commands. In the two following implementations, we are only interested by the *the second context*. Programs call `wait` to delay their execution either for a certain time (specified by the `for` parameter), or until a certain moment (defined by the `until` parameter); these both parameters cannot be used simultaneously. The `repeatEvery` argument can only be used in the context of an `InformationHandler` to define an alarm branch. All parameters are specified thanks to XPath[W3C]. In the following implementation, we consider that `for` is specified: `void wait (for)`.

The `for` parameter is expressed in XPath[W3C] and contains a duration (like the `repeatEvery` parameter). For example, `wait(''P3DT10H'')` stands for "wait for three days and ten hours". Java provides functions[8] allowing programmers to express XPath durations in milliseconds.

The `timeToReach` parameter is the date when the `wait` ends its execution. Before reaching this date, the caller checks every `SMoL.WAITING_TIME_STEP`[9] whether it must stop to wait, this duration is limited by the available time (if `timeToReach` will be reached). For example, if `SMoL.WAITING_TIME_STEP` is set to 3 milliseconds, the current time is 15 and `timeToReach` is 17, the sleeping time is: min(3, (17-15)) = min(3, 2) = 2 milliseconds. A `wait` must end before `timeToReach` if its context requires its termination (e.g., an exception occurred in another execution flow). Note that the variable named "caller" is a reference to the context of the command.

```
waitFor (caller,forDuration) {
    var timeToReach = getMilliseconds(forDuration) + currentDate
    var nowTime = currentDate
    While ( !  caller.mustBeStopped() AND (nowTime < timeToReach) ) {
        sleep( min(SMoL.WAITING_TIME_STEP,(timeToReach-nowTime)) )
        nowTime = currentDate
    }
}
```

Figure 4.1: waitFor Pseudo-Implementation

waitUntil

The following function implements `wait` where only `until` is specified. The parameter format is a date (at least a year, a month and a day must be specified), for example: `wait(''2007-06-30'')` means "wait until June 30, 2007 at 00:00".

XPath date can be parsed thanks to the Java class `DataFactory`[10]. This functions returns the timestamp of the date specified by `until`. The returned value is assigned to `timeValue`. Every `SMoL.WAITING_TIME_STEP`[4] milliseconds the function checks whether it must stop its execution seeing that the current context of `wait` requires its ending. `SMoL.WAITING_TIME_STEP` can be limited if `timeValue` is reached before `SMoL.WAITING_TIME_STEP` milliseconds.

```
waitUntil (caller,untilTime) {
    var until timeValue = getDate(untilTime)
    var nowTime = currentDate
    While ( !  caller.mustBeStopped AND (nowTime <untilTimeValue) ) {
        sleep ( Math.min( SMoL.WAITING_TIME_STEP, (untilTimeValue - nowTime) ) );
    }
}
```

Figure 4.2: waitUntil Pseudo-Implementation

exit

Command signature.

```
void exit ( )
```

[8]javax.xml.datatype.DatatypeFactory.newInstance().newDuration(duration).getTimeInMillis()
[9]See the Java class: org.smepp.extendFramework.SMoL
[10]javax.xml.datatype.DatatypeFactory.newInstance().newXMLGregorianCalendar(until).toGregorianCalendar().getTimeInMillis()

Command description. `exit` can be used in a service code or in a peer behaviour. In the first case, calling this command terminates the service. In this case, `exit` can be implemented by throwing a Java exception called `ThreadDeath`. This exception permits to stop all execution flows an directly stops the entire service class execution. Of course, programmers cannot catch themselves this kind of exceptions.

In the second case, the peer finishes its execution and all of its services are stopped. The SMEPP API provides a Java function implementing that primitive: `leaveSMEPP`.

API Primitives Translation

API primitives don't require any particular implementation; they must be simply translated to Java function calls on methods provided by the Java object providing the API. Currently, there are two objects implementing the SMEPP API: `org.smepp.api.PeerManager` and `org.smepp.api.ServiceManager`. An instance of the `PeerManager` is returned when an application calls `newPeer` (a static function). In the case of services, service classes must be `Runnable` classes and their constructors must have the following signature: `(ServiceManager svc, Serializable constrValue)`. So, services access to the API thanks to the `ServiceManager` instance given at the publication time when a new instance of the service class is created.

SMEPP primitives arguments are right expressions and translated as previously described. SMEPP primitives results are left expressions and are used in an assignment where the left part is the result and the right assignment part is the call to the Java function implementing the primitive. For example, in Figure C.2.2 (page 177) line 31-35, there are two arguments: `groupDescr` and `credentials`. Their translations are: `groupSequiTelDescription` and `myCredentials`. The primitive result must be assigned to the variable `groupSequiTel`, its translation is: `groupSequiTel`. The primitive translation is: `groupSequiTel = obj.createGroup(groupSequiTelDescription, myCredentials)` where `obj` is either an instance of `org.smepp.api.PeerManager`, or an instance of `org.smepp.api.ServiceManager`.

4.3 Translator Architecture

Previously, we saw how SMoL structures can be translated to Java. This section explains how the translator is able to perform translations. The SMoL translator is divided into two parts. There are a class translator for service contracts and another for behaviours translation. Both translators share common parts.

First, we begin by illustrating these common parts. Both translators use Java classes (see Figure E.1, page 226) abstracting the creation of Java codes. The first class, `JavaClassTranslator` provides high level functions allowing to manipulate generated Java classes. Each instance of `JavaClassTranslator` represents a generated Java class. Thanks to this class, the generated code can be easily manipulated. For example, changing the package of the class is achieved by calling `setPackageName`. When the class is ready to be written on the hard drive, a simple call to `writeClass` writes the generated Java classes on the hard drive. Generated Java methods are also modeled, an instance of `JavaMethodTranslator` represents the code of a Java method. We can add or remove methods from the generated Java class by calling `addClassMethod` or `removeClassMethod` on the associated `JavaClassTranslator`. Finally, the Java class `JavaClassWritter` provides a simple mechanism for code indentation (append a Java code to the current indentation depth, increment the current indentation depth, . . .) and for writing a Java class on the hard drive.

Instances of the Java class `JavaAnonymousClassTranslator` (displayed in Figure E.2, page 227) represent anonymous classes declared inside a block code of a Java method. Anonymous classes are particular Java classes seeing that they have not their own class file and must be written as other instructions of a method code. These instances are used to contain the generated code of a `Flow`, a `Pick`, an `InformationHandler`, the generated code of handlers and Java declarations of SMoL variables requiring the Java class `LocalVariablesDeclaration`.

Each time a SMoL command is encountered, a new instance of its associated SMoL command translator (extending `SMoLCommandTranslator`) is created. SMoL command translators require the SMoL command to translate, the Java method containing the translated command and the optional context (see relation `SMoLFatherCommand`) of the SMoL command i.e., another command (if it's not the main command of the behaviour), or an handler encapsulating the command (see `catch/catchAll`, `InformationHandler` and `Pick`). So, instances of `SMoLCommandTranslator` form a concrete syntax tree. The concrete syntax tree is built in the first pass. In the second pass, the Java code is generated and written on the hard drive (see the method `writeProject`). Note that class Diagrams containing the SMoL command translators are not displayed in this thesis. These translators can be found in the following packages: `org.smepp.translator.sharedObjects`, `org.smepp.translator.services.translatorObjects` and `org.smepp.translator.peer.translatorObjects`.

Both translators can be easily customized by providing its own instance of the Java class `ITranslatorConfiguration` (see Figure E.3, page 228). Instances must provide parameters such as: the API object reference (currently: `peerManager` and `serviceManager`), SMoL command translators and expression queries translators.

Each translation process is associated with an instance of the Java class `TranslatorWorkspace` which is probably the most used object. This instance contains references to managers (see following Subsections). Once these managers are added to the workspace, the translator is ready to be used. The translation process begins by providing to the workspace (see the method `generateCode`) the SMoL command to be translated and the Java method containing the translated command. Moreover, the workspace contains all generated Java classes which will be written on the hard drive. Once the translation is finished, the instance of `TranslatorWorkspace` is in charge of writing Java files (see `writeProject`).

The translator behaviour is build on a two-pass process. The first pass serves to construct hierarchy of `SMoLCommandTranslator` (concrete syntax three) thanks to the `generateCode` method of the workspace. The second pass produces the Java code and write it on the hard drive (see the method `writeProject` of the workspace). A two-pass translator is necessary because we need to know the entire SMoL code before producing its translation. For example, Java variables must be declared and typed before using them. But, some types have to be inferred thanks to their using contexts. So, we need to know how untyped variables are used before declaring their Java types. A single-pass parser cannot be used.

4.3.1 Peers Translator

Figure E.4 (page 229) contains classes reserved to the peers translator. `PeersTranslator` is the class used to translate SMoL peer behaviour. The peers translator allows users to translate a SMoL file containing the behaviour, an XML node referencing the SMoL behaviour, or the textual SMoL content. In all the cases, calling one of these features requires an instance of `PeerTranslatorConfiguration`. This instance returns translator parameters such as the name of the class that contains the translated SMoL behaviour, the name of the object referencing the SMEPP API (for peers) and SMoL command translators (restricted to translators of commands used by peers).

The peers translator is a main class allowing users to directly use the translator by providing program arguments: the SMoL behaviour file location (argument called `behaviourFile`), whether Java classes are referred by their full qualified names (`fullQualifiedNames`), the location of the translated files (`locationDir`), the Java package name of generated Java class files (`packageName`, expect for exceptions) and the name of the Java peer class (`peerClassName`). The main command uses program the arguments to provide the configuration of the translator (see the Java class `DefaultPeerTranslatorConfiguration`). In that case, other parameters of the translator (the API object name and the commands translators) cannot be specified thanks to the program arguments, they are statically specified.

Peer behaviours are translated to only one Java class having the name specified by the translator configuration (see above). The behaviour code is contained in the constructor. The Java peer class implements the Java interface `org.smepp.extendFramework.ISMEPPPeerClass` extending

`IThreadedCommand`. Indeed, even the main command of a behaviour must be in the scope of an `IThreadedCommand`. A sample peer translated is present in Figure C.3.1 (page 182 - 184). We can see in Figure C.4.1 (page 193 - 195) that the translated code is close to the final code.

4.3.2 Services Translator

Figure E.5 (page 230) illustrates classes reserved to the services translator. The services translator provides the same features as the peers translator (translating an XML file, an XML node referencing a SMoL behaviour and translating a textual SMoL content). The services translator must be customized by providing: the name of the object referencing the SMEPP API containing functions reserved to services and the SMoL command translators (restricted to commands used by services). This customization is achieved by an instance of the Java class `ServiceTranslatorConfiguration`.

As the peers translator, the services translator can be used as the main class to translate contract files. In this case, users have to provide the value of the following program parameters: `contractFile` (the contract file location), `fullQualifiedNames` (whether Java classes are referred by their full qualified names), `locationDir` (the location directory of translated files) and `packageName` (the Java package name of generated Java classes, expect for exceptions). Note that the name of the Java class containing the translated service behaviour is simply the service name defined in the service contract. The main command uses the program arguments to provide the configuration of the translator (see the Java class `DefaultServiceTranslatorConfiguration`). In that case, other parameters of the translator (the API object name and the commands translators) cannot be specified thanks to the program arguments, they are statically specified.

Services are translated to only one Java class having the name of the service (as specified by the service contract). Behaviour codes are contained inside a single method called "service". Service classes implement the interface `org.smepp.extendFramework.ISMEPPServiceClass` extending `IThreadedCommand`. Indeed, as previously mentioned, even the main command of a behaviour must be in the scope of the Java class `IThreadedCommand`. A sample service translation is displayed in Figure C.3.1 (page 185 - 187). We can see in Figure C.4.1 (page 196 - 198) that the generated code requires few modifications to be operational.

4.3.3 Events Manager

This manager is only used the case of service contract translations. The manager is in charge of checking that events declared in the signature part of a contract have a unique name. Figure E.6 (page 231) contains the class Diagram of this manager. Moreover, this manager can also be used to list events raised by a service (not currently used).

4.3.4 Exceptions Manager

Exceptions manager (displayed in Figure E.7, page 231) handles exceptions thrown or caught inside entities behaviours. During our internship, we have decided to place all SMEPP middleware exceptions inside a single package called `org.smepp.api.exceptions`. While user exceptions must belong to `org.smepp.api.exceptions.SMEPPException.CUSTOM`.

The translator begins by searching for classes belonging to the package of the `SMEPPException` class (this class can move to another package without disturbing the translator). These classes are considered to be `MiddlewareException`. Of course, the translator bin-path must be configured to see middleware exceptions. Each time the translator encounters a `catch/catchAll`, it calls `registerCaughtFaultName` and gives the exception name to be catch. When a `throw` command is found, `registerThrownFaultName` is called and the name of the thrown exception is notified.

Both registration functions create an instance of `ExceptionTranslator` (extends the `JavaClassTranslator` class) only if the given exception name is not already registered. Instances of this class model the translated SMoL exception. Note that only user exceptions lead to the creation

of Java class files having the name specified by their associated instances of `ExceptionTranslator`. These classes belong to `org.smepp.api.exceptions.SMEPPException.CUSTOM` and their location on the hard drive depend on the current translator configuration. A sample translation is explained in Subsection 4.2.2.

4.3.5 Expressions Manager

Each `TranslatorWorkspace` is associated with an instance of `ExpressionsTranslator`. The purpose of this manager is the translation of left/right expressions and assignment expressions. Each kind of expression is associated with a class containing the translation: `TranslatedAssignmentExpression`, `TranslatedLeftExpression`, `TranslatedRightExpression`. The manager is in charge of calling other managers to check that variables are correctly declared, to infer variable types and to check that types used in variables declarations are declared in the signature part of the service contract (only in the case of a service translation).

As previously mentioned, the sole recognized query language is XPath [W3C]. So, `XPathExpressionTranslator` is the only class implementing `IQueryExpressionTranslator`. Adding new recognized query languages is quite simple. Each language translator must be associated with a class implementing `IQueryExpressionTranslator`. Their instances must be returned by the translator configuration when `getQueriesTranslator` is called (see `TranslatorConfiguration`).

Note that left and right expressions are always in the scope of a SMoL command (see the `SMoLCommandTranslator` class modeling a SMoL command and its translation). Only assignments of global variables during their declarations are not in the scope of a command.

4.3.6 Types Manager

Service are able to define their own SMoL types in the signature part of their contracts. Some of these types are mapped to Java; in the context of the translator, only types mapped to Java are considered. The purpose of types manager is to keep track of these mappings. Once a variable declaration must be translated to Java, the translator asks the Java type associated with the SMoL type of this variable (see `getDatatypeMapping`). Of course, the manager is useless if variables are untyped. As previously mentioned, seeing that peer behaviours are not described through contracts, all variables used in peer behaviours are untyped.

Variables Types Inferring

Programs are allowed to unspecify variables. But, in order to minimize modifications required by the generated Java code, the translator must infer their types. Indeed, variables provided as argument of a command (a SMoL command or a SMEPP primitive) can be inferred since we know the type required for the argument (arguments are right expressions). We said that we infer the type of a variable used in a right expression having a known type. The translator is also able to infer the type of variables used to assign the result of a SMEPP primitive. Indeed, we know the type of values returned by the SMEPP primitives.

The Java class `org.smepp.translator.typesManager.JavaTypeInferring` provides a mechanism for types inferring. Each untyped variable is associated with an instance of this class. Every time the variable is used in a left or right expression, a corresponding method is called to infer its type (see the method `infersTypeRightExpression` and the method `infersTypeLeftExpression` of the `JavaTypeInferring` class).

Variables Used in a Right Expression. Figure E.11 (page 234) models the method inferring the types of untyped variables used in right expressions (see the method `infersTypeRightExpression`). This method is able to infer the type of a variable used as an argument of a SMoL command, or as an argument of a SMEPP primitive. `infersTypeRightExpression` begins by identifying the array dimension of the required type and its component Java type. For example, the variable `xyz` is used as

an argument of the type `String[]`, the required type is `String[]`, the array dimension is 1 and the component type is `String`. In the following explanation, we will not mentioned the following test: each time the function `infersTypeRightExpression` or `infersTypeLeftExpression` is called, we test that the array dimension is the same as the current dimension, otherwise it is an error. If the array dimension is greater than 0, the inferred type is the component type of the array.

During the first inference of a variable type, a tree containing the possible Java types is initialized. Upper nodes of this tree are super types of lower nodes. Figure E.12 (page 235) illustrates a sample initialized tree. A variable used in a right expression must be a subtype of the right expression type. In our example, variable `x` must be a subtype of `Y`. So, its possible types are lower nodes of `Y`. Unfortunately, Java doesn't provide a mechanism allowing to know all these subtypes. So, the research area is unknown. The current inferred type is `Y` (see the Diagram circle).

In the third case of Diagram E.11 (explained in Figure E.14, page 235), the situation is quite the same as the one previously described. The inferred variable has been only used in right expressions. Again, the variable is used in a right expression. We can restrict the research area to subtypes of the current inferred type. In our example, `x` was previously used in a right expression which must be a subtype of `Y`, now we learn that `x` must be a subtype of `Z`. The current inferred type becomes `Z` and the research area is the subtypes of `Z` (`Z` is included in the research area). The research area remains unknown.

The second case of Diagram E.11 (explained in Figure E.13, page 235) is called when the research area is known. It means that the inferred variable has only be used in left expressions (see below) before being used in a right expression. The variable type must be a super type of the last left expression. Now, we know that the inferred variable type must be a subtype of the right expression type. So, the inferred type must be between these two types. In our example, the variable `x` contains an expression of the type `C` so, the inferred variable type must be a super type of `C`. Later, we learn that it must be a subtype of `E`. So, its types is between `E` and `C`. The current inferred type is the lowest subtypes in this case, it's `C` (see the Diagram circle). The research area is said to be closed.

In the last case (explained in Figure E.15, page 235), we know that the real type of our inferred variable must be between two types. In our example, it must be between `E` and its subtype `C`. The inferred variable is now used in a right expression. So, the research area (which is closed) is restricted, the real variable type is between the new type and the lowest type. In our example, the research area is between `A` and `C`. The Java type `C` remains the current inferred type.

Variables Used in a Left Expression. If a variable used in a left expression is inferred for the first time, case 1 of Diagram E.16 (page 236) is applied. In this case, we know that the inferred variable contains an expression of a certain type. So, the inferred variable type must be a super type of that type. A sample case 1 is illustrated in Figure E.17 (page 237), in this case the inferred variable `x` contains an expression of the type `C`. So, the type of `x` must be a super type of `C`. The current inferred type is `C`.

In the second case (see Figure E.18, page 237), the inferred variable has only be used in left expressions before being used again in a left expression. This case allows to eliminate certain subtypes from the research area (which is said to be known) and allows to keep only super types of the latest left expression. In our example, variable `x` contains an expression of the type `C`. Thus, the type of `x` must be a super type of `C`. Then, `x` can also contain an expression of the type `A` which is a super type of `C`. This situation implies that the real type of `x` must be a super type of `A`. The current inferred type is the lowest type in this case, it's `A`. The research area remains known.

In the third case (see Figure E.19, page 237), the inferred variable has only be used in right expressions before being used in a left expression. In our example, we knew that the inferred type was a subtype of a certain type `Y`. Now, we learn that the variable can contain an expression of the type `Z`. So, it's real type must be a super type of `Z` and a subtype of `Y`. The research are is said to be closed. The current inferred type is the lowest type, in this case it's `Z`.

In the last case (see Figure E.20, page 237), the research area is closed i.e., we know that the inferred

type is a super type of a certain type and the subtype of another. This case permit to eliminate lower super types. Indeed, now, we learn that the untyped variable is used in a left expression having a certain type. So, nodes of the current research area which are not super types of this left expression type, are removed from the research area. In our example, the inferred type of `x` is between `Y` and `Z`. Now, we learn that the untyped variable contains an expression of the type `A`. So, the type of `x` must be between `Z` and `A`. The node `A` is the lowest type and becomes the current inferred type.

4.3.7 Variables Manager

As explained in Subsection 4.2.1, SMoL commands are able to declare their own variables. In the case of a basic command, its variables can only be used by the command. In the case of a structured command, local variables are only visible from its inner commands. Variables declared by handlers (`Pick` or `InformationHandler` branch) can be used by themselves (`ReceiveMessage`, `wait`, `ReceiveResponse` and `ReceiveEvent`) and by their commands. Variables declared by a `catch/catchAll` are only visible from its main command. The following manager allows the translator to check variable usages and infer their types.

Every time a variables declaration is encountered, the method `registerLocalVariables` (variables declared by an inner command i.e., local variables) or `registerGlobalVariables` (variables declared by the main command and visible from the entire SMoL code) is called. These methods create an instance of `SMoLVariableDeclaration` associating the command declaring the variable, the variable name, its optional type and its initial value (also optional). For each declared variable, only one check is achieved: variables belonging to the same declaration section cannot have the same name. There is no restriction on variables masking. Indeed, a variable included in the scope of another variable can have the same name. In this case, the inner variable scope has priority over the other variable scope.

Later, when a SMoL variable is used in a left or in a right expression the translator must check that this variable is visible from its context (i.e., from the command using that expression). This check is achieved by the function: `isVariableDeclared`. Moreover, if an untyped variable (see `hasDefinedType`) is used in an expression of a known type, the variable type is inferred by calling: `infersTypeRightExpression` or `infersTypeLeftExpression`. Finally, translating a SMoL variable used in an expression is performed by the method `getVariableLocation`.

SMoL variables declarations are translated by calling the method `getVariableDeclaration` on each instance of `SMoLVariableDeclaration` (each SMoL variable is associated with an instance of this class). The translation of a SMoL variable declaration is obtained by calling the method `getJavaType` and the method `getInitialValue`. The first one returns the Java variable type as declared by the SMoL variable, or as inferred by the translator. The second one is used only if the variable has an initial value (see `hasInitialValue`). In that case, the method returns the translation of the value used to assign the variable.

The way SMoL variables are translated depends on the variables context:

- **Global Variables.** Global variables are declared as fields of the Service class, or of the Peer Class. For example, the main command of the Operator peer (see Figure C.2.1, page 170) is a `Sequence`. Its associated variables (i.e., global variables) are declared in line 4 to 27. We can see that they are translated as fields of the Java class `OperatorPeer` (see Figure C.3.2, page 189), line 25 to 34.

- **Flow Command Variables.** Variables associated with a SMoL `Flow` command are translated to fields of the anonymous class extending `SMoLFlowBranch` which contains the translated `Flow` command. For example, in the monitoring service provided by the SequiTel equipment (see Figure C.1.2, page 176), two local variables called (see line 105-108) `temperature` and `bloodPressure` are associated with the command `while` (the SMoL `Flow` command). These variables are translated in line 106-107 of Figure C.3.1 (page 187).

- **Handlers Variables.** Variables declared by handlers (`Pick` or `InformationHandler` branches) are translated to fields of anonymous classes implementing these handlers. For example, two handler variables called `rcvMsg` and `temperature` are declared in an `InformationHandler` displayed on Figure C.1.2 (page 174) line 43-46. These handler variables are translated in Figure C.3.1 (page 186) on line 64-65.

- Otherwise, local variables are translated to fields of an anonymous class implementing `org.smepp.extendFramework.LocalVariablesDeclaration`. One of its method called `command` contains the translation of the command declaring these local variables.

4.4 Conclusion

This Chapter aimed to describe the SMoL2Java translator created during our internship in the University of Pisa (Italy). Firstly, we have introduced opportunities, issues and the context of such a translator. Secondly, Section 4.2 has shown how it is possible to translate SMoL structures to Java. Section 4.3 has decomposed the translator architecture: the translator can be used by calling two Java classes translating service contracts, or behaviour files. These classes share the same managers handling: SMoL events, caught/thrown exceptions, expressions used in assignments or in command calls, SMoL types and SMoL variables.

Appendix C illustrates a sample application translation. This application, the SequiTel application, has been introduced in Subsection 2.2.3. Thanks to our translator, we can easily obtain a complex EP2P Java application from a SMoL application. Firstly, in Section C.1 (page 167), we begin by defining the pseudo-SMoL code of SequiTel. The code is composed of three elements. First, the equipment peer (see Figure C.1.1, page 167) is the application deployed in health-care equipments. This application provides the monitoring service described in Figure C.1.1 (page 168). Finally, the operator peer (see Figure C.1.2, page 169) is the application used by SequiTel operators.

The SequiTel application composed of three files (two behaviour files and one service contract) requires three executions of the SMoL2Java translator to be translated. These SMoL files are displayed in Section C.2 (page 170). Figure 4.3 displayed how we have generated the SequiTel application[11]. The Java code, displayed in Section C.3 (page 182), is obtained after few seconds. We can see that it is quite closed with SMoL structures even if it is longer than the SMoL application. The generated code cannot be directly compiled. Indeed, opaque values must be defined, some SMoL types has not been inferred, some context checks can be removed (see: `if ( mustBeStopped () ) return ;`), etc. The final SequiTel application is displayed in Section C.4 (page 193).

```
org.smepp.translator.peer.PeersTranslator -fullQualifiedNames false -locationDir .
                                          -behaviorFile equipmentPeer.xml
                                          -packageName com.SequiTel.equipment
                                          -peerClassName EquipmentPeer

org.smepp.translator.services.ServicesTranslator -contractFile monitoringService.xml
                                                 -fullQualifiedNames false -locationDir .
                                                 -packageName com.SequiTel.equipment.monitoring

org.smepp.translator.peer.PeersTranslator -fullQualifiedNames false -locationDir .
                                          -behaviorFile operatorPeer.xml
                                          -packageName com.SequiTel.operator
                                          -peerClassName OperatorPeer
```

Figure 4.3: How Translating SequiTel

[11]For further details on the translator arguments, please refer to Subsection E.4 and Subsection E.5

Chapter 5

Perspectives

Chapter 3 formalize the SMEPP service model. Even if it did not claim to formalize the entire service model, it suffers from simplifications and offers a particular view on the model. In a first time differences between the SMEPP service model and its B-implementation will be explained, followed by a feed back on the B-method.

The previous chapter explained how the SMoL translator implementation has met its goals. However, there is a gap between the definition of some SMoL commands and their implementations built on top of the SMEPP API which brings its own limitations. In a second time, this chapter identifies drawbacks, weaknesses of the translator and compromises we made. Finally, some suggestions about future work are drawn.

5.1 Service Model

As mentioned for the implementation of the primitives, some simplifications have been made:

- Type checking on the input/output of `invoke`, `receiveMessage` and `receiveResponse` are not performed.
- The process checking that the given credentials match another credentials has been simplified as a simple equality test, avoiding to decompose structures of credentials.
- Only the first security level in the group admission has been considered (see the `createGroup` primitive).
- Group descriptions have been considered as atomic values in place of structures containing group names and security levels.
- The matching process of service contracts has been simplified. Instead of taking into account their structures, the matching process is simply based on an equality test (see implementation of `getServices`).
- Some optional arguments have been modeled as mandatory:
  - `getGroups`: `groupDescription`,
  - `getPeers`: `groupId`,
  - `getServices`: `groupId`, `peerId`, `serviceContract` and `maxResults`,
  - `invoke`: `doReturnResult`.
- Re-publishing a service has not been implemented. But, it can be implemented by calling `unpublish` followed by the `publish` primitive.
- The `invoke` primitive called with the `doReturnResult` argument set to true doesn't comply with the SMEPP service model. Instead of returning the invocation result, `invoke` must be followed by a call to `receiveResponse` which returns the result.

```
peers,exception <-- getPeers(groupId, credentials) =
PRE
    (groupId ∈ GroupId) ∧ (credentials ∈ Credentials) ∧
THEN
     IF
        /* Optional part in the case of a service - InvalidGroupId */
        ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) ∧ (group'cred = credentials) )
     THEN
        ANY peerIdSet
        WHERE
              ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) ∧
              (group'cred = credentials) ∧ (peerIdSet = group'members) )
        THEN peers,exception ≡ peerIdSet, rec(``'',null)
     ELSE
        peers,exception ≡ ∅, rec(``InvalidGroupId'', null)
     END
END
```

Figure 5.1: getPeers Implementation With Exceptions Support

5.1.1 Unsupported Constructions

Some aspects of the SMEPP service model and SMoL cannot be directly translated in B structures. The purpose of this subsection is to highlight well-known concepts which have to be re-imagined before implementing them in a B-machine.

Notion of Time

Our B-model doesn't support the notion of time. So, the `timeout` argument is not handled by the following operations: `receiveEvent`, `invoke`, `receiveMessage` and `receiveResponse`. But, this notion can be modeled thanks to a timestamp contained in a B-variable. Passing time is equivalent to an incrementation of the timestamp variable achieved by a B-operation. Moreover, we can model the incrementation process as a process running in background of other operations thanks to the usage of CSP[MB05]. Indeed, the time passing process is a looping process executed in a parallel way (see the parallel CSP operator) with other operations calls.

Exceptions

Java exceptions have no counterpart in the B-method. Indeed, a B-operation is only able to return some values at the same time. So, exceptions must be modeled as particular values. To that end, one returned variable can be added to B-operations throwing exceptions.

The following example is one of the possible implementations. The type of the extra variable is a pair where the first element has the exception name and the second contains exception data. It is denoted: `struct(name ∈ STRING, data ∈ Data)`. After that, we must assign a particular pair value representing the situation where no exception has been thrown. For example, the couple has an empty name and null data: `rec(``'', null)`. If the value of the added variable is different from that value, we consider that an exception has been thrown (no matter other returned values). As on example, the `getPeers` primitive implemented in Figure B.2.8 (page 145) is re-implemented in Figure 5.1.

5.1.2 SMoL Commands

SMoL commands have not been modeled. In fact, the B-language is not able to provide a simple mechanism for implementing them. Indeed, a B-machine cannot specify how its operations must be used: prohibited/granted sequences of operation calls, how operations are orchestrated, etc. But, thanks to the co-usage of our B-machine and CSP[MB05], SMoL commands can be implemented as CSP processes where calls to B-operation are conceived as dialogs over communication channels. For

This is a sample FaultHandler written in pseudo-SMoL:

```
FaultHandler
     mainCommand
     ReceiveEvent(emergency)
         command1
     waitFor(100ms)
         command2
End FaultHandler
```

One of its possible implementations written in CSP (exceptions are not modeled):

```
MCMD = mainCommand → FINISH → SKIP
H1 = (receiveEvent!emergency → ( H1 ||| (command1 → SKIP) )
       | (receiveEvent!emergency → FINISH → SKIP)
       | (FINISH → SKIP)
H2 = (waitFor!100ms → ( H2 ||| (command2 → SKIP) )
       | (waitFor!100ms → FINISH → SKIP)
       | (FINISH → SKIP)
FAULTHANDLER = (MCMD [|FINISH|] H1) [|FINISH|] H2
```

Figure 5.2: Implementation of A FaultHandler in CSP

example, the SMoL `Sequence` can be implemented as sequential CSP processes. A SMoL `Flow` has a counterpart in CSP: the interleaving operator. `If-then-Else`, `While`, `RepeatUntil` can be implemented by using recursive processes and the interal choice operator. `InformationHandler` and `Pick` require the interleaving operator, the internal choice and recursive processes.

Figure 5.2 illustrates a sample `FaultHandler` composed of two handlers. Handlers are associated with CSP processes called `H1` and `H2`. An handler can be directly stopped, or it can start its execution. Once the execution of an handler is finished, its associated process ends, or its command starts while it is ready to be executed again. The main command is implemented in the CSP process called `MCMD`. This process executes the main command and then, leads to the `FaultHandler` ending (see the `FINISH` action). The main process called `FAULTHANDLER` executes `MCMD`, `H1` and `H2` in a parallel way; they are synchronized on the action called `FINISH` (they must execute it at the same time).

5.1.3 Feed Back on the B-Method

The main part of this thesis concerns the formalization of the SMEPP service model. The following items inventory difficulties occurred during the modeling process.

Limited Modeling Language

The B-method was initially used because of its capacity to abstractly described operation behaviour by modeling how they manipulate data structures. The interest of such a language is to model input/output properties on data structures that functions must satisfy. Inputs stand for properties that the data structure must satisfy before calling an operation while output properties describe how data structures are updated by an operation execution. These functional descriptions are not able to model other kinds of characteristics such as: how processes must communicate and interacting together during their executions, properties on operation behaviour, etc.

Another Model

The current version of the B-machine doesn't model several entities interacting together. Indeed, only one peer identified by the constant `currentPeer` and its services are allowed to interact with a random SMEPP network i.e., the current network state is chosen at the creation time of the B-machine. By

calling operations, we can see how operations interact with the current network state where other kinds of entities are not able to response.

But simple changes on our B-machine can take into account other entities. First we have to eliminate the constant `currentPeer`. Secondly we must add one argument to every B-operation. This argument identifies the caller: a peer or a service. We have already made this modification on several operations, but the caller should be `currentPeer` or one of its services. Now, this constraint is replaced by the following constraint: in the case of a service, the caller must be a current published service. In the case of a peer, the caller must be a peer belonging to the peer set of the current network state (see `network`).

Every time `currentPeer` is used, it must be replaced by the identifier of the peer associated with the caller identifier which is the argument, we have added to every operation. If the caller identifier is a service identifier, `currentPeer` must be replaced by the peer that published the service. Otherwise, the caller identifier is a peer identifier and `currentPeer` must be replaced by the caller identifier. So, the operation context is dynamic, different entities (different peers and services) are allowed to call B-operations. Interactions among the network entities are taken into account.

We can make deep changes by taking into account several SMEPP networks. In that model, the variable `network` doesn't refer to a SMEPP network state but to a set of SMEPP network states. B-operations are callable by different entities belonging to different networks. Each operation call must be associated with a given SMEPP network (identified by its credentials) and the caller identifier. Thanks to these both arguments, B-operations are able to identify their contexts: on which network they must act and which entity requires the modification on the current network, or an information from the network.

Minimizing Code

Several predicates contained in the `PROPERTIES` clause of our B-machine, but also in operation preconditions and in substitutions (see Table 3.5, page 42) share common parts. These shared parts can be extracted and placed in the `DEFINITIONS` clause of the B-machine under a new definition referred by a unique name. Extracting common parts and replacing them by the name of the associated definition brings a readable and understandable B-code. Further details over this B-clause can be found in [Cle07, p. 8-10]. Unfortunately, few predicate parts are shared in our B-machine and its final version doesn't contain any definition.

ProB Limitation

An advantage of the B-language is the existence of a tool called ProB[Pro09] able to animate and to proceed to model checking. Unfortunately, ProB seams to suffer from limitations in properties checking. Indeed, ProB never succeed to load our B-machine. Failures appeared when it tried to check predicates of the `PROPERTIES` clause, especially predicates related to structure definitions.

Moreover, ProB doesn't provide any further information allowing programmers to understand why loadings failed and how they can eventually change their models. Different solutions have been tried like adding properties that enumerate structure values in order to have enumerated sets with finite values (less than 10 values for each set). But, adding these properties brings more properties which have to be checked and the animator couldn't satisfy some properties previously satisfied. So currently, the B-model has not been simulated yet. It has only been proved.

If-then-else Predicates

Another drawback of the B-method concerns if-then-else constructions inside predicates. Indeed, a predicate having the following syntax: $(P \Rightarrow Q) \wedge (\neg P \Rightarrow R)$ implies to write `P` and its negation. But, in our model some `P` are very long to write and the code becomes unreadable for humans.

As previously mentioned, we considered that the evaluation process of disjunctions is lazy (only the first disjunction clause evaluated to true is considered). Thus, this structure can have the following form:

`( (P ∧ Q) ∨ R)`. But, the B-documentation doesn't mention how disjunctions are evaluated. However, ProB complies with this process characteristic.

B-Structures

As explained in Subsection 3.4.2, records structures are a technical mechanism provided by the B-language, used to facilitate the manipulation of sets based on cartesian products. Indeed, records allow to directly access to an element of a n-uplet, avoiding complex manipulation and data typing. For example, instead of declaring a set containing pairs of strings and naturals: `MyStructure = (STRING × NAT)`, we can define a structure: `MyStructure2 = struct(first: STRING, second: NAT)`.

In the first implementation, when we need to access an n-uplet element, three varibles are required for destructuring the pair:

```
nUplet ∈ MyStructure
firstElement ∈ STRING
secondElement ∈ NAT
∀ nUplet ∈ Mystructure .  (
  (nUplet = (firstElement, secondElement) ∧
  ( (firstElement = ‘‘’’) ⇒ (secondElement >0) )
)
```

In the second implementation, we can simply write:

```
nUplet ∈ MyStructure
∀ nUplet ∈ Mystructure .  (
  ( (nUplet’first = ‘‘’’) ⇒ (nUplet’second >0) )
)
```

As a result, records have been used instead of using cartesian products.

5.2 Future Work on the SMoL Translator

The purpose of our internship was the creation of a translator able to translate any kinds of contracts and behaviour files to Java. The translator cannot make any assumption on the way the SMEPP API is implemented. This section is dedicated to possible improvements and evolutions.

5.2.1 Structured Commands Implementation

An improvement of the translator was planned in a second step of our internship. This version intends to eliminate drawbacks of the first implementation. Unfortunately, this second version has not been implemented yet because of a delay in its development. In that version, translation of structured commands are optimized because they are pushed inside the API. Indeed, placing them in a lower level brings more computing and memory performance, but also an improved compliance with the service model.

The current implementation of `ReceiveEvent` can lead to losts of events because after a certain interval of time[1] applications stop listening to events and check whether their current contexts require that they stop listening. In these cases, primitives stop listening, otherwise they continue to listen. Unfortunately, events arrived when applications that were not listening, are lost. This problem can be resolved if the context (see the `IThreadedCommand` class) of `ReceiveEvent` is given to the API when the primitive is called. Placing that test inside the API avoids events loosing because the API makes that test by itself.

[1] See `org.smepp.extendFramework.SMoL.WAITING_TIME_STEP`

`Pick` requires one thread listening to `ReceiveMessage`, `ReceiveResponse` and `alarm` branches plus a thread for each `ReceiveEvent` (can be customized). Even if only one branch will be executed, these threads will be created and executed concurrently. But, if handlers are executed inside the API, only one thread is necessary. Indeed, if we provide `Pick` handlers and their commands to the API, once it receives an event, a response, a message or if the activation date of an alarm is reached, the API is able to search (by itself) for a corresponding handler. If an handler is found, its command starts executing. Only one thread has been used and no event has been lost since the API doesn't stop to listen. The context of the `Pick` must also be provided to the API in order to know whether the given `Pick` must be stopped.

`InformationHandler` encounters the same problem while trying to receive events. We can modify the implementation as previously described. A thread executing the main command remains necessary, but threads executing `ReceiveEvent` handlers (a thread per `ReceiveEvent` handler) and the only thread executing other kinds of handlers are no more necessary. Indeed, handlers of an `InformationHandler` can be executed inside the API under one thread. So only one extra thread is required by the entire `InformationHandler`. Each time the activation date of an alarm is reached, or once an event, a response, or an invocation is received, the API can create a new thread used to execute the associated handler command. At the same time, the API can check that the `InformationHandler` context must not be stopped. So, we have reduced the number of threads while avoiding to loose events since the API doesn't stop to listen.

5.2.2 XPath Queries

As previously mentioned, XPath queries are limited. Indeed, translations of many XPath queries can be improved. For example, XPath functions are not translated to their counterparts in Java; expressions containing functions are simply copied to Java where all XPath variables are mapped to Java. In order to improve the translator, instead of using an XPath evaluator, we need an XPath parser (for example, Jaxen[jax09]) which allows the translator to better understand XPath contents. Once the structure of an XPath expression is understood, the translator is able to provide a better translation. It's able to know which XPath functions are used inside a query, but also whether XML nodes are referred by a query, etc. For the first release of the translator, the current translation process has been approved by M. Benigni.

5.2.3 Implementation Correctness

The following subsection has been inspired by [BL08]. The SMEPP middleware intends to be used in application requiring reliability, safety and correctness (prohibition of unexpected behaviours). First, the SMEPP middleware implementation needs to be proved compliant to the SMEPP service model. Secondly, the SMoL translator must keep semantics of SMoL programs.

Proving the translator can be achieved by using transformational models. To establish the equivalence between a SMoL program and its translation to Java, the SMoL program and its translation must have the same semantic. A translation is a transformation from a model (describing a SMoL application) to another model (describing a Java application). But, in order to know whether the transformational process keeps semantics, we need to abstract both models by creating their meta-models. A meta-model is able to give meanings to a model. YAWL [BP08b] can be used as a meta-model of SMoL models (thanks to transformation rules mapping SMoL constructs to YAWL while keeping their semantics). But, providing a metal-model for the translated Java code (using an abstract version of Java) is not a trivial work.

Currently, only unit and integrated tests have been made. One by one SMoL commands have been translated to Java, trying every optional and mandatory argument. Translated primitives must be compilable and must correspond to the code that a human could write. Translation of SMoL basic commands have been executed in different scenarios: commands contained in a structured command and contexts where exceptions are raised. Executions must behave as expected. After that, each type of variables declarations (global variables, handler variables, local variables, etc.) have been tried, the translated code must act as expected. Structured commands have been essentially executed in integrated

tests. These tests are based on real applications. Translated SMoL applications must act as expected. But, tests of applications based on an unfinished SMEPP API were complicated: some primitives are unimplemented, there are bugs in the current implementation, etc. Moreover, testing distributed applications is an hard work: many realistic scenarios are possible, analyzing and finding problems encountered in interactions, etc. So, the translator itself has been tested, but classes implementing SMoL structures (`Pick`, `Flow` and `InformationHandler`) should be tested more. These classes have to be tested in more applications once the following problems related to the implementation of the SMEPP API will be resolved:

- The `ReceiveResponse` primitive was not implemented in the SMEPP API so, `ReceiveResponse` handlers (see the SMoL `Pick` and `InformationHandler`) have not been tested.
- Events are not received by other peers. They are only received by the sender, or by its services.
- Invocations on services provided by other peers are not received by these services.

Chapter 6

Conclusion

Nowadays, new kinds of networking models have emerged taking advantage of the improvement in computer power, bandwidth and storage space. Among them, the "Peer-to-Peer model (P2P)" has been specifically addressed in the context of this thesis. The P2P model is no more based on the classical client-server model. Indeed, each network element acts simultaneously as a server and as a client. Moreover, at the same time, the recent technological advances in short distance wireless communications and embedded system have opened up new areas of system. Embedded Peer-to-Peer systems are P2P system where small, low-powered, low-cost embedded systems collaborate in the processing and transmission of information using wireless channels. Network elements can join or leave the network as they want and where they want. Such systems represent a new challenge in application developments: involved devices are poor computer power and prone to frequent and unpredicted disconnections since they communicate on unreliable wireless technologies. In addition, these systems are extremely vulnerable in terms of security.

These challenges can be overcome thanks to the **SMEPP middleware** able to wrap these difficulties by hiding the complexity of the underlying network. The SMEPP middleware can be used by programmers through the usage of the **SMEPP API** which provides a set of abstract Java primitives interacting with the middleware. These primitives are the only way to interact with it.

Moreover, we have mentioned in Chapter 2 that an abstract language able to describe observable behaviour of entities interacting on the network by showing how programs orchestrate the SMEPP primitives, is needed. This language has been developed under the name of **SMEPP Modeling Language** (SMoL). This language allows to build complex Embedded Peer-to-Peer programs while being able to reason formally on the behaviour of network entities. The purpose of our **internship** was the **development of a SMoL translator** producing Java applications from any kind of program descriptions written in SMoL. So, programmers can easily define complex Embedded Peer-to-Peer programs automatically translated to Java without any human manipulation. We have seen (see Appendix C) that the translated code is so closed to the final Java code that it doesn't require many changes.

After an introduction to the SMEPP project in Chapter 2, we have formalized in Chapter 3 the SMEPP service model by modeling its API primitives. Firstly all these primitives have been introduced one by one: the primitive signature and a short description about its effects. Indeed, thinking in terms of primitives allows to have a global meaning of the SMEPP service model.

Once introduced, we have seen in Section 3.2 how SMoL structures manipulate these primitives in order to design EP2P applications. In the following Subsection, a sample health-care application has been imagined and implemented in SMoL. In few lines, we have created an EP2P application much simpler to program and understand than an application written in Java, or any other programming language.

Subsections 3.3 and 3.4, the **main contribution** of this thesis, intend to **formalize** how **SMEPP primitives** interact with the SMEPP network by modeling in the B-language their effects on the current SMEPP network state. The B-method, introduced in Subsection 3.3 was initially used because of its capacity to abstractly described operation behaviour by modeling how they manipulate data structures in terms of input/output properties: input stands for properties that data structures must satisfy before calling an operation while output properties describe how data structures are updated by an operation execution. The data structure modeling all the possible SMEPP network states have been conceived in a first time. The second part of Chapter 3 has modeled one by one API primitives as B-operations acting on a given network state and updating it. Our model didn't aim to model several peers interacting together inside a SMEPP network, but to highlight interactions of an API primitive with the network in the point of a single peer view.

As described above, the project achieved during our internship in the University of Pisa (Italy) was the creation of a SMoL2Java translator. The first part of Chapter 4 related to its conception has shown **how it's possible to translate SMoL structures to Java**. We have begun by the most basic concepts of any language: right expressions (called `from`), left expressions (called `to`) and assignments (see the `Assign` operator). After that, we made a comparison between SMoL exceptions and Java exceptions which has proved that SMoL exceptions are so closed to Java exceptions that their implementations in Java require few manipulations. Finally, we have finished by defining how SMoL commands can be implemented in Java. Some of them namely, `Flow`, `Pick` and `InformationHandler`, require further development since they have no counterparts in Java.

In the rest of this Chapter, Section 4.3 explained **implementation** choices made during the creation of the SMoL2Java **translator**. The translator is composed of two Java classes translating service contracts or behaviour files. It's also composed of managers handling: events, exceptions, expressions, types and variables. The way the translator generates Java codes has been described in the Section 4.2.

The critical analysis given in Chapter 5 allowed us to elicit some simplifications made on our SMEPP B-model and differences between the SMEPP service model and its B-implementation. Despite that, some suggestions about future work which could be interesting to investigate to *easily* create a B-machine modeling any SMEPP network are drawn.

Thereafter, although the SMoL translator implementation has met its goals, some new work perspectives have been made. An improvement of the translator was planned in a second step of our internship which was not achieved because of a delay in the development. In the first version, generated codes use the current version of the SMEPP API bringing its own limitations leading to a gap between the definition of some SMoL commands and their implementations. Indeed, the current implementations of `Pick`, `InformationHandler` and `ReceiveEvent` possibly loose some events. A second version can also provide an XPath translator able to translate more complex queries. Finally, we have seen that testing the translator is not a trivial work. Among these difficulties are testing translated codes using an unfinished API, proving that the translator keeps semantics of distributed SMoL programs.

Our goal was to develop a translator able to create Java applications described by SMoL files without making any assumption on the way the SMEPP API is implemented. We believe that we succeeded to fulfill these requirements even if a second version bringing more computing performance, a better compliance with the service model and an improved expressiveness of XPath queries can be realized. Finally, we have highlighted functional properties that the SMEPP primitives must satisfy. Thanks to our contribution, we have completed the SMEPP API documentation by providing a different point of view about the API. Furthermore, our contribution suggests further researches about possible associations between the B-language and other formalisms in order to create a method able to describe different aspects of distributed applications such as: functional properties of operations interacting on networks, descriptions of granted/prohibited operation behaviours (i.e., not in a functional point of view, but in a imperative way), how operations can call other operations and how entities can dialog together.

Bibliography

[Abr06] J-R Abrial. *The B-Book.* Cambridge University Press, 2006.

[BL08] Jean-Louis Buchholz and Julien Lange. On Secure Services for Embedded Mobile Peer-to-peer Systems : a Conceptual Framework and its Implementation in the Coordination Language SecureLime, 2007-2008.

[BP08a] Antonio Brogi and Razvan Popescu. Workflow Semantics of Peer and Service Behaviour. TASE, 2008.

[BP08b] Antonio Brogi and Razvan Popescu. Workflow Semantics of Peer and Service Behaviour. 2008.

[Cle07] ClearSy. Manuel de Référence Du Langage B. `http://www.atelierb.eu/php/documents-fr.php`, February 26, 2007.

[Coa07a] JXTA Coalition. Juxtapose (homepage), 2007. `http://www.jxta.org`.

[Coa07b] SMEPP Coalition. Deliverable 1.1: State of the Art and Generic Middleware Requirements. 2007. `http://www.smepp.org`.

[Coa08a] SMEPP Coalition. Deliverable 2.1: Service Model Description. 2008. `http://www.smepp.org`.

[Coa08b] SMEPP Coalition. Deliverable 3.2: Conceptual Architecture of Secure EP2P Middleware. 2008. `http://www.smepp.org`.

[Doy00] Stephen Doyle. *Understanding Information Technology.* Nelson Thornes, 2000.

[Eng08] Vincent Englebert. *Conception des systèmes distribués et coopératifs, INFO2208*, chapter Modèle de communication, page 5. Universty of Namur - Computing Science Faculty, 2007-2008.

[FP6] FP6 Description. `http://en.wikipedia.org/wiki/Sixth_Framework_Programme`.

[GP05] G. Gehlen and L. Pham. Mobile Web Services for Peer-to-Peer Applications. In *Proceedings of the Consumer Communications and Networking Conference*, pages 427–433, 2005.

[GRA31] Technische Universitaet Graz, 2009-03-31. `http://portal.tugraz.at/portal/page/portal/TU_Graz`.

[IIR31] Institue for Infocomm Research, 2009-03-31. `http://www.i2r.a-star.edu.sg/About_Us.html`.

[jax09] jaxen. jaxen, 2009. `http://jaxen.codehaus.org/`.

[Leu09] Michael Leuschel. B-language Introduction Lesson. 2009.

[Loo07] Alfred Wai-Sing Loo. *Peer-to-peer Computing: Building Supercomputers with Web TechnologiesPeer-to-peer Computing: Building Supercomputers with Web Technologies.* Springer, 2007.

[LZ04] R. Lucchi and G. Zavattaro. Wssecspaces: a Secure Data-Driven Coordination Service for Web Services Applications. In R. L. Wainwright H. Haddad, A. Omicini and L. M. Liebroc, editors, *SAC*, pages 487–491. ACM, 204.

[MAL31] Universidad de Málaga, 2009-03-31. http://internacional.universia.net/espanya/uma/inf_general_ing.htm.

[MB05] Michael Leuschel Michael Butler. Combining CSP and B for Specification and Property Verification. In *International Symposium of Formal Methods Europe*, volume 3582, pages 221–236, 2005.

[MBJ05] G. D. Modica M. Bisignano and O. Tomarchio. Jmobipeer. A Middleware for Mobile Peer-to-Peer Computing in MANETs. In *ICDCS Workshops*, pages 785–791. IEEE Computer Society, 2005.

[NBZ04] Alberto Montresor Nadia Busi and Gianluigi Zavattaro. Data-Driven Coordination in Peer-to-Peer Information Systems. 13(1):63–89, March 2004.

[OAS] OASIS. Bpel. http://www.oasis-open.org/committees/wsbpel/.

[Pro09] ProB. ProB, an Animator and Model Checker for the B-Method, 2009. http://www.stups.uni-duesseldorf.de/ProB/.

[Pro15] Amigo Project. Specification of the Amigo Abstract Middleware Architecture, 2009-04-15. http://www.hitech-projects.com/euprojects/amigo/deliverables/Amigo_WP2_D2.1_v10%20final.pdf.

[RHR06] G. Hackmann R. Handorean, R. Sen and G.-C. Roman. Supporting Predictable Service Provision in MANETs Via Context Aware Session Management. (3):1–26, 2006.

[S. 05] S. Alda and A. B. Cremers. Towards Composition Management for Component-Based Peer-to-Peer Architectures. *Electr. Notes Theor. Compu. Sci.*, 114:47–64, 2005.

[SIE31] Siemens - IT Consulting and Systems Integeration, 2009-03-31. http://www.it-solutions.siemens.com.

[TEC31] Tecnatom, 2009-03-31. http://www.tecnatom.es/english/portada.shtml.

[TEL31] Telefónica International Corporate Site, 2009-03-31. http://info.telefonica.es/acercadetelefonica/eng/.

[UPI31] Università di Pisa, 2009-03-31. http://www.di.unipi.it/main.html.

[VTT31] VTT Technical Research Centre of Finland, 2009-03-31. http://www.vtt.fi/vtt/index.jsp.

[W3C] W3C. Xpath 2.0. http://www.w3.org/TR/xpath20/.

[WSD15] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2009-04-15. http://www.w3.org/TR/wsdl20/.

[Wu05] J. Wu. *Handbook On Theoretical And Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*. Auerbach Publications, 2005.

[XML07] Altova XMLSpy, 2009-04-07. http://www.altova.com.

Index

Abstract Machine
  Definition, 41
AbstractDataType
  Definition, 48

ConnectedPeers
  Mathematical Definition, 50
CONSTANTS
  Cartesian Product
    Construct, 44
  Definition of This B-Clause, 42
  Set Comprehension
    Construct, 44
Contract
  Behavior
    Definition, 7
  Definition, 7
  Grounding
    Definition, 7
  Implementation
    Definition, 7
  Mathematical Definition, 47
  Profile
    Definition, 7
  Properties
    Definition, 7
  QoS
    Definition, 7
  Signature
    Definition, 7
Credentials
  Definition, 6
  Group Credentials, 7
  Mathematical Definition, 45
CurrentGroups
  Mathematical Definition, 52

Distributed Transient Networks
  Definition, 3

Embedded Peer-to-Peer Network
  Definition, 4
Event
  Definition, 9
EventsRaised
  Mathematical Definition, 48
Exception
  Definition, 9
  Middleware Exception
    Definition, 35
  User Exception
    Definition, 35
    Throwing An User Exception, 29, 35
ExceptionResultMessage
  Constraint, 80
  Mathematical Definition, 60

Fault
  Definition, 9

Group
  Constraint, 68
  Definition, 6, 9
  Mathematical Definition, 51
GroupId
  Constraint, 68
  Mathematial Definition, 45
GroupServiceId
  Constraint, 67
  Introduction, 8
  Mathematical Definition, 45

INITIALIZATION
  Definition of This B-Clause, 44
INVARIANT
  Definition of This B-Clause, 44
Invocation
  Mathematical Definition, 58
Invocations
  Constraint, 77
InvocationsState
  Mathematical Definition, 59

Mobile Ad-hoc Network (MANET)
  Definition, 4

NetworkState
  Constraint, 81
  Mathematical Definition, 61
NormalResultMessage
  Constraint, 80
  Mathematical Definition, 59

Operation
  B-Operation
    Definition, 44

Pre-condition, 44
Deferred Synchronous
Definition, 8
Definition, 8
Mathematical Definition, 48
One-way
Definition, 8
Request-response
Definition, 8
Synchronous
Definition, 8

Peer
Constraint, 68
Definition, 6
Mathematical Definition, 50
Peer-to-Peer Network
Definition, 3
PeerId
Mathematical Definition, 45
PeerServiceId
Constraint, 66
Introduction, 8
Mathematical Definition, 45
PROPERTIES
Definition of This B-Clause, 44

ReplyMessagesState
Constraint, 80
Mathematical Definition, 60

Self-Organizing Networks
Definition, 3
SentEvent
Constraint, 77
Mathematical Definition, 56
SentEventsState
Constraint, 77
Mathematical Definition, 57
Service
Composition, 7
Constraint, 66
Definition, 6
Instance
Definition, 7
Mathematical Definition, 49
State-Full
Definition, 8
State-Full Session-Full
Definition, 8
State-Full Session-Less
Definition, 8
State-Less
Definition, 8
ServiceInfo
Mathematical Definition, 47
Session
Definition, 8, 27
SessionId
Constraint, 66
Mathematical Definition, 45
SETS
Deferred
Construct, 42
Definition of This B-Clause, 42
Enumeration
Construct, 42
SMEPP API
createGroup
Definition, 22
Formalization, 82
Definition, 6
event
Definition, 30
Formalization, 89
getGroupDescription
Definition, 23
Formalization, 83
getGroups
Definition, 23
Formalization, 83
getIncludingGroups
Definition, 24
Formalization, 84
getPeerId
Definition, 22
Formalization, 82
getPeers
Definition, 25
Formalization, 84
getPublishingGroup
Definition, 24
Formalization, 84
getServiceContract
Definition, 26
Formalization, 86
getServices
Definition, 26
Formalization, 86
invoke
Definition, 27
Formalization, 87
joinGroup
Definition, 24
Formalization, 83
leaveGroup
Definition, 24
Formalization, 83
newPeer
Definition, 21
Formalization, 82
publish
Definition, 25

Formalization, 85
receiveEvent
Definition, 31
Formalization, 90
receiveMessage
Definition, 28
Formalization, 88
receiveResponse
Definition, 30
Formalization, 89
reply
Definition, 29
Formalization, 88
startSession
Definition, 27
Formalization, 87
subscribe
Definition, 31
Formalization, 90
unpublish
Definition, 26
Formalization, 85
unsubscribe
Definition, 32
Formalization, 92
SMoL
Assign
Definition, 36
Implementation, 98
Branch, 38, 39
catch
Definition, 35
catchAll
Definition, 35
empty
Definition, 34, 106
exit
Definition, 35, 107
FaultHandler
Definition, 40
Flow
Definition, 37
Implementation, 100
Handler, 38, 39
If-Then-Else
Definition, 37
Implementation, 106
Information Handler
Branch Disabled, 39
Branch Enabled, 39
Can Be Executed, 39
Definition, 39
Implementation, 103
Main Command, 39
Introduction, 34
Parser, 95
Pick
Branch Disabled, 38
Branch Enabled, 38
Can Be Executed, 38
Definition, 38
Implementation, 101
RepeatUntil
Definition, 37
Implementation, 105
Sequence
Definition, 36
Implementation, 106
tFrom, 97
throw
Definition, 35
tTo, 96
Variable, 96
wait
Definition, 34, 106
While
Definition, 37
Implementation, 105
SubscriptionAllEventsAllGroups
Constraint, 74
Mathematical Definition, 53
SubscriptionAllEventsInAGroup
Constraint, 72
Mathematical Definition, 53
SubscriptionEventAllGroups
Constraint, 72
Mathematical Definition, 52
SubscriptionEventInAGroup
Constraint, 69
Mathematical Definition, 52
SubscriptionsState
Constraint, 69
Mathematical Definition, 53

Toperation
Mathematical Definition, 46
Tservice
Mathematical Definition, 46

UnsubscriptionAllEventsInAGroup
Constraint, 75
Mathematical Definition, 55
UnsubscriptionEvent
Constraint, 74
Mathematical Definition, 55
UnsubscriptionEventInAGroup
Constraint, 75
Mathematical Definition, 55
UnsubscriptionsState
Constraint, 74
Mathematical Definition, 56

VARIABLES

Definition of This B-Clause, 44

Appendices

Appendix A

B Summary

A Concise Summary of the B mathematical toolkit[1]

Each construct will be presented in its publication form, followed by the boxed `ASCII form` that is used with the BToolkit.

In the following: P, Q and R denote predicates; x and y denote single variables; z denotes a list of variables; S and T denote set expressions; U denotes a set of sets; E and F denote expressions; m and n denote lists of integer expressions; f and g denote functions; r denotes a relation; s and t denote sequence expressions; G, H and I denote a generalized substitutions.

1 Predicates

A predicate is a function from some set X to Boolean. In B an implementation of the type *BOOL* is available from the machine *Bool_TYPE*.

The meta-predicate $z \setminus E$ ("z not free in E") means that none of the variables in z occur *free* in E. This meta-predicate is defined recursively on the structure of E, but we won't do that here. The base cases are: $z \setminus (\forall z \cdot P)$, $z \setminus (\exists z \cdot P)$, $z \setminus \{z|P\}$, $z \setminus (\lambda z \cdot (P|E))$, and $\neg(z \setminus z)$.

A predicate P *constrains* the variable x if it contains a predicate of the form: $x \in S$, $x \subseteq S$, $x \subset S$, or $x = E$, where $x \setminus S$, $x \setminus E$.

1. Conjunction: $P \wedge Q$ `P & Q`
2. Disjunction: $P \vee Q$ `P or Q`
3. Implication: $P \Rightarrow Q$ `P => Q`
4. Equivalence: $P \iff Q$ `P <=> Q`.
   $P \iff Q = P \Rightarrow Q \wedge Q \Rightarrow P$
5. Negation: $\neg P$ `not P`
6. Universal quantification:
   $\forall z \cdot (P \Rightarrow Q)$ `!(z).(P => Q)`
   For all values of z satisfying P, Q (is true)
   P must *constrain* the variables in z.
7. Existential quantification:
   $\exists z \cdot (P \wedge Q)$ `#(z).(P & Q)`
   There exists some values of z satisfying P for which Q. P must *constrain* the variables in z.
8. Substitution: $[G]\ P$ `[G] P`
9. Equality: $E = F$ `E = F`
10. Inequality: $E \neq F$ `E /= F`

2 Sets

1. Singleton set: $\{E\}$ `{E}`
2. Set enumeration: $\{E, F\}$ `{E, F}`
   Notice that the pattern E, F can be applied recursively to yield any finite enumeration.
3. Empty set: $\{\}$ `{}`
4. Set comprehension: $\{\ z \mid P\ \}$ `{ z | P }`
   The set of all values of z that satisfy the predicate P. P must *constrain* the variables in z.
5. Union: $S \cup T$ `S \/ T`
6. Intersection: $S \cap T$ `S /\ T`
7. Difference: $S - T$ `S-T`
   $S - T = \{x \mid x \in S \wedge x \notin T\}$
8. Ordered pair: $E \mapsto F$ `E |-> F`.
   $E \mapsto F = E, F$
   Note: in most places $E \mapsto F$ must be used, and E, F will not be accepted, but there are a few contexts, for example set comprehension: $\{x, y|P\}$, where $E \mapsto F$ is not accepted!
9. Cartesian product: $S \times T$ `S * T`
   $S \times T = \{x, y \mid x \in S \wedge y \in T\}$
10. Powerset: $\mathbb{P}(S)$ `POW(S)`
    $\mathbb{P}(S) = \{s \mid s \subseteq S\}$
11. Non-empty subsets: $\mathbb{P}_1(S)$ `POW1(S)`
    $\mathbb{P}_1(S) = \mathbb{P}(S) - \{\{\}\}$
12. Finite subsets: $\mathbb{F}(S)$ `FIN(S)`
13. Finite non-empty subsets: $\mathbb{F}_1(S)$ `FIN1(S)`
14. Cardinality: $\mathsf{card}(S)$ `card(S)`
    Defined only for finite sets
15. Generalized union: $\mathsf{union}(U)$ `union(U)`
    The union of all the elements of U.
    $\forall U \cdot U \in \mathbb{P}(\mathbb{P}(S)) \Rightarrow$
    $\mathsf{union}(U) = \{x \mid x \in S \wedge (\exists s \cdot s \in U \wedge x \in s)\}$
    where $x, s \setminus U$
16. Generalized intersection: $\mathsf{inter}(U)$ `inter(U)`
    The intersection of all the elements of U.
    $\forall U \cdot U \in \mathbb{P}(\mathbb{P}(S)) \Rightarrow$
    $\mathsf{inter}(U) = \{x \mid x \in S \wedge (\forall s \cdot s \in U \Rightarrow x \in s)\}$
    where $x, s \setminus U$
17. Generalized union:
    $\bigcup z \cdot (P \mid E)$ `UNION (z).(P | E)`
    P must *constrain* the variables in z.
    $(\forall z \cdot (P \Rightarrow E \subseteq T)) \Rightarrow$
    $\bigcup z \cdot (P \mid E) = \{x \mid x \in T \wedge (\exists z \cdot (P \wedge x \in E))\}$
    where $z \setminus T, P, E$
18. Generalized intersection:
    $\bigcap z \cdot (P \mid E)$ `INTER (z).(P | E)`
    P must *constrain* the variables in z.

[1]Version March 5, 2003©1996-2003 Ken Robinson

$(\forall z \cdot (P \Rightarrow E \subseteq T)) \Rightarrow$
$\bigcap z \cdot (P \mid E) = \{x \mid x \in T \wedge (\forall z \cdot (P \Rightarrow x \in E))\}$
where $z \setminus T, P, E$

2.1 Set predicates

1. Set membership: $E \in S$ `E : S`
2. Set non-membership: $E \notin S$ `E /: S`
3. Subset: $S \subseteq T$ `S <: T`
4. Not a subset: $S \not\subseteq T$ `S /<: T`
5. Proper subset: $S \subset T$ `S <<: T`
6. Not a proper subset: $s \not\subset t$ `S /<<: T`

3 Numbers

The following is based on the set of natural numbers (non-negative integers), but the operators extend (directly in most cases) to the set of integers.

1. The set of natural numbers: $\mathbb{N}$ `NAT`
2. The set of positive natural numbers: $\mathbb{N}_1$ `NAT1`
   $\mathbb{N}_1 = \mathbb{N} - \{0\}$
3. Minimum: $\mathsf{min}(S)$ `min(S)`
   Note: $S : \mathbb{F}_1(\mathbb{N})$
4. Maximum: $\mathsf{max}(S)$ `max(S)`
   Note: $S : \mathbb{F}_1(\mathbb{N})$
5. Sum: $m + n$ `m + n`
6. Difference: $m - n$ `m - n`
7. Product: $m \times n$ `m * n`
8. Quotient: m/n `m / n`
9. Remainder: $m \;\mathsf{mod}\; n$ `m mod n`
10. Interval: $m \,..\, n$ `m .. n`
    $m \,..\, n = \{\ i \mid m \leq i \leq n\ \}$.
11. Set summation:
    $\Sigma\ z \cdot (P \mid E)$ `SIGMA(z).(P | E)`
    $\{z \mid P\} = \{\} \Rightarrow \Sigma\ z \mid (P \mid E) = 0$.
12. Set product: $\Pi z \mid (P \mid E)$ `PI(z).(P | E)`
    Defined only for $\{z \mid P\} \neq \{\}$.

3.1 Number predicates

1. Greater: $m > n$ `m > n`
2. Less: $m < n$ `m < n`
3. Greater or equal: $m \geq n$ `m >= n`
4. Less or equal: $m \leq n$ `m <= n`

4 Relations

A relation is a set of ordered pairs; a many to many mapping.

1. Relations: $S \leftrightarrow T$ `S <-> T`
   $S \leftrightarrow T = \mathbb{P}(S \times T)$
2. Domain: $\mathsf{dom}(r)$ `dom(r)`
   $\forall r \cdot r \in S \leftrightarrow T \Rightarrow$
   $\mathsf{dom}(r) = \{x \mid (\exists y \cdot x \mapsto y \in r)\}$
3. Range: $\mathsf{ran}(r)$ `ran(r)`
   $\forall r \cdot r \in S \leftrightarrow T \Rightarrow$
   $\mathsf{ran}(r) = \{y \mid (\exists x \cdot x \mapsto y \in r)\}$
4. Forward composition: $p\,;q$ `p ; q`
   $\forall p, q \cdot p \in S \leftrightarrow T \wedge q \in T \leftrightarrow U \Rightarrow$
   $p\,;q = \{x, y \mid (\exists z \cdot x \mapsto z \in p \wedge z \mapsto y \in q)\}$
5. Backward composition: $p \circ q$ `p circ q`
   $p \circ q = q\,;p$
6. Identity: $\mathsf{id}(S)$ `id(S)`
   $\mathsf{id}(S) = \{x, y \mid x \in S \wedge y \in S \wedge x = y\}$.
7. Domain restriction: $S \lhd r$ `S <| r`
   $S \lhd r = \{x, y \mid x \mapsto y \in r \wedge x \in S\}$.
8. Domain substraction: $S \ntriangleleft r$ `S <<| r`
   $S \ntriangleleft r = \{x, y \mid x \mapsto y \in r \wedge x \notin S\}$.
9. Range restriction: $r \rhd T$ `r |> T`
   $r \rhd T = \{x, y \mid x \mapsto y \in r \wedge y \in T\}$.
10. Range subtraction: $r \ntriangleright T$ `r |>> T`
    $r \ntriangleright T = \{x, y \mid x \mapsto y \in r \wedge y \notin T\}$.
11. Inverse: r^{-1} `r~`
    $r^{-1} = \{y, x \mid x \mapsto y \in r\}$.
12. Relational image: $r[S]$ `r[S]`
    $r[S] = \{y \mid \exists x \cdot x \in S \wedge x \mapsto y \in r\}$.
13. Right overriding: $r_1 \Lleftarrow r_2$ `r1 <+ r2`
    $r_1 \Lleftarrow r_2 = r_2 \cup (\mathsf{dom}(r_2) \ntriangleleft r_1)$.
14. Left overriding: $r_1 \Rrightarrow r_2$ `r1 +> r2`
    $r_1 \Rrightarrow r_2 = r_1 \cup (\mathsf{dom}(r_1) \ntriangleleft r_2)$.
15. Direct product: $p \otimes q$ `p >< q`
    $p \otimes q = \{x, (y, z) \mid x \mapsto y \in p \wedge x \mapsto z \in q\}$.
16. Parallel product: $p \parallel q$ `p || q`
    $p \parallel q = \{(x, y), (m, n) \mid x \mapsto m \in p \wedge y \mapsto n \in q\}$.
17. Iteration: r^n `iterate(r,n)`
    $r \in S \leftrightarrow S \Rightarrow r^0 = \mathsf{id}(S) \wedge r^{n+1} = r\,;r^n$.
18. Closure: r^* `closure(r)`
    $r^* = \bigcup n \cdot (n \in \mathbb{N} \mid r^n)$.
19. Projection: $\mathsf{prj1}(S, T)$ `prj1(S,T)`
    $\mathsf{prj1}(S, T) =$
    $\{(x, y), z \mid x, y \in S \times T \wedge z = x\}$.
20. Projection: $\mathsf{prj2}(S, T)$ `prj2(S,T)`
    $\mathsf{prj2}(S, T) =$
    $\{(x, y), z \mid x, y \in S \times T \wedge z = y\}$.

4.1 Functions

A function is a relation with the restriction that each element of the domain is related to a unique element in the range; a many to one mapping.

1. Partial functions: $S \pto T$ `S +-> T`
   $S \pto T = \{r \mid r \in S \leftrightarrow T \wedge r^{-1} \,;\, r \subseteq \mathsf{id}(T)\}$.

2. Total functions: $S \longrightarrow T$ `S --> T`
   $S \longrightarrow T = \{f \mid f \in S \pto T \wedge \mathsf{dom}(f) = S\}$.

3. Partial injections: $S \pinj T$ `S >+> T`
   $S \pinj T = \{f \mid f \in S \pto T \wedge f^{-1} \in T \pto S\}$.
   One-to-one relations.

4. Total injections: $S \rightarrowtail T$ `S >-> T`
   $S \rightarrowtail T = S \pinj T \cap S \longrightarrow T$.

5. Partial surjections: $S \psur T$ `S +->> T`
   $S \psur T = \{f \mid f \in S \pto T \wedge \mathsf{ran}(f) = T\}$.
   Onto relations.

6. Total surjections: $S \twoheadrightarrow T$ `S -->> T`
   $S \twoheadrightarrow T = S \psur T \cap S \longrightarrow T$.

7. Bijections: $S \rightarrowtail\!\!\!\!\twoheadrightarrow T$ `S >->> T`
   $S \rightarrowtail\!\!\!\!\twoheadrightarrow T = S \rightarrowtail T \cap S \twoheadrightarrow T$.
   One-to-one and onto relations.

8. Lambda abstraction:
   $\lambda z \cdot (P \mid E)$ `%z.(P|E)`
   P must *constrain* the variables in z.
   $\lambda z \cdot (P \mid E) = \{z, y \mid z \in \{z \mid P\} \wedge y = E\}$, where $y \setminus P$ and $y \setminus E$.

9. Function application: $f(E)$ `f(E)`
   $E \mapsto y \in f \Rightarrow f(E) = y$.

4.2 Sequences

Sequences are ordered aggregations, and can be modelled by functions whose domains are finite, coherent domains $1 \,..\, n$.

1. The empty sequence: $[\,]$ `<>`
   $[\,] = \{\}$.
   Note: $[\,]$ is used for all sequences except the empty ASCII sequence!

2. The set of finite sequences: $\mathsf{seq}(S)$ `seq S`
   $\mathsf{seq}(S) = \{f \mid f \in \mathbb{N}_1 \pto S \wedge \exists\, n \cdot n \in \mathbb{N} \wedge \mathsf{dom}(f) = 1 \,..\, n\}$.

3. The set of finite non-empty sequences:
   $\mathsf{seq}_1(S)$ `seq1(S)`
   $\mathsf{seq}_1(S) = \mathsf{seq}(S) - \{[\,]\}$.

4. The set of injective sequences:
   $\mathsf{iseq}(S)$ `iseq(S)`
   $\mathsf{iseq}(S) = \mathsf{seq}(S) \cap (\mathbb{N}_1 \pinj S)$.

5. Permutations: $\mathsf{perm}(S)$ `perm(S)`
   $\mathsf{perm}(S) = \mathsf{iseq}(S) \cap (\mathbb{N}_1 \twoheadrightarrow S)$.
   The set of bijective sequences.

6. Sequence concatenation: $s \frown t$ `s^t`
   $s \frown t$ is the sequence formed by appending the sequence t to the sequence s.

7. Prepend element: $E \rightarrow s$ `E -> s`
   $E \rightarrow s = [E] \frown s$.

8. Append element: $s \leftarrow E$ `s <- E`
   $s \leftarrow E = s \frown [E]$.

9. Singleton sequence: $[E]$ `[E]`
   $[E] = \{1 \mapsto E\}$.

10. Sequence construction: $[E, F]$ `[E,F]`
    $[E, F] = [E] \leftarrow F$.

11. Size: $\mathsf{size}(s)$ `size(s)`
    $\mathsf{size}(s) = \mathsf{card}(s)$.

12. Reverse: $\mathsf{rev}(s)$ `rev(s)`
    $\forall\, i \cdot i \in \mathsf{dom}(s) \Rightarrow$
    $\mathsf{rev}(s)(i) = s(\mathsf{size}(s) + 1 - i)$.

13. Take: $s \uparrow n$ `s /|\n`
    $s \uparrow n = 1 \,..\, n \lhd s$.

14. Drop: $s \downarrow n$ `s \|/ n`
    $s \downarrow n = (\lambda m \cdot (m \in \mathbb{N} \mid m + n)) \,;\, (1 \,..\, n \lhd\!\!\!- s)$.
    $(s \downarrow n)(i) = s(i + n)$

15. First element: $\mathsf{first}(s)$ `first(s)`
    $\mathit{first}(s) = s(1)$
    Defined only for non-empty sequence.

16. Last element: $\mathsf{last}(s)$ `last(s)`
    $\mathsf{last}(s) = s(\mathit{size}(s))$
    Defined only for non-empty sequence.

17. Tail: $\mathsf{tail}(s)$ `tail(s)`
    $\mathit{tail}(s) = s \downarrow 1$
    Defined only for non-empty sequence.
    $\mathsf{first}(s) \rightarrow \mathsf{tail}(s) = s$.

18. Front: $\mathsf{front}(s)$ `front(s)`
    $\mathsf{front}(s) = s \uparrow (\mathit{size}(s) - 1)$
    Defined only for non-empty sequence.
    $\mathsf{front}(s) \leftarrow \mathsf{last}(s) = s$.

19. Generalized concatenation:
    $\mathsf{conc}(ss)$ `conc(ss)`
    Defined on sequences of sequences.
    $\mathsf{conc}([\,]) = [\,]$
    $\mathsf{conc}(s \leftarrow E) = \mathsf{conc}(s) \frown E$.

20. Strings: "..." `"..."`
    Sequences of characters are delimited by quotes.

5 Substitutions

The state of a machine can be changed by substituting values for the variables in the state. The following substitutions formalize a number of alternative ways of achieving this.

1. Substitution: $[G]P$ `[G]P`
   $[G]P$ is a predicate obtained by replacing the values of the variables in P according to the substitution G.

2. The null substitution: *skip* `skip`
   $[\mathit{skip}]R = R$.

3. Simple substitution: $x := E$ `x := E`
   Replace free occurrences of x by E.

4. Boolean substitution: $x{:=}\mathsf{bool}(P)$ `x := bool(P)`
   Substitute the Boolean values *TRUE* and *FALSE* according to the truth of P.

5. Choice from set: $x :\in S$ `x :: S`
   Arbitrarily choose a value from the set S.

6. Choice by predicate: $x : P$ `x : P`
   Arbitrarily choose a value that satisfies the predicate P. P must *constrain* the variable x.

7. Functional override: $f(x) := E$ `f(x) := E`
   Substitute the value E for the expression f at point x.
   $f(x) := E \ = \ f := f \lessdot \{x \mapsto E\}$.

8. Multiple substitution:
   $x, y := E, F$ `x,y := E,F`
   Concurrent substitution of the values E and F for the free occurrences of x and y, respectively.

9. Parallel substitution: $G \parallel H$ `G || H`
   Apply the substitutions G and H concurrently. Parallel substitution is not given a general definition; it is eliminated by rewriting rules. Notice $[x := E]R \parallel [y := F]R = [x, y := E, F]R$.

10. Sequential substitution: $G \,;\, H$ `G ; H`
    Apply the substitution G and then H.
    $[G \,;\, H]R = [G]([H]R)$.

11. Precondition: $P \mid G$ `P | G`
    Substitution G is subject to a precondition, P.
    $[P \mid G]R = P \wedge [G]R$.

12. Guarding: $P \Longrightarrow G$ `P ==> G`
    Substitution G applies only if state satisfies the guard P.
    $[P \Longrightarrow G]R \ = \ P \Rightarrow [G]R$.

13. Alternatives: $G \,[\!]\, H$ `G [] H`
    Either G or H.
    $[G \,[\!]\, H]R = [G]R \wedge [H]R$.

14. Unbounded choice: $@\, z \cdot G$ `@z . G`
    Choose any values for z. $[@\, z \cdot G]R = \forall\, z \cdot [G]R$.

5.1 Alternative syntax

1. Grouping: **BEGIN** G **END**

2. **PRE** P **THEN** G **END**
   $= P \mid G$

3. **IF** P **THEN** G **ELSE** H **END**
   $= (P \Longrightarrow G) \,[\!]\, (\neg P \Longrightarrow H)$

4. **IF** P **THEN** G **END**
   $=$ **IF** P **THEN** G **ELSE** *skip* **END**

5. **IF** P_1 **THEN** G_1 **ELSIF** P_2 **THEN** G_2 …**ELSE** G_n **END**

6. **IF** P_1 **THEN** G_1 **ELSIF** P_2 **THEN** G_2 …**ELSIF** P_n **THEN** G_n **END**

7. **CHOICE** G **OR** H **END**
   $= G \,[\!]\, H$

8. **SELECT** P **THEN** G **WHEN** …**WHEN** Q **THEN** H **ELSE** I **END**
   $= P \Longrightarrow G \,[\!]\, \ldots \,[\!]\, Q \Longrightarrow H \,[\!]\, \neg P \wedge \ldots \wedge \neg Q \Longrightarrow I$

9. **SELECT** P **THEN** G **WHEN** …**WHEN** Q **THEN** H **END**
   $= P \Longrightarrow G \,[\!]\, \ldots \,[\!]\, Q \Longrightarrow H$

10. **CASE** E **OF EITHER** m **THEN** G **OR** n **THEN** H …**ELSE** I **END**
    $= E \in \{m\} \Longrightarrow G \,[\!]\, E \in \{n\} \Longrightarrow H \ldots E \notin \{m, n, \ldots\} \Longrightarrow I$

11. **CASE** E **OF EITHER** m **THEN** G **OR** n **THEN** H …**END**
    default case *skip*

12. **VAR** z **IN** G **END**
    $= @\, z \cdot G$

13. **ANY** z **WHERE** P **THEN** G **END**
    $= @\, z \cdot P \Longrightarrow G$

14. **LET** x **BE** x = E **IN** G **END**
    $= @\, x \cdot x = E \Longrightarrow G$, where $x \setminus E$

5.2 While loop substitution

WHILE P **DO** G **VARIANT** E **INVARIANT** Q **END**

The while-loop substitution is allowed only in implementation machines. The definition of the substitution [**WHILE** P **DO** G **VARIANT** E **INVARIANT** Q **END**]R involves a least fixed point and is not normally used. Instead, an approximation to the substitution is used.

Given some predicate R:

$$\begin{array}{lcl} Q \wedge P & \Rightarrow & [G]\; Q \\ Q \wedge P & \Rightarrow & E \in \mathbb{N} \\ Q \wedge P & \Rightarrow & [n := E][G](E < n) \\ \neg P \wedge Q & \Rightarrow & R \end{array}$$

$\Rightarrow$

$Q \ \Rightarrow$ [**WHILE** P **DO** G **VARIANT** E **INVARIANT** Q **END**] R

where n is a *new* variable satisfying $n \setminus E$ and $n \setminus G$.

Appendix B

B-Operations

B.1 Peer Management Primitives

B.1.1 newPeer

```
INITIALISATION
    ANY
      currentNetworkState,peer
    WHERE
      (currentNetworkState ∈= NetworkState) ∧
      (peer ∈ currentNetworkState'peers) ∧
      (peer'services = ∅) ∧
      (peer'pid = currentPeer)
    THEN
      /* The peer joins the network or creates the network */
      network ≡ currentNetworkState
    END
```

Figure B.1: newPeer

B.1.2 getPeerId

```
peerId <-- getPeerId =
           peerId currentPeer
```

Figure B.2: getPeerId

```
returnedPeerId <-- getPeerId2(caller,serviceId) =
PRE
    (serviceId ∈ ServiceId) (caller ∈ (PeerId ∪ PeerServiceId)) ∧
    ∃(peerCalled,serviceCalled).( (peerCalled ∈ network'peers) ∧
                                  (serviceCalled ∈ peerCalled'services) ∧
                                  (
                                    ( (serviceId ∈ SessionId) ∧
                                      (serviceID = serviceCalled'sessions) ) ∨
                                    ( (serviceId ∈ PeerServiceId) ∧
                                      (serviceCalled'psid = serviceId) )
                                  ) /* InvalidServiceId */ ∧

                                  /* The caller and the referred service belong to the same group ?*/
                                  (
                                    ( (caller ∈ PeerId) ∧ (caller = currentPeer) ∧
                                      ∃(group).( (group ∈ network'groups) ∧
                                                 (group'gid = serviceCalled'gid) ∧
                                                 (caller ∈ group'members) /* InvalidServiceId */ )
                                    ) ∨
                                    ( (caller ∈ PeerServiceId) ∧
                                      ∃(peerCaller,serviceCaller).( (peerCaller ∈ network'peers) ∧
                                                   (peerCaller'pid = currentPeer) ∧
                                                   (serviceCaller ∈ peerCaller'services) ∧
                                                   /* InvalidServiceId */
                                                   (serviceCaller'gid = serviceCalled'gid) )
                                    )
                                  )
    )
THEN
     ANY
         pid
     WHERE
         (pid ∈ PeerId) ∧
         ∃(peer, service).(
                             (peer ∈ network'peers) ∧
                             (pid = peer'pid) ∧
                             (service ∈ peer'services) ∧
                             (
                               ( (serviceId ∈ PeerServiceId) ∧ (serviceId = service'psid) ) ∨
                               ( (serviceId ∈ SessionId) ∧ (serviceId ∈ service'sessions) )
                             )
         )
     THEN
         returnedPeerId ≡ pid
     END
END
```

Figure B.3: getPeerId with Arguments

B.2 Group Management Primitives

B.2.1 createGroup

```
newGroupId <-- createGroup(groupDescription, credentials) =
PRE
(groupDescription ∈ Gdescr) ∧ (credentials ∈ Credentials)
THEN
     ANY
         groupId, newNetwork
     WHERE
         ∄(group).( (group ∈ network'groups) ∧ (groupId = group'gid) ) ∧
         (newNetwork ∈ NetworkState) ∧
         (newNetwork'cred = network'cred) ∧ (newNetwork'peers = network'peers) ∧
         (newNetwork'groups = network'groups ∪
                             { rec(groupId, credentials, groupDescription,currentPeer) } ) ∧
         (newNetwork'subscriptions = network'subscriptions) ∧
         (newNetwork'unsubscriptions = network'unsubscriptions) ∧
         (newNetwork'sentEvents = network'sentEvents) ∧
         (newNetwork'invocations = network'invocations) ∧
         (newNetwork'replies = network'replies)
     THEN
         newGroupId,network ≡ groupId, newNetwork
     END
END
```

Figure B.4: createGroup

B.2.2 getGroups

```
groups <-- getGroups(groupDescription,credentials) =
PRE
    (groupDescription ∈ Gdescr) ∧ (credentials ∈ Credentials)
THEN
     groups ≡ {group | (group ∈ network'groups) ∧ (group'cred = credentials) ∧
                       (group'gdescr = groupDescription) /* optional */ }
END
```

Figure B.5: getGroups

B.2.3 getGroupDescription

```
groupDescription <-- getGroupDescription(groupId,credentials) =
PRE
    (groupId ∈ GroupId) ∧ (credentials ∈ Credentials) ∧
    ∃(group).( (group :  network'groups) ∧ (group'gid = groupId) /* InvalidGroupId */ ∧
               (group'cred = credentials) /* InvalidGroupId */ )
THEN
     ANY
         descr
WHERE
      (descr :  Gdescr) ∧ ∃(group).( (group :  network'groups) ∧ (group'gid = groupId) )
THEN
     groupDescription := descr
END
```

Figure B.6: getGroupDescription

B.2.4 joinGroup

```
joinGroup(groupId,credentials) =
PRE
    (groupId ∈ GroupId) ∧ (credentials ∈ Credentials) ∧
    ∃(group).( (group :  network'groups) ∧ (group'gid = groupId) /* InvalidGroupId */ ∧
               (group'cred = credentials) /* AccessDenied */ )
THEN
     ANY
         networkUpdated
     WHERE
           (networkUpdated ∈ NetworkState) ∧ (networkUpdated ∈ NetworkState) ∧
           (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
           ∃ (group,group2).( (group ∈ network'groups) ∧ (group'gid = groupId) ∧
                              (group2 ∈ Group) ∧ (group2'gid = group'gid) ∧
                              (group2'cred = group'cred) ∧ (group2'gdescr = group'gdescr) ∧
                              (group2'members = group'members ∪ currentPeer ) ∧
                              (networkUpdated'groups = network'groups - {group} ∪ {group2}) )
           (networkUpdated'subscriptions = network'subscriptions) ∧
           (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
           (networkUpdated'sentEvents = network'sentEvents) ∧
           (networkUpdated'invocations = network'invocations) ∧
           (networkUpdated'replies = network'replies)
     THEN
          network ≡ networkUpdated
     END
END
```

Figure B.7: joinGroup

B.2.5 leaveGroup

```
leaveGroup(groupId) =
PRE
    (groupId ∈ GroupId) ∧
    ∃(group).( (group ∈ network'groups) ∧ (groupId = group'gid) /* InvalidGroupId */ ∧
              (currentPeer ∈ group'members) /* CallerNotInGroup */ )
THEN
     ANY
         networkUpdated
     WHERE
           (networkUpdated ∈ NetworkState) ∧ (networkUpdated ∈ NetworkState) ∧
           (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
           ∃ (group,group2).( (group ∈ network'groups) ∧ (group'gid = groupId) ∧
                              (group2 ∈ Group) ∧ (group2'gid = group'gid) ∧
                              (group2'cred = group'cred) ∧ (group2'gdescr = group'gdescr) ∧
                              (group2'members = group'members - currentPeer ) ∧
                              (
                                (
                                  (group2'members = ∅) ∧
                                  (networkUpdated'groups = network'groups - {group})
                                ) ∨
                                (
                                  (group2'members ≠ ∅) ∧
                                  (networkUpdated'groups = network'groups - {group} ∪ {group2}) )
                                )
                              )
           ) ∧
           (networkUpdated'subscriptions = network'subscriptions) ∧
           (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
           (networkUpdated'sentEvents = network'sentEvents) ∧
           (networkUpdated'invocations = network'invocations) ∧
           (networkUpdated'replies = network'replies)
     THEN
          network ≡ networkUpdated
     END
END
```

Figure B.8: leaveGroup

B.2.6 getIncludingGroups

```
results <-- getIncludingGroups =
results ≡ {groupId |
                   ∃(group).( (group ∈ network'groups) ∧ (currentPeer ∈ group'members) ∧
                              (groupId = group'gid) ) }
```

Figure B.9: getIncludingGroups

B.2.7 getPublishingGroup

```
groupId <-- getPublishingGroup(caller,serviceTypeId) =
PRE
    (serviceTypeId ∈ ServiceTypeId) ∧ (caller ∈ (PeerId ∪ PeerServiceId)) ∧
    ∃(peerCalled,serviceCalled).(
                                  (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                                  (
                                    ( (serviceTypeId ∈ GroupServiceId) ∧
                                      (serviceTypeId = serviceCalled'gsid) ) ∨
                                    ( (serviceTypeId ∈ PeerServiceId) ∧
                                      (serviceTypeId = serviceCalled'psid) ) ∨
                                    ( (serviceTypeId ∈ SessionId) ∧
                                      (serviceTypeId ∈ serviceCalled'sessions) )
                                  ) /* InvalidId */ ∧

                                  /* The caller and the referred service belong to the same group ?*/
                                  (
                                    ( (caller ∈ PeerId) ∧ (caller = currentPeer) ∧
                                      ∃(group).( (group ∈ network'groups) ∧
                                                 (group'gid = serviceCalled'gid) ∧
                                                 (caller ∈ group'members) /* InvalidServiceId */ )
                                    ) ∨
                                    ( (caller ∈ PeerServiceId) ∧
                                      ∃(peerCaller,serviceCaller).( (peerCaller ∈ network'peers) ∧
                                                   (peerCaller'pid = currentPeer) ∧
                                                   (serviceCaller ∈ peerCaller'services) ∧
                                                   /* InvalidServiceId */
                                                   (serviceCaller'gid = serviceCalled'gid) )
                                    )
    )
THEN
     ANY
         gid
     WHERE
           (gid ∈ GroupId) ∧
           ∃(peer,service).((peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                            (
                              ((serviceTypeId ∈ GroupServiceId)∧(serviceTypeId = service'gsid)) ∨
                              ((serviceTypeId ∈ PeerServiceId)∧(serviceTypeId = service'psid)) ∨
                              ((serviceTypeId ∈ SessionId)∧(serviceTypeId ∈ service'sessions))
                            ) ∧
                            (gid = service'gid)
           )
     THEN
          groupId ≡ gid
     END
END
```

Figure B.10: getPublishingGroup

B.2.8 getPeers

```
peers <-- getPeers(groupId, credentials) =
PRE
    (groupId ∈ GroupId) ∧ (credentials ∈ Credentials) ∧
    /* Optional part in the case of a service - InvalidGroupId */
    ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) ∧ (group'cred = credentials) )
THEN
ANY
    peerIdSet
WHERE
      ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) ∧
                 (group'cred = credentials) ∧ (peerIdSet = group'members) )
THEN
     peers ≡ peerIdSet
END
```

Figure B.11: getPeers

B.3 Service Management Primitives

B.3.1 publish

```
newgsid,newpsid <-- publish(groupId, serviceContract, serviceGrounding) =
PRE
    (groupId ∈ GroupId) ∧ (serviceContract ∈ Scontract) /* InvalidServiceSpecification */ ∧
    (serviceGrounding ∈ Sgrounding) /* InvalidGrounding */ ∧
    ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) /* InvalidGroupId */ ∧
               (currentPeer ∈ group'members) /* CallerNotInGroup */ ) ∧
    /* Check that this service has not been already published by the same peer in the same group */
    ∄(peer,service).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                      (service ∈ peer'services) ∧ (service'contract = serviceContract) ∧
                      (service'gid = groupId)
    )
THEN
     ANY
         gsid, psid, networkUpdated, peerUpdated
     WHERE
           (gsid ∈ GroupServiceId) ∧ (psid ∈ PeerServiceId) ∧
           (networkUpdated ∈ NetworkState) ∧ (peerUpdated ∈ Peer) ∧
           /* gsid initialization */
           (
             ∃(peer,service).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                               (serviceContract = service'contract) ∧ (service'gid = groupId) ∧
                               (gsid = service'gsid) ) ∨
             ∄(peer,service).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                               (serviceContract = service'contract) ∧ (service'gid = groupId) ∧
                               (gsid ∈ { gsid2 | (gsid2 ∈ GroupServiceId) ∧
                                         ∄(peer,service).( (peer ∈ network'peers) ∧
                                                           (service ∈ peer'services) ∧
                                                           (service'gsid = gsid2) ) })
             )
           ) ∧

           /* psid initialization */
           ∄(peer,service).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                             (service'psid = psid) ) ∧

           /* update network */
           ∃(peer).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                     (peerUpdated'pid = peer'pid) ∧
                     (peerUpdated'services = peer'services ∪ {rec(groupId, serviceContract,
                                                       serviceGrounding,psid,gsid,∅)}) ∧
                     (networkUpdated'peers = network'peers - {peer} ∪ {peerUpdated})
           ) ∧

           /* other network fields don't change */
           (networkUpdated'cred = network'cred) ∧ (networkUpdated'groups = network'groups) ∧
           (networkUpdated'subscriptions = network'subscriptions) ∧
           (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
           (networkUpdated'sentEvents = network'sentEvents) ∧
           (networkUpdated'invocations = network'invocations) ∧
           (networkUpdated'replies = network'replies)
     THEN network,newgsid,newpsid ≡ networkUpdated,gsid,psid
     END
END
```

Figure B.12: publish

B.3.2 unpublish

```
unpublish(peerServiceId) =
PRE
  (peerServiceId ∈ PeerServiceId) ∧
  ∃(peer,service).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                    (peer'pid = currentPeer) /* CallerNotServiceOwner */ ∧
                    (service'psid = peerServiceId) /* InvalidServiceId */ )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    /* update peers */
    ∃(peerOld,peerUpdated,serviceRemoved).( (peerOld ∈ network'peers) ∧ (peerOld'pid = currentPeer) ∧
          (serviceRemoved ∈ peerOld'services) ∧ (serviceRemoved'psid = peerServiceId) ∧
          (peerUpdated ∈ Peer) ∧ (peerUpdated'services = peerOld'services - {serviceRemoved}) ∧
          (networkUpdated'peers = network'peers - {peerOld} ∪ {peerUpdated}) ) ∧

    (networkUpdated'subscriptions'firstType = {subscription |
          (subscription ∈ network'subscriptions'firstType) ∧
          (subscription'subscriber ≠ peerServiceId) }) ∧
    (networkUpdated'subscriptions'secondType = {subscription |
          (subscription ∈ network'subscriptions'secondType) ∧
          (subscription'subscriber ≠ peerServiceId) }) ∧
    (networkUpdated'subscriptions'thirdType = {subscription |
          (subscription ∈ network'subscriptions'thirdType) ∧
          (subscription'subscriber ≠ peerServiceId) }) ∧
    (networkUpdated'subscriptions'fourthType = {subscription |
          (subscription ∈ network'subscriptions'fourthType) ∧
          (subscription'subscriber ≠ peerServiceId) }) ∧

    (networkUpdated'unsubscriptions'firstType = {unsubscription |
          (unsubscription ∈ network'unsubscriptions'firstType) ∧
          (unsubscription'unsubscriber ≠ peerServiceId) }) ∧
    (networkUpdated'unsubscriptions'firstType = {unsubscription |
          (unsubscription ∈ network'unsubscriptions'firstType) ∧
          (unsubscription'unsubscriber ≠ peerServiceId) }) ∧

    (networkUpdated'unsubscriptions'firstType = {unsubscription |
          (unsubscription ∈ network'unsubscriptions'firstType) ∧
          (unsubscription'unsubscriber ≠ peerServiceId) }) ∧
    (networkUpdated'invocations = {invocation |
          (invocation ∈ network'invocations) ∧
          (invocation'provider ≠ peerServiceId) ∧
          (invocation'caller ≠ peerServiceId) }) ∧

    (networkUpdated'replies'normalResults = {reply | (reply ∈ network'replies'normalResults) ∧
          (reply'caller ≠ peerServiceId) }) ∧
    (networkUpdated'replies'missResults = {reply | (reply ∈ network'replies'missResults) ∧
          (reply'caller ≠ peerServiceId) }) ∧

    /* other network fields don't change */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'groups = network'groups)
    ∧ (networkUpdated'sentEvents = network'sentEvents)
  THEN network ≡ networkUpdated
  END
END
```

Figure B.13: unpublish

B.3.3 getServices

```
results <-- getServices(groupId, peerId, serviceContract, maxResults, credentials) =
PRE
    (groupId ∈ GroupId) ∧ (peerId ∈ PeerId) ∧
    (serviceContract ∈ Scontract) /* InvalidServiceSpecification */ ∧ (credentials ∈ Credentials) ∧
    ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) ) /* optional - InvalidGroupId */ ∧
    ∃(peer).( (peer ∈ network'peers) ∧ (peer'pid = peerId) ) /* optional - InvalidPeerId */
THEN
     ANY
         resultsVar, result
     WHERE
          (resultsVar ⊆ struct(groupId ∈ GroupId, groupServiceId ∈ GroupServiceId,
                               peerServiceId ∈ PeerServiceId) ) ∧
          (result ∈ resultsVar) ∧
          (card(resultsVar) = maxResults) /* optional */ ∧
          ∃(peer, service).( (peer ∈ network'peers) ∧
                             (peer'pid = peerId) /* optional */ ∧
                             (service ∈ peer'services) ∧
                             (serviceContract = service'contract) /* optional */ ∧
                             /* optional */
                             ∃(group).( (group ∈ network'groups) ∧
                                        (group'gid = service'gid) ∧
                                        (group'gid = groupId) ∧
                                        (group'cred = credentials) ) ∧
                             /* if the caller is a service, it is not defined :  */
                             (result'groupId = service'gid) ∧
                             (result'groupServiceId = service'gsid) ∧
                             (result'peerServiceId = service'psid)
          )
     THEN
          results ≡ resultsVar
     END
END
```

Figure B.14: getServices

B.3.4 getServiceContract

```
serviceContract <-- getServicecontract(caller,serviceTypeId) =
PRE
    (serviceTypeId ∈ ServiceTypeId) ∧ (serviceTypeId ∉ SessionId) ∧
    (caller ∈ (PeerId ∪ PeerServiceId)) ∧
    ∃(peerCalled,serviceCalled).( (peerCalled ∈ network'peers) ∧
                 (serviceCalled ∈ peerCalled'services) ∧
                 (
                   ( (serviceTypeId ∈ GroupServiceId) ∧
                     (serviceCalled'gsid = serviceTypeId) ) ∨
                   ( (serviceTypeId ∈ PeerServiceId) ∧
                     (serviceCalled'psid = serviceTypeId) )
                 ) /* InvalidServiceId */ ∧

                 /* The caller and the referred service belong to the same group ?*/
                 (
                   ( (caller ∈ PeerId) ∧ (caller = currentPeer) ∧
                     ∃(group).( (group ∈ network'groups) ∧
                                (group'gid = serviceCalled'gid) ∧
                                (caller ∈ group'members) /* InvalidServiceId */ )
                   ) ∨
                   ( (caller ∈ PeerServiceId) ∧
                     ∃(peerCaller,serviceCaller).( (peerCaller ∈ network'peers) ∧
                                  (peerCaller'pid = currentPeer) ∧
                                  (serviceCaller ∈ peerCaller'services) ∧
                                  /* InvalidServiceId */
                                  (serviceCaller'gid = serviceCalled'gid) )
                   )
                 )
    )
THEN
     ANY
         contract
     WHERE
           ∃(peer,service).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                  (
                    ( (serviceTypeId ∈ GroupServiceId) ∧ (service'gsid = serviceTypeId) ) ∨
                    ( (serviceTypeId ∈ PeerServiceId) ∧ (service'psid = serviceTypeId) )
                  ) ∧
                  (contract = service'contract) )
     THEN
          serviceContract ≡ contract
     END
END
```

Figure B.15: getServiceContract

B.3.5 startSession

```
newSessionId <-- startSession(caller,serviceTypeId) =
PRE
  (serviceTypeId ∈ ServiceTypeId)∧(serviceTypeId ∉ SessionId)∧(caller ∈ (PeerId ∪ PeerServiceId))∧
  ∃(peerCalled,serviceCalled).( (peerCalled ∈ network'peers)∧(serviceCalled ∈ peerCalled'services)∧
        ( /* InvalidServiceId */
          ( (serviceTypeId ∈ GroupServiceId) ∧ (serviceCalled'gsid = serviceTypeId) ) ∨
          ( (serviceTypeId ∈ PeerServiceId) ∧ (serviceCalled'psid = serviceTypeId) )
        ) ∧ (service'contract'type = sessionFull) /* InvalidServiceId */ ∧
        /* The caller and the referred service belong to the same group ?*/
        (
          ( (caller ∈ PeerId) ∧ (caller = currentPeer) ∧
            ∃(group).( (group ∈ network'groups) ∧ (group'gid = serviceCalled'gid) ∧
                       (caller ∈ group'members) /* AccessDenied */ ) ) ∨
          ( (caller ∈ PeerServiceId) ∧
            ∃(peerCaller,serviceCaller).( (peerCaller ∈ network'peers) ∧
                         (peerCaller'pid = currentPeer) ∧ (serviceCaller ∈ peerCaller'services) ∧
                         (serviceCaller'gid = serviceCalled'gid) /* InvalidServiceId */ ) )
        )
  )
THEN
  ANY sessionId, serviceUpdated, networkUpdated
  WHERE
     /* Choose a non-used session identifier */
     ∄(peer, service).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
                        (sessionId ∈ service'sessions) ) ∧
     /* Choose a service associated to this identifier and the associated peer */
     (serviceUpdated ∈ {serviceUpdated | (serviceUpdated ∈ struct(peer ∈ Peer, service ∈ Service)) ∧
                        ∃(peer,service).((peer ∈ network'peers)∧(serviceUpdated'service = service)∧
                               (serviceUpdated'peer = peer) ∧ (service ∈ peer'services) ∧
                               (
                                 ((serviceTypeId ∈ PeerServiceId)∧(serviceTypeId = service'psid)) ∨
                                 ((serviceTypeId ∈ GroupServiceId)∧(serviceTypeId = service'gsid))
                               ) }) ∧
     (networkUpdated ∈ NetworkState) ∧
     /* Updated peer and its updated service */
     ∃(peer2,service2).( (peer2 ∈ Peer) ∧ (peer2'pid = serviceUpdated'peer'pid) ∧
             (service2 ∈ Service) ∧ (service2'gid = serviceUpdated'service'gid) ∧
             (service2'contract = serviceUpdated'service'contract) ∧
             (service2'grounding = serviceUpdated'service'grounding) ∧
             (service2'psid = serviceUpdated'service'psid) ∧
             (service2'gsid = serviceUpdated'service'gsid) ∧
             (service2'sessions = serviceUpdated'service'sessions ∪ {sessionId}) ∧
             (peer2'services = (serviceUpdated'peer'services -
                                {serviceUpdated'service}) ∪ {service2}) ∧
             (networkUpdated'peers = network'peers - {serviceUpdated'peer} ∪ {peer2})
     ) ∧
     /* other network fields are equal */
     (networkUpdated'cred = network'cred) ∧ (networkUpdated'groups = network'groups) ∧
     (networkUpdated'subscriptions = network'subscriptions) ∧
     (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
     (networkUpdated'sentEvents = network'sentEvents) ∧
     (networkUpdated'invocations = network'invocations) ∧ (networkUpdated'replies = network'replies)
  THEN newSessionId,network ≡ sessionId,networkUpdated
  END
END
```

Figure B.16: startSession

B.4 Message Management Primitives

B.4.1 invoke

```
invoke(invokeCaller, serviceTypeId, operationName, input, doReturnResult) =
PRE
  (invokeCaller ∈ CallerId) ∧ (serviceTypeId ∈ ServiceTypeId) ∧
  (operationName ∈ OperationName) ∧ (input ∈ Input) ∧ (doReturnResult ∈ BOOL) ∧
  ∃(peerCaller,peerCalled,serviceCalled,operationCalled).(
      (peerCaller ∈ network'peers) ∧ (peerCaller'pid = currentPeer) ∧
      (peerCalled ∈ network'peers) ∧ (serviceCalled ∈ peerCalled'services) ∧
      (
        ( (serviceTypeId ∈ GroupServiceId) ∧ (serviceCalled'gsid = serviceTypeId) ∧
          (serviceCalled'contract'type = stateLess) /* InvalidServiceId */) ∨
        ( (serviceTypeId ∈ PeerServiceId) ∧ (serviceCalled'psid = serviceTypeId) ∧
          (serviceCalled'contract'type ≠ sessionFull) /* InvalidServiceId */ ) ∨
        ( (serviceTypeId ∈ SessionId) ∧ (serviceTypeId ∈ serviceCalled'sessions) ∧
          (serviceCalled'contract'type = sessionFull) /* InvalidSessionId */)
      ) ∧ (operationCalled ∈ serviceCalled'contract'operations) ∧
      (operationCalled'name = operationName) /* InvalidOperation */ ∧
      (
        ( (invokeCaller ∈ PeerId) ∧ (invokeCaller ∈ PeerId) ∧ (invokeCaller = currentPeer) ∧
          ∃(group).( (group ∈ network'groups) ∧ (currentPeer ∈ group'members) /* AccessDenied */ ∧
                     (serviceCalled'gid = group'gid) ) ) ∨
        ( (invokeCaller ∉ PeerId) ∧
          ∃(serviceCaller).( (serviceCaller ∈ peerCaller'services) ∧
              (serviceCaller'gid = serviceCalled'gid) /* AccessDenied */ ∧
              ( ( (invokeCaller ∈ SessionId) ∧ (invokeCaller ∈ serviceCaller'sessions) ∧
                  (serviceCaller'contract'type = sessionFull) ) ∨
                  ( (invokeCaller ∈ PeerServiceId) ∧ (invokeCaller = serviceCaller'psid) ∧
                    (serviceCaller'contract'type ≠ sessionFull) )
              ) )
        )
      ) ∧ ( (doReturnResult = TRUE) ⇒ (operationCalled'type = requestResponse) ) ∧
      ( (operationCalled'type = requestResponse) ∧ (serviceTypeId ∈ GroupServiceId) ⇒
        ∄(invocation).( (invocation ∈ network'invocations) ∧ (invocation'caller = invokeCaller) ∧
         (invocation'callee = serviceTypeId)∧(invocation'op = operationName)/*ConcurrentRequest*/)
  ) ∧ /* InvalidServiceId :*/
  ( (operationCalled'type = requestResponse) ∧ (serviceTypeId ∈ GroupServiceId) ⇒
      ∄(invocation).( (invocation ∈ network'invocations) ∧ (invocation'caller = invokeCaller) ∧
           (invocation'callee = serviceTypeId) ∧ (invocation'op = operationName) ) ∨
      (card({service | ∃(peer).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
         ∄(invocation).( (invocation ∈ network'invocations) ∧ (invocation'caller = invokeCaller) ∧
           (invocation'op = operationName) ∧ (invocation'provider = service'psid) ) ) }) > 1) ) ∧
         (service'gsid = serviceTypeId)
THEN
  ANY networkUpdated
  WHERE
      (networkUpdated ∈ NetworkState) ∧ (networkUpdated'sentEvents = network'sentEvents) ∧
      (networkUpdated'invocations = network'invocations
              ∪ {rec(invokeCaller,serviceTypeId,empty,operationName,input)}) ∧
      (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
      (networkUpdated'groups = network'groups)∧ (networkUpdated'replies = network'replies) ∧
      (networkUpdated'subscriptions = network'subscriptions) ∧
      (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
  THEN network ≡ networkUpdated END
END
```

Figure B.17: invoke

B.4.2 receiveMessage

```
callerId, msgInput ← receiveMessage(calleeId, operationName) =
PRE
  (calleeId ∈ ServiceId) ∧ (operationName ∈ OperationName) ∧
  ∃(peer,service,operation).( (peer ∈ network'peers) ∧ (service ∈ peer'services) ∧
      (peer'pid = currentPeer) ∧ (operation ∈ service'contract'operations) ∧
      (operation'name = operationName) /* InvalidOperation */ ∧ (operation'type = requestResponse) ∧
      (
        ( (calleeId ∈ SessionId) ∧ (calleeId ∈ service'sessions) ∧
          (service'contract'type = sessionFull) ) ∨
        ( (calleeId ∈ PeerServiceId) ∧ (calleeId = service'psid) ∧
          (service'contract'type ≠ sessionFull) ) )
  )
THEN
  ANY invocation, networkUpdated
  WHERE
      (networkUpdated ∈ NetworkState) ∧ (invocation ∈ network'invocations) ∧
      (invocation'provider = empty) ∧
      /* ConcurrentRequest already checked by the invoke primitive */
      ( (invocation'callee ∉ GroupServiceId) ⇒ (invocation'callee = calleeId) ) ∧
      ( (invocation'callee ∈ GroupServiceId) ⇒ (
           ∃(peer,service).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                (service ∈ peer'services) ∧
                (calleeId = service'psid) ∧
                (service'gsid = invocation'callee) ∧
                ∄(invocation2).( (invocation2 ∈ network'invocations) ∧
                     (invocation2'caller = invocation'caller) ∧
                     (invocation2'provider = calleeId) ∧
                     (invocation2'op = invocation'op) )
           )
      ) ) ∧
      (networkUpdated'invocations = network'invocations - {invocation}
              ∪ {rec(invocation'caller,invocation'callee,invocation'provider,
                     invocation'op,invocation'in)} ) ∧
      /* other network fields don't change */
      (networkUpdated'cred = network'cred) ∧
      (networkUpdated'peers = network'peers) ∧
      (networkUpdated'groups = network'groups)∧
      (networkUpdated'subscriptions = network'subscriptions) ∧
      (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
      (networkUpdated'replies = network'replies) ∧
      (networkUpdated'sentEvents = network'sentEvents)
  THEN
      callerId,msgInput,network ≡ invocation'caller,invocation'in,networkUpdated
  END
END
```

Figure B.18: receiveMessage

B.4.3 reply

```
reply(calleeId, callerId, operationName, output, faultName) =
PRE
  (calleeId ∈ ServiceId) ∧ (callerId ∈ CallerId) ∧ (operationName ∈ OperationName) ∧
  (output ∈ Output) ∧ (faultName ∈ (FaultName ∃ {empty})) ∧
  ∃(peerCalled,serviceCalled,operation).( (peerCalled ∈ network'peers) ∧
      (peerCalled'pid = currentPeer) ∧ (serviceCalled ∈ peerCalled'services) ∧
      (operation ∈ serviceCalled'contract'operations) ∧ (operation'type = requestResponse) ∧
      (operation'name = operationName) /* InvalidOperation */ ∧
      (
        ( (calleeId ∈ SessionId) ∧ (calleeId ∈ serviceCalled'sessions) ∧
          (serviceCalled'contract'type = sessionFull) ) ∨
        ( (calleeId ∈ PeerServiceId) ∧ (calleeId = serviceCalled'psid) ∧
          (serviceCalled'contract'type ≠ sessionFull) )
      ) ∧
      ∃(peerCaller).( (peerCaller ∈ network'peers) ∧
        ( (callerId ∈ PeerId) ⇒ (peerCaller'pid = callerId) /* InvalidPeerId */ ∧
          ∃(group).( (group ∈ network'groups) ∧ (serviceCalled'gid = group'gid) ∧
              (peerCaller'pid ∈ group'members) ) ) ∧
        ( (callerId ∈ ServiceId) ⇒
          ∃(serviceCaller).( (serviceCaller ∈ peerCaller'services) ∧
              (serviceCaller'gid = serviceCalled'gid) ∧
              (
              ( (callerId ∈ SessionId) ∧ (callerId ∈ serviceCaller'sessions) ∧
                (serviceCaller'contract'type = sessionFull) ) /* InvalidSessionId */ ∨
              ( (callerId ∈ PeerServiceId) /* InvalidServiceId */ ∧ (callerId = serviceCaller'psid) ∧
                (serviceCaller'contract'type ≠ sessionFull) )
          ) )
        )
      )
  ) ∧
  /* MissingInvokeMessage */
  ∃(invocation).( (invocation ∈ network'invocations) ∧ (invocation'caller = callerId) ∧
      (invocation'provider = calleeId) ∧ (invocation'op = operationName) ) ∧
  ∃(result).( (result ∈ network'replies'normalResults) ∧ (result'caller = callerId) ∧
      (result'callee = calleeId) ∧ (result'op = operationName) ) ∧
  ∃(result).( (result ∈ network'replies'missResults) ∧ (result'caller = callerId) ∧
      (result'callee = calleeId) ∧ (result'op = operationName) )
THEN
  ANY networkUpdated
  WHERE
      (networkUpdated ∈ NetworkState) ∧
      (
        ( (faultName = empty) ∧ (networkUpdated'replies'normalResults = network'replies'normalResults
                                   ∪ {rec(calleeId,callerId,operationName,output)}) ) ∨
        ( (faultName ≠ empty) ∧ (networkUpdated'replies'missResults = network'replies'missResults ∪
                                   {rec(calleeId,callerId,operationName,output,faultName)}) )
      ) ∧
      (networkUpdated'cred = network'cred) ∧ (networkUpdated'sentEvents = network'sentEvents)
      (networkUpdated'peers = network'peers) ∧ (networkUpdated'groups = network'groups)∧
      (networkUpdated'subscriptions = network'subscriptions) ∧
      (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
  THEN output,network ≡ resultOutput,networkUpdated
  END
END
```

Figure B.19: reply

B.4.4 receiveResponse

```
output <-- receiveResponse(callerId, serviceTypeId, operationName) =
PRE (callerId ∈ CallerId) ∧ (serviceTypeId ∈ ServiceTypeId) ∧ (operationName ∈ OperationName) ∧
 ∃(peerCalled,serviceCalled,operation).( (peerCalled ∈ network'peers) ∧
   (serviceCalled ∈ peerCalled'services) ∧ (operation ∈ serviceCalled'contract'operations) ∧
   (operation'name = operationName) /* InvalidOperation */ ∧ (operation'type = requestResponse) ∧
   (
     ( /* InvalidSessionId */ (serviceTypeId ∈ SessionId) ∧ (serviceTypeId ∈ serviceCalled'sessions) ∧
       (serviceCalled'contract'type = sessionFull) ) ∨
     ( /* InvalidServiceId */ (serviceTypeId ∈ PeerServiceId) ∧ (serviceTypeId = serviceCalled'psid) ∧
       (serviceCalled'contract'type ≠ sessionFull) ) ∨
     ( /* InvalidServiceId */ (serviceTypeId ∈ GroupServiceId) ∧ (serviceTypeId = serviceCalled'gsid) ∧
       (serviceCalled'contract'type = stateLess) )
   ) ∧
   ∃(peerCaller).( (peerCaller ∈ network'peers) ∧
       ( (callerId ∈ PeerId) ⇒ (peerCaller'pid = callerId) ∧ ∃(group).( (group ∈ network'groups) ∧
             (serviceCalled'gid = group'gid) ∧ (peerCaller'pid ∈ group'members) ) ) ∧
       ( (callerId ∈ ServiceId) ⇒ ∃(serviceCaller).( (serviceCaller ∈ peerCaller'services) ∧
             (serviceCaller'gid = serviceCalled'gid) ∧
             (
               ( (callerId ∈ SessionId) ∧ (callerId ∈ serviceCaller'sessions) ∧
                 (serviceCaller'contract'type = sessionFull) ) ∨
               ( (callerId ∈ PeerServiceId) ∧ (callerId = serviceCaller'psid) ∧
                 (serviceCaller'contract'type ≠ sessionFull) )
             ) )
       ) )
 ) ∧ ∃(invocation).( (invocation ∈ network'invocations) ∧ (invocation'caller = callerId) ∧
      (invocation'callee = serviceTypeId) ∧ (invocation'op = operationName) ) /* MissingInvokeMessage */
THEN
 ANY networkUpdated WHERE
      (resultOutput ∈ struct(fault ∈ (StringNames ∪ {empty}), output ∈ Output)) ∧
      (
        ∃(reply).( (reply ∈ network'replies'normalResults) ∧
            (reply'callee = serviceTypeId) ∧ (reply'caller = callerId) ∧
            (reply'op = operationName) ∧ (resultOutput'fault = empty) ∧
            (resultOutput'output = reply'out) ) ∨
        ∃(reply).( (reply ∈ network'replies'missResults) ∧
            (reply'callee = serviceTypeId) ∧ (reply'caller = callerId) ∧
            (reply'op = operationName) ∧ (resultOutput'fault = reply'fault) ∧
            (resultOutput = rec(reply'fault, reply'out) ) )
      ) ∧ (networkUpdated ∈ NetworkState) ∧
      (networkUpdated'invocations = {invoke | (invoke ∈ network'invocations) ∧
          ( (invoke'caller ≠ callerId) ∨ (invoke'callee ≠ serviceTypeId) ∨
            (invoke'op ≠ operationName) ) }) ∧
      (networkUpdated'replies'missResults = {result | (result ∈ network'replies'normalResults) ∧
          ( (result'caller ≠ callerId) ∨ (result'callee ≠ serviceTypeId) ∨
            (result'op ≠ operationName) ) }) ∧
      (networkUpdated'replies'missResults = {result | (result ∈ network'replies'missResults) ∧
          ( (result'caller ≠ callerId) ∨ (result'callee ≠ serviceTypeId) ∨
            (result'op ≠ operationName) ) }) ∧
      (networkUpdated'cred = network'cred) ∧ (networkUpdated'sentEvents = network'sentEvents)
      (networkUpdated'peers = network'peers) ∧ (networkUpdated'groups = network'groups)∧
      (networkUpdated'subscriptions = network'subscriptions) ∧
      (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
 THEN output,network ≡ resultOutput,networkUpdated END
END
```

Figure B.20: receiveResponse

B.5 Event Management Primitives

B.5.1 event

```
event(callerId, groupId, eventName, input) =
PRE
    (callerId ∈ CallerId) ∧ (groupId ∈ (GroupId ∪ {empty})) ∧
    (eventName ∈ EventName) ∧ (input ∈ Input) ∧
    ( (callerId ∈ PeerId) ⇒ (callerId = currentPeer) ∧
      ( (groupId ≠ empty) ⇒ ∃(group).( (group ∈ network'groups) ∧
                                  (group'gid = groupId) /* InvalidGroupId */ ∧
                                  (callerId ∈ group'members) /* CallerNotInGroup */ ) )
    ) ∧
    ( (callerId ∉ PeerId) ⇒ (groupId = empty) ∧
          ∃(peerCaller,serviceCaller).( (peerCaller ∈ network'peers) ∧
            (peerCaller'pid = currentPeer)∧(serviceCaller ∈ peerCaller'services) ∧
            ( /* Used because callerId has been added */
              ( (callerId ∈ SessionId) ∧ (callerId ∈ serviceCaller'sessions) ∧
                (serviceCaller'contract'type = sessionFull) ) ∨
              ( (callerId ∈ PeerServiceId) ∧ (callerId = serviceCaller'psid) ∧
                (serviceCaller'contract'type ≠ sessionFull) )
            ) ∧ (eventName ∈ serviceCaller'contract'eventsRaised) /* InvalidEvent */
          )
    )
THEN
     ANY networkUpdated
     WHERE
           (networkUpdated ∈ NetworkState) ∧ (sentGroups ⊆ GroupId) ∧
           /* sentGroups initialization */
           ( (callerId ∉ PeerId) ⇒
             ∃(peerCaller,serviceCaller).( (peerCaller ∈ network'peers) ∧
                (serviceCaller ∈ peerCaller'services) ∧
                (
                  ( (callerId ∈ SessionId) ∧ (callerId ∈ serviceCaller'sessions) ) ∨
                  ( (callerId ∈ PeerServiceId) ∧ (callerId = serviceCaller'psid) )
                ) ∧ (sentGroups = {serviceCaller'gid}) )
           ) ∧
           ( (callerId ∈ PeerId) ∧ (groupId = empty) ⇒ (sentGroups = groupId)) ∧
           ( (callerId ∈ PeerId) ∧ (groupId ≠ empty) ⇒
             (sentGroups = {groupId | (groupId ∈ GroupId) ∧
                ∃(group).( (group ∈ network'groups) ∧
                   (group'gid = groupId) ∧ (callerId ∈ group'members) )})
           ) ∧
           /* add new event */
           (networkUpdated'sentEvents = network'sentEvents
                                      ∪ {rec(eventName,input,sentGroups,callerId)}) ∧
           (networkUpdated'cred = network'cred) ∧
           (networkUpdated'peers = network'peers) ∧
           (networkUpdated'groups = network'groups)∧
           (networkUpdated'subscriptions = network'subscriptions) ∧
           (networkUpdated'unsubscriptions = network'unsubscriptions) ∧
           (networkUpdated'invocations = network'invocations) ∧
           (networkUpdated'replies = network'replies)
     THEN network ≡ networkUpdated
     END
END
```

Figure B.21: event Primitive for Peers (no groupId)

B.5.2 receiveEvent

```
eventSenderId, input <-- receiveEvent(callerId, groupId, eventName) =
PRE
    (callerId ∈ (PeerId ∪ PeerServiceId)) ∧ (groupId ∈ (GroupId ∪ empty)) ∧
    (eventName ∈ EventName) ∧
    ( (callerId ∈ PeerId) ⇒ (callerId = currentPeer) ∧
      ( (groupId ≠ empty) ⇒ ∃(group).( (group ∈ network'groups) ∧
                                        (group'gid = groupId) /* InvalidGroupId */ ∧
                                        (callerId ∈ group'members) /* CallerNotInGroup */ ) )
    ) ∧
    ( (callerId ∈ PeerServiceId) ∧ (groupId = empty) /* InvalidCall */ ∧
      ∃(peer,service).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                        (service ∈ peer'services) ∧ (service'psid = callerId))
    )
THEN
     ANY event, gid
     WHERE
           (event ∈ network'sentEvents) ∧ (gid ∈ GroupId) ∧ ((groupId ≠ empty)⇒(gid = groupId)) ∧
           ( (groupId = empty) ⇒ (gid ∈ event'groups) )∧
           ( (callerId ∈ PeerId) ⇒ ∃(group).( (group ∈ network'groups) ∧ (group'gid = gid) ∧
                                              (callerId ∈ group'members) ) ) ∧
           ( (callerId ∈ PeerServiceId) ⇒
             ∃(peer,service).( (peer ∈ network'peers) ∧
                               (peer'pid = currentPeer) ∧ (service ∈ peer'services) ∧
                               (service'psid = callerId) ∧ (service'gid = gid) ) ) ∧
           (
             ∃(subscription1).( (subscription1 ∈ network'subscriptions'firstType) ∧
                                (subscription1'event = eventName) ∧
                                (subscription1'subscriber = callerId) ∧
                                (subscription1'group = gid) ) ∨
             (
               (
                 ∃(subscription2).( (subscription2 ∈ network'subscriptions'secondType) ∧
                                    (subscription2'event = eventName) ∧
                                    (subscription2'subscriber = callerId) ) ∨
                 ∃(subscription3).( (subscription3 ∈ network'subscriptions'thirdType) ∧
                                    (subscription3'group = gid) ∧
                                    (subscription3'subscriber = callerId) ) ∨
                 ∃(subscription4).( (subscription4 ∈ network'subscriptions'fourthType) ∧
                                    (subscription4'subscriber = callerId) )
               ) ∧
               ∄(unsubscription1).( (unsubscription1 ∈ network'unsubscriptions'firstType) ∧
                                    (unsubscription1'event = eventName) ∧
                                    (unsubscription1'unsubscriber = currentPeer) ) ∧
               ∄(unsubscription2).( (unsubscription2 ∈ network'unsubscriptions'secondType) ∧
                                    (unsubscription2'event = eventName) ∧
                                    (unsubscription2'group = gid) ∧
                                    (unsubscription2'unsubscriber = currentPeer) ) ∧
               ∄(unsubscription3).( (unsubscription3 ∈ network'unsubscriptions'thirdType) ∧
                                    (unsubscription3'group = gid) ∧
                                    (unsubscription3'unsubscriber = currentPeer) )
             )
           )
     THEN eventSenderId,input ≡ event'sender,event'in END
END
```

Figure B.22: receiveEvent

B.5.3 subscribe

```
subscribeFirstType(eventName, groupId) =
PRE
    (eventName ∈ EventName) ∧ (groupId ∈ GroupId) ∧
    ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) /* InvalidGroupId */ ∧
               (currentPeer ∈ group'members) /* CallerNotInGroup */ )
THEN
  ANY networkUpdated
  WHERE
    /* update subscription only if there is no more generic subscription */
    (networkUpdated ∈ NetworkState) ∧
    (
      (
        (
          ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                            (subscription'event = eventName) ∧
                            (subscription'subscriber = currentPeer) ) ∨
          ∃(subscription).( (subscription ∈ network'subscriptions'thirdType) ∧
                            (subscription'group = groupId) ∧
                            (subscription'subscriber = currentPeer) ) ∨
          ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                            (subscription'subscriber = currentPeer) )
        ) ∧
        ∄(unsubscription).( (unsubscription ∈ network'unsubscriptions'firstType) ∧
                            (unsubscription'unsubscriber = currentPeer) ∧
                            (unsubscription'event = eventName) ) ∧
        ∄(unsubscription).( (unsubscription ∈ network'unsubscriptions'thirdType) ∧
                            (unsubscription'unsubscriber = currentPeer) ∧
                            (unsubscription'group = groupId) ) ∧
        (networkUpdated'subscriptions'firstType = network'subscriptions'firstType)
      ) ∨
      networkUpdated'subscriptions'firstType = network'subscriptions'firstType
                                              ∪ {rec(eventName, groupId, currentPeer)}
    ) ∧
    (networkUpdated'subscriptions'secondType = network'subscriptions'secondType) ∧
    (networkUpdated'subscriptions'thirdType = network'subscriptions'thirdType) ∧
    (networkUpdated'subscriptions'fourthType = network'subscriptions'fourthType) ∧

    (networkUpdated'unsubscriptions'firstType = network'unsubscriptions'firstType) ∧
    /* sub1(E,G,caller) ∧ usub2(E,G,caller) ⇒ FALSE */
    (networkUpdated'unsubscriptions'secondType =
                {unsubscription | (unsubscription ∈ network'unsubscriptions'secondType) ∧
                                  (
                                    (unsubscription'unsubscriber ≠ currentPeer) ∨
                                    (unsubscription'event ≠ eventName) ∨
                                    (unsubscription'group ≠ groupId)
                                  ) } ) ∧
    (networkUpdated'unsubscriptions'thirdType = network'unsubscriptions'thirdType) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧ (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧ (networkUpdated'replies = network'replies)
  THEN network ≡ networkUpdated
  END
END
```

Figure B.23: subscribe - First Type

Table B.1: Subscriptions State Evolution

	Current state	Action	Evolution
1	...	sub4(caller)	∪ {sub4(caller)} -{sub1(*,*,caller), sub2(*,caller), sub3(*,caller)}
1	∅	sub3(G,caller)	∪ {sub3(G,caller)}
2	sub4(caller)	sub3(G,caller)	SKIP
3	sub2(E,caller)	sub3(G,caller)	∪ {sub3(G,caller)}
4	sub1(E,G,caller)	sub3(G,caller)	∪ {sub3(G,caller)} -{(sub1(E,G,caller)}
5	sub4(caller) + usub1(E,caller)	sub3(G,caller)	∪ {sub1(E,G,caller)}
6	sub3(G,caller) + usub1(E,caller)	sub3(G,caller)	∪ {sub1(E,G,caller)}
7	sub4(caller) + usub2(E,G,caller)	sub3(G,caller)	-{usub2(E,G,caller)}
8	sub3(G,caller) + usub2(E,G,caller)	sub3(G,caller)	- {usub2(E,G,caller)}
9	sub2(E,caller) + usub2(E,G,caller)	sub3(G,caller)	∪ {sub3(G,caller)} - {usub2(E,G,caller)}
10	sub4(caller) + usub3(G,caller)	sub3(G,caller)	- {usub3(G,caller)}
11	sub2(E,caller) + usub3(G,caller)	sub3(G,caller)	∪ {sub3(G,caller)} - {usub3(G,caller)}
1	∅	sub2(E,caller)	∪ {sub2(E,caller)}
2	sub4(caller)	sub2(E,caller)	SKIP
3	sub3(G,caller)	sub2(E,caller)	∪ {sub2(E,caller)}
4	sub1(E,G,caller)	sub2(E,caller)	∪ {sub2(E,caller)} -{(sub1(E,G,caller)}
5	sub4(caller) + usub1(E,caller)	sub2(E,caller)	- {usub1(E,caller)}
6	sub3(G,caller) + usub1(E,caller)	sub2(E,caller)	∪ {sub2(E,caller)} - {usub1(E,caller)}
7	sub4(caller) + usub2(E,G,caller)	sub2(E,caller)	- {usub2(E,G,caller)
8	sub3(G,caller) + usub2(E,G,caller)	sub2(E,caller)	∪ {sub2(E,caller)} - {usub2(E,G,caller)}
9	sub2(E,caller) + usub2(E,G,caller)	sub2(E,caller)	- {usub2(E,G,caller)}
10	sub4(caller) + usub3(G,caller)	sub2(E,caller)	∪ {sub1(E,G,caller)}
11	sub2(E,caller) + usub3(G,caller)	sub2(E,caller)	∪ {sub1(E,G,caller)}
1	∅	sub1(E,G,caller)	∪ {sub1(E,G,caller)}
2	sub4(caller)	sub1(E,G,caller)	SKIP
3	sub3(G,caller)	sub1(E,G,caller)	SKIP
4	sub2(E,caller)	sub1(E,G,caller)	SKIP
5	sub4(caller) + usub1(E,caller)	sub1(E,G,caller)	∪ {sub1(E,G,caller)}
6	sub3(G,caller) + usub1(E,caller)	sub1(E,G,caller)	∪ {sub1(E,G,caller)}
7	sub4(caller) + usub2(E,G,caller)	sub1(E,G,caller)	- {usub2(E,G,caller)}
8	sub3(G,caller) + usub2(E,G,caller)	sub1(E,G,caller)	- {usub2(E,G,caller)}
9	sub2(E,caller) + usub2(E,G,caller)	sub1(E,G,caller)	- {usub2(E,G,caller)}
10	sub4(caller) + usub3(G,caller)	sub1(E,G,caller)	∪ {sub1(E,G,caller)}
11	sub2(E,caller) + usub3(G,caller)	sub1(E,G,caller)	∪ {sub1(E,G,caller)}

```
subscribeSecondType(caller, eventName) =
PRE
    (caller ∈ (PeerId ∪ PeerServiceId)) ∧ (eventName ∈ EventName) ∧
    ( /* Used because receiveResponseCaller has been added */
      ∃(peer,service).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                        (service ∈ peer'services) ∧ (service'psid = caller) ) ∨
      (caller = currentPeer) )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    (
      ( /* see line 10 and 11 */
        ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'thirdType) ∧
                            (unsubscription'unsubscriber = caller) ) ∧
        (networkUpdated'subscriptions'firstType = network'subscriptions'firstType ∪
            {subscription | (subscription ∈ SubscriptionEventInAGroup) ∧
                ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'thirdType) ∧
                    (unsubscription'unsubscriber = caller) ∧
                    (subscription'group = unsubscription'group) ∧
                    (subscription'event = eventName) ∧ (subscription'subscriber = caller) ) } )
      ) ∧
      /* see line 4, 7, 8 and 9 */
      (networkUpdated'subscriptions'firstType = {subscription |
          (subscription ∈ network'subscriptions'firstType) ∧
          ( (subscription'subscriber ≠ caller) ∨ (subscription'event ≠ eventName) ) } )
    ) ∧
    ( /* see line 2, 5, 7, 9, 10 and 11 */
      (
        ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
             (subscription'subscriber = caller) ) ∨
        (
          ∃(subscription).((subscription ∈ network'subscriptions'secondType) ∧
               (subscription'event = eventName) ∧ (subscription'subscriber = caller) ) ∧
          (
            ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'thirdType) ∧
                (unsubscription'unsubscriber = caller) ) ∨
            ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'secondType) ∧
                (unsubscription'event = eventName) ∧ (unsubscription'unsubscriber = caller) )
          )
        ) ∧ (networkUpdated'subscriptions'secondType = network'subscriptions'secondType)
      ) ∨
      /* see line 1, 3, 4, 6 and 8 */
      networkUpdated'subscriptions'secondType = network'subscriptions'secondType
                                              ∪ {rec(eventName,caller)}
    ) ∧
    (networkUpdated'subscriptions'thirdType = network'subscriptions'thirdType) ∧
    (networkUpdated'subscriptions'fourthType = network'subscriptions'fourthType) ∧
    /* sub2(E,caller) ∧ usub1(E,caller) ⇒ FALSE */
    (networkUpdated'unsubscriptions'firstType = {unsubscription |
        (unsubscription ∈ network'unsubscriptions'firstType) ∧
        ((unsubscription'unsubscriber ≠ caller)∨(unsubscription'event ≠ eventName)) } ) ∧
    /* usub2(E,G,caller) followed by sub2(E,caller) ⇒ remove usub2(E,G,caller) */
    (networkUpdated'unsubscriptions'secondType = {unsubscription |
        (unsubscription ∈ network'unsubscriptions'secondType) ∧
        ( (unsubscription'unsubscriber ≠ caller)∨(unsubscription'event ≠ eventName) ) })∧
    (networkUpdated'unsubscriptions'thirdType = network'unsubscriptions'thirdType) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧ (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧ (networkUpdated'replies = network'replies)
  THEN network ≡ networkUpdated END
END
```

Figure B.24: subscribe - Second Type

```
subscribeThirdType(groupId) =
PRE
    (groupId ∈ GroupId) ∧
    ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) /* InvalidGroupId */ ∧
               (currentPeer ∈ group'members) /* CallerNotInGroup */ )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    (
      ( /* see line 5 and 6 */
        ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'firstType) ∧
                            (unsubscription'unsubscriber = currentPeer) ) ∧
        (networkUpdated'subscriptions'firstType = network'subscriptions'firstType ∪
            {subscription | (subscription ∈ SubscriptionEventInAGroup) ∧
                ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'firstType) ∧
                (unsubscription'unsubscriber = currentPeer) ∧
                (subscription'event = unsubscription'event) ∧
                (subscription'group = groupId) ∧
                (subscription'subscriber = currentPeer) ) } )
      ) ∨
      /* see line 4, 7, 8 and 9 */
      (networkUpdated'subscriptions'firstType = {subscription |
          (subscription ∈ network'subscriptions'firstType) ∧
          ( (subscription'subscriber ≠ currentPeer) ∨ (subscription'group ≠ groupId) ) } )
    ) ∧ (networkUpdated'subscriptions'secondType = network'subscriptions'secondType) ∧
    ( /* see line 2, 5, 6, 7, 8 and 10 */
      (
        ∃(subscription).((subscription ∈ network'subscriptions'fourthType) ∧
                         (subscription'subscriber = currentPeer) ) ∨
        (
          ∃(subscription).((subscription ∈ network'subscriptions'thirdType) ∧
                           (subscription'group = groupId) ∧
                           (subscription'subscriber = currentPeer) ) ∧
          ( ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'secondType) ∧
                                (unsubscription'group = groupId) ∧
                                (unsubscription'unsubscriber = currentPeer) ) ∨
          ∃(unsubscription).( (unsubscription ∈ network'unsubscriptions'firstType) ∧
                              (unsubscription'unsubscriber = currentPeer) ) )
        ) ∧
        (networkUpdated'subscriptions'thirdType = network'subscriptions'thirdType)
      ) ∨
      /* see line 1, 3, 4, 9 and 11 */
      networkUpdated'subscriptions'thirdType = network'subscriptions'thirdType
                    ∪ {rec(groupId,currentPeer)}
    ) ∧
    (networkUpdated'subscriptions'fourthType = network'subscriptions'fourthType) ∧
    (networkUpdated'unsubscriptions'firstType = network'unsubscriptions'firstType) ∧
    /* sub3(G,caller) ∧ usub3(G,caller) ⇒ FALSE */
    (networkUpdated'unsubscriptions'thirdType = {unsubscription |
        (unsubscription ∈ network'unsubscriptions'thirdType) ∧
        ((unsubscription'unsubscriber ≠ currentPeer)∨(unsubscription'group ≠ groupId)) } ) ∧
    /* usub2(E,G,caller) followed by sub3(G,caller) ⇒ remove usub2(E,G,caller) */
    (networkUpdated'unsubscriptions'secondType = {unsubscription |
        (unsubscription ∈ network'unsubscriptions'secondType) ∧
        ((unsubscription'unsubscriber ≠ currentPeer)∨(unsubscription'group ≠ groupId)) } ) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧ (networkUpdated'replies = network'replies)
    (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧
  THEN network ≡ networkUpdated END
END
```

Figure B.25: subscribe - Third Type

```
subscribeFourthType(caller) =
PRE
    (caller ∈ (PeerId ∪ PeerServiceId)) ∧
    /* Used because caller has been added */
    (
      ∃(peer,service).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                        (service ∈ peer'services) ∧ (service'psid = caller) ) ∨
      (caller = currentPeer)
    )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    (networkUpdated'subscriptions'firstType = {subscription |
                    (subscription ∈ network'subscriptions'firstType) ∧
                    (subscription'subscriber ≠ caller) }) ∧
    (networkUpdated'subscriptions'secondType = {subscription |
                    (subscription ∈ network'subscriptions'secondType) ∧
                    (subscription'subscriber ≠ caller) }) ∧
    (networkUpdated'subscriptions'thirdType = {subscription |
                    (subscription ∈ network'subscriptions'thirdType) ∧
                    (subscription'subscriber ≠ caller) }) ∧
    (networkUpdated'subscriptions'fourthType = network'subscriptions'fourthType
                                             ∪ {rec(caller)}) ∧
    (networkUpdated'unsubscriptions'firstType = {unsubscription |
                   (unsubscription ∈ network'unsubscriptions'firstType) ∧
                   (unsubscription'unsubscriber ≠ caller) } ) ∧
    (networkUpdated'unsubscriptions'secondType = {unsubscription |
                   (unsubscription ∈ network'unsubscriptions'secondType) ∧
                   (unsubscription'unsubscriber ≠ caller) } ) ∧
    (networkUpdated'unsubscriptions'thirdType = {unsubscription |
                   (unsubscription ∈ network'unsubscriptions'thirdType) ∧
                   (unsubscription'unsubscriber ≠ caller) } ) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧
    (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧
    (networkUpdated'replies = network'replies)
  THEN network ≡ networkUpdated
  END
END
```

Figure B.26: subscribe - Fourth Type

B.5.4 unsubscribe

```
unsubscribeFourthType(caller) =
PRE
    (caller ∈ (PeerId ∪ PeerServiceId)) ∧
    /* Used because caller has been added */
    (
      ∃(peer,service).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                        (service ∈ peer'services) ∧ (service'psid = caller) ) ∨
      (caller = currentPeer)
    ) ∧
    /* NotSubscribed */
    (
      ∃(subscription).( (subscription ∈ network'subscriptions'firstType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'thirdType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                        (subscription'subscriber = caller) )
    )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    (networkUpdated'subscriptions'firstType = {subscription |
                    (subscription ∈ network'subscriptions'firstType) ∧
                    (subscription'subscriber ≠ caller) }) ∧
    (networkUpdated'subscriptions'secondType = {subscription |
                    (subscription ∈ network'subscriptions'secondType) ∧
                    (subscription'subscriber ≠ caller) }) ∧
    (networkUpdated'subscriptions'thirdType = {subscription |
                    (subscription ∈ network'subscriptions'thirdType) ∧
                    (subscription'subscriber ≠ caller) }) ∧
    (networkUpdated'subscriptions'fourthType = {subscription |
                    (subscription ∈ network'subscriptions'fourthType) ∧
                    (subscription'subscriber ≠ caller) }) ∧
    (networkUpdated'unsubscriptions'firstType = {unsubscription |
                    (unsubscription ∈ network'unsubscriptions'firstType) ∧
                    (unsubscription'unsubscriber ≠ caller) } ) ∧
    (networkUpdated'unsubscriptions'secondType = {unsubscription |
                    (unsubscription ∈ network'unsubscriptions'secondType) ∧
                    (unsubscription'unsubscriber ≠ caller) } ) ∧
    (networkUpdated'unsubscriptions'thirdType = {unsubscription |
                    (unsubscription ∈ network'unsubscriptions'thirdType) ∧
                    (unsubscription'unsubscriber ≠ caller) } ) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧
    (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧
    (networkUpdated'replies = network'replies)
  THEN network ≡ networkUpdated
  END
END
```

Figure B.27: unsubscribe - Fourth Type

Table B.2: Unsubscriptions Evolution

	Current state	Action	Evolution
1	sub1(E,G,caller)	usub1(E,caller)	-{sub1(E,G,caller)}
2	sub2(E,caller)	usub1(E,caller)	- {sub2(E,caller)}
3	sub3(G,caller)	usub1(E,caller)	∪ {usub1(E,caller)}
4	sub4(caller)	usub1(E,caller)	∪ {usub1(E,caller)}
5	sub2(E,caller) + usub2(E,G,caller)	usub1(E,caller)	- {sub2(E,caller), usub2(E,G,caller)}
6	sub3(G,caller) + usub2(E,G,caller)	usub1(E,caller)	∪ {usub1(E,caller)} -{usub2(E,G,caller)}
7	sub4(caller) + usub2(E,G,caller)	usub1(E,caller)	∪ {usub1(E,caller)} -{usub2(E,G,caller)}
8	sub2(E,caller) + usub3(G,caller)	usub1(E,caller)	-{sub2(caller), usub3(G,caller)}
9	sub2(E2,caller) + sub2(E,caller) + usub3(G,caller)	usub1(E,caller)	-{sub2(caller)}
10	sub4(caller) + usub3(G,caller)	usub1(E,caller)	∪ {usub1(E,caller)}
1	sub1(E,G,caller)	usub2(E,G,caller)	-{sub1(E,G,caller)}
2	sub2(E,caller)	usub2(E,G,caller)	∪ {usub1(E,G,caller)}
3	sub3(G,caller)	usub2(E,G,caller)	∪ {usub1(E,G,caller)}
4	sub4(caller)	usub2(E,G,caller)	∪ {usub1(E,G,caller)}
5	sub3(G,caller) + usub1(E,caller)	usub2(E,G,caller)	SKIP
6	sub4(caller) + usub1(E,caller)	usub2(E,G,caller)	SKIP
7	sub2(E,caller) + usub3(G,caller)	usub2(E,G,caller)	SKIP
8	sub4(caller) + usub3(G,caller)	usub2(E,G,caller)	SKIP
1	sub1(E,G,caller)	usub3(G,caller)	-{sub1(E,G,caller)}
2	sub2(E,caller)	usub3(G,caller)	∪ {usub3(G,caller)}
3	sub3(G,caller)	usub3(G,caller)	-{sub3(G,caller)}
4	sub4(caller)	usub3(G,caller)	∪ {usub3(G,caller)}
5	sub3(G,caller) + usub1(E,caller)	usub3(G,caller)	-{sub3(G,caller),usub1(E,caller)}
6	sub3(G2,caller) + sub3(G,caller) + usub1(E,caller)	usub3(G,caller)	-{sub3(G,caller)}
7	sub4(caller) + usub1(E,caller)	usub3(G,caller)	∪ {usub3(G,caller)}
8	sub2(E,caller) + usub2(E,G,caller)	usub3(G,caller)	∪ {usub3(G,caller)} -{usub2(E,G,caller)}
9	sub3(G,caller) + usub2(E,G,caller)	usub3(G,caller)	- {sub3(G,caller), usub2(E,G,caller)}
10	sub4(caller) + usub2(E,G,caller)	usub3(G,caller)	∪ {usub3(G,caller)} -{usub2(E,G,caller)}
1	...	usub4(caller)	-{sub1(*,*,caller), sub2(*,caller), sub3(*,caller), sub4(caller) }

```
unsubscribeThirdType(groupId) =
PRE
    (groupId ∈ GroupId) ∧ ∃(group).( (group ∈ network'groups) ∧
                                     (group'gid = groupId) /* InvalidGroupId */ ∧
                                     (currentPeer ∈ group'members) /* CallerNotInGroup */ ) ∧
    ( /* NotSubscribed */
      ∃(subscription).( (subscription ∈ network'subscriptions'firstType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'thirdType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                        (subscription'subscriber = caller) )
    )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    (networkUpdated'subscriptions'firstType = {subscription |
                (subscription ∈ network'subscriptions'firstType) ∧
                ( (subscription'group ≠ groupId) ∨
                  (subscription'subscriber ≠ currentPeer) ) }) ∧
    (networkUpdated'subscriptions'secondType = network'subscriptions'secondType) ∧
    /* usub3(G,caller) ∧ sub3(G,caller) ⇒ FALSE */
    (networkUpdated'subscriptions'thirdType = {subscription |
                (subscription ∈ network'subscriptions'thirdType) ∧
                ( (subscription'group ≠ groupId) ∨
                  (subscription'subscriber ≠ currentPeer) ) }) ∧
    (networkUpdated'subscriptions'fourthType = network'subscriptions'fourthType) ∧
    /* see line 5 and 6 */
    (networkUpdated'unsubscriptions'firstType = {unsubscription |
                (unsubscription ∈ network'unsubscriptions'firstType) ∧
                ∃(subscription).((subscription ∈ network'subscriptions'thirdType) ∧
                                 (subscription'subscriber = currentPeer) ∧
                                 (subscription'group = groupId) ) }) ∧
    ( /* second unsubscription type */
      ( /* see line 2, 4, 7, 8, 10 */
        (
          ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                            (subscription'subscriber = currentPeer) ) ∨
          ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                            (subscription'subscriber = currentPeer) )
        ) ∧
        networkUpdated'unsubscriptions'thirdType = network'unsubscriptions'thirdType
                                                   ∪ {rec(groupId,currentPeer)}
      ) ∨
      /* see line 1, 3, 5, 6, 9 */
      networkUpdated'unsubscriptions'thirdType = network'unsubscriptions'thirdType
    ) ∧
    /* usub3(G,caller) ∧ usub2(E,G,caller) ⇒ FALSE */
    (networkUpdated'unsubscriptions'secondType = {unsubscription |
                (unsubscription ∈ network'unsubscriptions'secondType) ∧
                ( (unsubscription'unsubscriber ≠ currentPeer) ∨
                  (unsubscription'group ≠ groupId) )}) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧ (networkUpdated'replies = network'replies)
    (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧
  THEN network ≡ networkUpdated
  END
END
```

Figure B.28: unsubscribe - Third Type

```
unsubscribeSecondType(eventName, groupId) =
PRE
    (eventName ∈ EventName) ∧ (groupId ∈ GroupId) ∧
    ∃(group).( (group ∈ network'groups) ∧ (group'gid = groupId) /* InvalidGroupId */ ∧
               (currentPeer ∈ group'members) /* CallerNotInGroup */ ) ∧
    /* NotSubscribed */
    (
      ∃(subscription).( (subscription ∈ network'subscriptions'firstType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'thirdType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                        (subscription'subscriber = caller) )
    )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    /* usub2(E,G,caller) ∧ sub(E,G,caller) ⇒ FALSE */
    (networkUpdated'subscriptions'firstType = {subscription |
                    (subscription ∈ network'subscriptions'firstType) ∧
                    ( (subscription'event ≠ eventName) ∨ (subscription'group ≠ groupId) ∨
                      (subscription'subscriber ≠ currentPeer) ) }) ∧
    (networkUpdated'subscriptions'secondType = network'subscriptions'secondType) ∧
    (networkUpdated'subscriptions'thirdType = network'subscriptions'thirdType) ∧
    (networkUpdated'subscriptions'fourthType = network'subscriptions'fourthType) ∧
    (networkUpdated'unsubscriptions'firstType = network'unsubscriptions'firstType) ∧
    ( /* second unsubscription type */
      (
        (
          ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                            (subscription'subscriber = currentPeer) ∧
                            (subscription'event = eventName) ) ∨
          ∃(subscription).( (subscription ∈ network'subscriptions'thirdType) ∧
                            (subscription'subscriber = currentPeer) ∧
                            (subscription'group = groupId) ) ∨
          ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                            (subscription'subscriber = currentPeer) )
        ) ∧ (
          ∄(unsubscription).( (unsubscription ∈ network'unsubscriptions'firstType) ∧
                              (unsubscription'unsubscriber = currentPeer) ∧
                              (unsubscription'event = eventName) ) ∧
          ∄(unsubscription).( (unsubscription ∈ network'unsubscriptions'secondType) ∧
                              (unsubscription'unsubscriber = currentPeer) ∧
                              (unsubscription'group = groupId) )
        ) ∧
        (networkUpdated'unsubscriptions'secondType = network'unsubscriptions'secondType ∪
                                                {rec(eventName,groupId,currentPeer)} )
      ) ∨
      networkUpdated'unsubscriptions'secondType = network'unsubscriptions'secondType
    ) ∧
    (networkUpdated'unsubscriptions'thirdType = network'unsubscriptions'thirdType) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧
    (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧
    (networkUpdated'replies = network'replies)
  THEN network ≡ networkUpdated
  END
END
```

Figure B.29: unsubscribe - Second Type

```
unsubscribeFirstType(caller, eventName) =
PRE
    (caller ∈ (PeerId ∪ PeerServiceId)) ∧ (eventName ∈ EventName) ∧
    ( /* Used because caller has been added */
      ∃(peer,service).( (peer ∈ network'peers) ∧ (peer'pid = currentPeer) ∧
                        (service ∈ peer'services) ∧ (service'psid = caller) ) ∨
      (caller = currentPeer)
    ) ∧
    ( /* NotSubscribed */
      ∃(subscription).( (subscription ∈ network'subscriptions'firstType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'thirdType) ∧
                        (subscription'subscriber = caller) ) ∨
      ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                        (subscription'subscriber = caller) )
    )
THEN
  ANY networkUpdated
  WHERE
    (networkUpdated ∈ NetworkState) ∧
    (networkUpdated'subscriptions'firstType = {subscription |
                (subscription ∈ network'subscriptions'firstType) ∧
                ( (subscription'event ≠ eventName) ∨
                  (subscription'subscriber ≠ caller) ) }) ∧
    /* usub1(E,caller) ∧ sub2(E,caller) ⇒ FALSE */
    (networkUpdated'subscriptions'secondType = {subscription |
                (subscription ∈ network'subscriptions'secondType) ∧
                ( (subscription'event ≠ eventName) ∨
                  (subscription'subscriber ≠ caller) ) }) ∧
    (networkUpdated'subscriptions'thirdType = network'subscriptions'thirdType) ∧
    (networkUpdated'subscriptions'fourthType = network'subscriptions'fourthType) ∧
    ( /* first unsubscription type */
      ( /* see line 3, 4, 6, 7, 10 */
        ( ∃(subscription).( (subscription ∈ network'subscriptions'thirdType) ∧
                            (subscription'subscriber = caller) ) ∨
          ∃(subscription).( (subscription ∈ network'subscriptions'fourthType) ∧
                            (subscription'subscriber = caller) ) ) ∧
        networkUpdated'unsubscriptions'firstType = network'unsubscriptions'firstType
                                                  ∪ {rec(eventName,caller)}
      ) ∨
      /* see line 1, 2, 5, 8, 9 */
      networkUpdated'unsubscriptions'firstType = network'unsubscriptions'firstType
    ) ∧
    /* usub1(E,caller) ∧ usub2(E,G,caller) ⇒ FALSE */
    (networkUpdated'unsubscriptions'secondType = {unsubscription |
                (unsubscription ∈ network'unsubscriptions'secondType) ∧
                ( (unsubscription'unsubscriber ≠ caller) ∨
                  (unsubscription'event ≠ eventName) )}) ∧
    /* see line 8 and 9 */
    (networkUpdated'unsubscriptions'thirdType = {unsubscription |
                (unsubscription ∈ network'unsubscriptions'thirdType) ∧
                ∃(subscription).( (subscription ∈ network'subscriptions'secondType) ∧
                                  (subscription'subscriber = caller) ∧
                                  (subscription'event = eventName) ) }) ∧
    /* other network fields are equal */
    (networkUpdated'cred = network'cred) ∧ (networkUpdated'peers = network'peers) ∧
    (networkUpdated'groups = network'groups) ∧ (networkUpdated'replies = network'replies)
    (networkUpdated'sentEvents = network'sentEvents) ∧
    (networkUpdated'invocations = network'invocations) ∧
  THEN network ≡ networkUpdated END
END
```

Figure B.30: unsubscribe

Appendix C

SequiTel Examples

C.1 Pseudo-SMoL Codes

C.1.1 Equipment

Equipment Peer

```
Sequence
         myCredentials = opaque
         newPeer(myCredentials)

         securityInfoSequiTel = opaque
         groupSequiTelDescription = <"SequiTel-Operators",groupSequiTelDescription>
         groupsSequiTel[] = getGroups(groupSequiTelDescription, myCredentials)

         securityInfoFriends = opaque
         groupFriendsDescription = <"SequiTel-JohnSMITH",securityInfoFriends>
         groupsFriends[] = getGroups(groupFriendsDescription, myCredentials)

         If groupsSequiTel[0] = null
             groupsSequiTel[0] = createGroup(groupSequiTelDescription, myCredentials)
         Else
             joinGroup(groupsSequiTel[0], myCredentials)
         End If

         If groupsFriends[0] = null
             groupsFriends[0] = createGroup(groupFriendDescription, myCredentials)
         Else
             joinGroup(groupsFriends[0], myCredentials)
         End If

         monitoringServiceContract = opaque
         monitoringServiceGrounding = opaque

         <serviceSequiTelGroup.gsid,serviceSequiTelGroup.psid> =
                                   publish(groupSequiTel, monitoringServiceContract,
                                           monitoringServiceGrounding)
         <serviceFriendsGroup.gsid,serviceFriendsGroup.psid> =
                                    publish(groupFriends, monitoringServiceContract,
                                            monitoringServiceGrounding)

         wait("P1Y") End Sequence
```

Figure C.1: Equipment Peer - Pseudo-SMoL

Equipment Monitoring

```
Flow
     InformationHandler
                         Sequence
                                  wait("P1Y")
                                  throw("applicationTermination")
                         End Sequence

                         <caller> = receiveMessage("get_temperature")
                                   Sequence
                                            temperature = opaque
                                            reply(caller, temperature)
                                   End Sequence

                         <caller> = receiveMessage("get_blood_pressure")
                                   Sequence
                                            bloodPressure = opaque
                                            reply(caller, bloodPressure)
                                   End Sequence
     End InformationHandler

     While true
           Sequence
                    temperature = opaque
                    bloodPressure = opaque

                    If (temperature < 37) or (bloodPressure > 38)
                       event("temperatureMonitoring", temperature)
                    Else
                       empty
                    End If

                    If (bloodPressure < 9) or (bloodPressure > 12)
                       event("bloodPressureMonitoring", bloodPressure)
                    Else
                       empty
                    End If

                    wait("P0DT0H0M30S")
           End Sequence
     End While
End Flow
```

Figure C.2: Equipment Monitoring - Pseudo-SMoL

C.1.2 Operator

Operator Peer

```
Sequence
        myCredentials = opaque
        newPeer(myCredentials)

        securityInfoSequiTel = opaque
        groupSequiTelDescription = <"SequiTel-Operators",groupSequiTelDescription>
        groupsSequiTel[] = getGroups(groupSequiTelDescription, myCredentials)

        If groupsSequiTel[0] = null
            groupsSequiTel[0] = createGroup(groupSequiTelDescription, myCredentials)
        Else
            joinGroup(groupsSequiTel[0], myCredentials)
        End If

        InformationHandler
                  wait("P1Y")

                  <caller,value> = receiveEvent ("temperatureMonitoring")
                          If peopleInTrouble.contains(caller)
                             empty
                          Else
                             RepeatUntil (bloodPressure <9) || (bloodPressure >12) ||
                                         (temperature <37) || (temperature >38)
                                   bloodPressure = invoke(caller, "get_blood_pressure", null, true)
                                   temperature = invoke(caller, "get_temperature", null, true)
                             End RepeatUntil
                             Empty /* remove caller from people in trouble */
                          End If

                  <caller,value> = receiveEvent ("bloodPressureMonitoring")
                          If peopleInTrouble.contains(caller)
                             empty
                          Else
                             RepeatUntil (bloodPressure <9) || (bloodPressure >12) ||
                                         (temperature <37) || (temperature >38)
                                   bloodPressure = invoke(caller, "get_blood_pressure", null, true)
                                   temperature = invoke(caller, "get_temperature", null, true)
                             End RepeatUntil
                             Empty /* remove caller from people in trouble */
                          End If
        End InformationHandler
End Sequence
```

Figure C.3: Operator Peer - Pseudo-SMoL

C.2 SMoL codes

C.2.1 Equipment

Equipment peer

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Process xmlns="http://www.smepp.org/schema/smol4peersdocument" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://www.smepp.org/schema/smol4peersdocument">
4    <variables>
5      <variable name="myCredentials">  <init opaque="" /> </variable>
6
7      <variable name="securityInfoSequiTel"> <init opaque="" /> </variable>
8      <variable name="groupSequiTelDescription">  <init>
9          <literal>new GroupDescription("SequiTel-Operators", securityInfoSequiTel,
10                  "SequiTel-Operators␣group" )</literal> </init>
11     </variable>
12     <variable name="groupsSequiTel" />
13     <variable name="groupSequiTel" />
14     <variable name="serviceSequiTelGroup" />
15
16     <variable name="securityInfoFriends"> <init opaque="" /> </variable>
17     <variable name="groupFriendsDescription"> <init>
18         <literal>new GroupDescription("SequiTel-JohnSMITH", securityInfoSequiTel,
19                  "SequiTel-Friends␣group" )</literal> </init>
20     </variable>
21     <variable name="groupsFriends" />
22     <variable name="groupFriends" />
23     <variable name="serviceFriendsGroup" />
24
25     <variable name="monitoringServiceContract"> <init opaque="" /> </variable>
26     <variable name="monitoringServiceGrounding"> <init opaque="" /> </variable>
27   </variables>
```

Figure C.4: Equipment Peer - SMoL - Part 1

```
29  <Sequence>
30    <NewPeer>
31      <credentials variable="myCredentials" />
32    </NewPeer>
33
34    <GetGroups>
35      <result variable="groupsSequiTel" />
36      <groupDescr variable="groupSequiTelDescription" />
37      <credentials variable="myCredentials" />
38    </GetGroups>
39    <IfElse>
40      <guard> <literal>(groupsSequiTel == null) || (groupsSequiTel.length == 0)</literal> </guard>
41      <activity>
42        <CreateGroup>
43          <result variable="groupSequiTel" />
44          <groupDescr variable="groupSequiTelDescription" />
45          <credentials variable="myCredentials" />
46        </CreateGroup>
47      </activity>
48      <elseActivity>
49        <Sequence>
50          <Assign>
51            <copy>
52              <from><literal>groupsSequiTel[0]</literal></from>
53              <to variable="groupSequiTel" />
54            </copy>
55          </Assign>
56          <JoinGroup> <groupId variable="groupSequiTel" /> </JoinGroup>
57        </Sequence>
58      </elseActivity>
59    </IfElse>
60
61    <GetGroups>
62      <result variable="groupsFriends" />
63      <groupDescr variable="groupFriendsDescription" />
64      <credentials variable="myCredentials" />
65    </GetGroups>
66    <IfElse>
67      <guard> <literal>(groupsFriends == null) || (groupsFriends.length == 0)</literal> </guard>
68      <activity>
```

Figure C.5: Equipment Peer - SMoL - Part 2

```
69          <CreateGroup>
70            <result variable="groupFriends" />
71            <groupDescr variable="groupFriendsDescription" />
72            <credentials variable="myCredentials" />
73          </CreateGroup>
74        </activity>
75        <elseActivity>
76          <Sequence>
77            <Assign>
78              <copy>
79                <from><literal>groupsFriends[0]</literal></from>
80                <to variable="groupFriends" />
81              </copy>
82            </Assign>
83            <JoinGroup> <groupId variable="groupFriends" /> </JoinGroup>
84          </Sequence>
85        </elseActivity>
86      </IfElse>
87
88      <Publish>
89        <result variable="serviceSequiTelGroup"/>
90        <groupId variable="groupSequiTel" />
91        <contract variable="monitoringServiceContract" />
92        <grounding variable="monitoringServiceGrounding" />
93      </Publish>
94
95      <Publish>
96        <result variable="serviceFriendsGroup"/>
97        <groupId variable="groupFriends" />
98        <contract variable="monitoringServiceContract" />
99        <grounding variable="monitoringServiceGrounding" />
100     </Publish>
101
102     <Wait>
103       <for> <literal>"P1Y"</literal> </for>
104     </Wait>
105   </Sequence>
106 </Process>
```

Figure C.6: Equipment Peer - SMoL - Part 3

Equipment Monitoring

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Contract xmlns:sign="http://www.w3.org/ns/wsdl" xmlns:smol="http://www.smepp.org/schema/SMoL"
3      xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:ha="smepp://homeAutomation.owl" xsi:noNamespaceSchemaLocation="Contract.xsd"
5      xmlns:qos="smepp://qos.owl">
6      <Profile name="EquipmentMonitoring" category="cat1"/>
7      <sign:Signature>
8        <sign:types language="java">
9          <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
10           <xsd:complexType name="outputMsg">
11             <xsd:attribute name="result" type="java.lang.Float"></xsd:attribute>
12           </xsd:complexType>
13         </xsd:schema>
14       </sign:types>
15
16       <sign:interface>
17         <sign:operation name="get_temperature" pattern="request-response">
18           <sign:output element="outputMsg" messageLabel="result"></sign:output>
19         </sign:operation>
20
21         <sign:operation name="get_blood_pressure" pattern="request-response">
22           <sign:output element="outputMsg" messageLabel="result"></sign:output>
23         </sign:operation>
24
25           <sign:event name="temperatureMonitoring" />
26         <sign:event name="bloodPressureMonitoring" />
27       </sign:interface>
28     </sign:Signature>
29
30     <Behavior type="state-less" maxInstances="1">
31     <Process name="EquipmentMonitoring" targetNamespace="http://myAuction.org">
```

Figure C.7: Equipment Monitoring - SMoL - Part 1

```
33      <Flow>
34        <InfoHandler>
35          <Sequence>
36            <Wait> <for> <literal>"P1Y"</literal> </for> </Wait>
37            <Throw>
38              <faultName>applicationTermination</faultName>
39            </Throw>
40          </Sequence>
41
42          <RcvMsgHandler>
43            <variables>
44              <variable name="rcvMsg" />
45              <variable name="temperature" />
46            </variables>
47            <RcvMsg>
48              <result variable="rcvMsg" />
49              <operationName>
50                <literal>"get_temperature"</literal>
51              </operationName>
52              <inputType> <literal>new Class[]{}</literal> </inputType>
53            </RcvMsg>
54
55            <Sequence>
56              <Assign>
57                <copy>
58                  <from opaque="" />
59                  <to variable="temperature" />
60                </copy>
61              </Assign>
62
63              <Reply>
64                <callerId variable="rcvMsg" part="caller" />
65                <operationName>
66                  <literal>"get_temperature"</literal>
67                </operationName>
68                <output variable="temperature" />
69              </Reply>
70            </Sequence>
71          </RcvMsgHandler>
```

Figure C.8: Equipment Monitoring - SMoL - Part 2

```
73        <RcvMsgHandler>
74          <variables>
75            <variable name="rcvMsg" />
76            <variable name="bloodPressure" />
77          </variables>
78          <RcvMsg>
79            <result variable="rcvMsg" />
80            <operationName>
81              <literal>"get_blood_pressure"</literal>
82            </operationName>
83            <inputType> <literal>new Class[]{}</literal> </inputType>
84          </RcvMsg>
85
86          <Sequence>
87            <Assign>
88              <copy>
89                <from opaque="" />
90                <to variable="bloodPressure" />
91              </copy>
92            </Assign>
93
94            <Reply>
95              <callerId variable="rcvMsg" part="caller" />
96              <operationName>
97                <literal>"get_blood_pressure"</literal>
98              </operationName>
99              <output variable="bloodPressure" />
100           </Reply>
101         </Sequence>
102       </RcvMsgHandler>
103     </InfoHandler>
```

Figure C.9: Equipment Monitoring - SMoL - Part 3

```
105        <variables>
106          <variable name="temperature"><init opaque="" /></variable>
107          <variable name="bloodPressure"><init opaque="" /><variable>
108        </variables>
109        <While>
110          <guard> <literal>true</literal> </guard>
111          <activity>
112            <Sequence>
113              <IfElse>
114                <guard> <literal>(temperature &lt; 37) ||
115                (temperature &gt; 38)</literal> </guard>
116                          <activity>
117                            <Event>
118                              <eventName><literal>
119                                "temperatureMonitoring"
120                              </literal></eventName>
121                              <input variable="temperature" />
122                            </Event>
123                </activity>
124                        </IfElse>
125                    <IfElse>
126                      <guard> <literal>(bloodPressure &lt; 9) ||
127                (bloodPressure &gt; 12)</literal> </guard>
128                <activity>
129                        <Event>
130                    <eventName><literal>
131                      "bloodPressureMonitoring"
132                    </literal> </eventName>
133                            <input variable="bloodPressure" />
134                            </Event>
135                      </activity>
136                    </IfElse>
137              <Wait> <for><literal>"P0DT0H0M30S"</literal></for> </Wait>
138            </Sequence>
139          </activity>
140        </While>
141      </Flow>
142    </Process>
143    </Behavior>
144 </Contract>
```

Figure C.10: Equipment Monitoring - SMoL - Part 4

C.2.2 Operator

Operator peer

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Process xmlns="http://www.smepp.org/schema/smol4peersdocument" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://www.smepp.org/schema/smol4peersdocument">
4    <variables>
5      <variable name="myCredentials">  <init opaque="" /> </variable>
6
7      <variable name="securityInfoSequiTel"> <init opaque="" /> </variable>
8      <variable name="groupSequiTelDescription">
9        <init>
10         <literal>new GroupDescription("SequiTel-Operators", securityInfoSequiTel,
11                "SequiTel-Operators␣group" )</literal>
12       </init>
13     </variable>
14     <variable name="groupsSequiTel" />
15     <variable name="groupSequiTel" />
16
17     <variable name="peopleInTrouble" />
18   </variables>
19
20   <Sequence>
21     <NewPeer> <credentials variable="myCredentials" /> </NewPeer>
22
23     <GetGroups>
24       <result variable="groupsSequiTel" />
25       <groupDescr variable="groupSequiTelDescription" />
26       <credentials variable="myCredentials" />
27     </GetGroups>
28     <IfElse>
29       <guard> <literal>(groupsSequiTel == null) || (groupsSequiTel.length == 0)</literal> </guard>
30       <activity>
31         <CreateGroup>
32           <result variable="groupSequiTel" />
33           <groupDescr variable="groupSequiTelDescription" />
34           <credentials variable="myCredentials" />
35         </CreateGroup>
36       </activity>
```

Figure C.11: Operator Peer - SMoL - Part 1

```
37      <elseActivity>
38        <Sequence>
39          <Assign>
40            <copy>
41              <from><literal>groupsSequiTel[0]</literal></from>
42              <to variable="groupSequiTel" />
43            </copy>
44          </Assign>
45          <JoinGroup> <groupId variable="groupSequiTel" /> </JoinGroup>
46        </Sequence>
47      </elseActivity>
48    </IfElse>
49
50    <Subscribe>
51      <eventName> <literal>"temperatureMonitoring"</literal> </eventName>
52    </Subscribe>
53    <Subscribe>
54      <eventName> <literal>"bloodPressureMonitoring"</literal> </eventName>
55    </Subscribe>
56
57    <InfoHandler>
58      <Wait> <for> <literal>"P1Y"</literal> </for> </Wait>
59
60      <RcvResHandler>
61        <variables>
62          <variable name="rcvEvt" />
63          <variable name="bloodPressure" />
64          <variable name="temperature" />
65        </variables>
66        <RcvEvt>
67          <result variable="rcvEvt" />
68          <eventName> <literal>"temperatureMonitoring"</literal> </eventName>
69        </RcvEvt>
```

Figure C.12: Operator Peer - SMoL - Part 2

```
71      <IfElse>
72        <guard> <literal>peopleInTrouble.contains(rcvEvt.caller)</literal> </guard>
73        <activity> <Empty /> <!-- ignore --> </activity>
74        <elseActivity>
75          <RepeatUntil>
76            <guard><literal>(bloodPressure &lt; 9) || (bloodPressure &gt; 12)
77            || (temperature &lt; 37) || (temperature &gt; 38)</literal></guard>
78
79            <activity>
80              <Sequence>
81                <Invoke>
82                  <result variable="temperature" />
83                  <targetId variable="rcvEvt"part="caller"/>
84                  <opName>
85                  <literal>"get_temperature"</literal>
86                  </opName>
87                  <input>
88                  <literal>new Serializable[]{}</literal>
89                  </input>
90                  <returnResult>true</returnResult>
91                </Invoke>
92                <Invoke>
93                  <result variable="bloodPressure" />
94                  <targetId variable="rcvEvt"part="caller"/>
95                  <opName>
96                  <literal>"get_blood_pressure"</literal>
97                  </opName>
98                  <input>
99                  <literal>new Serializable[]{}</literal>
100                 </input>
101                 <returnResult>true</returnResult>
102               </Invoke>
103             </Sequence>
104           </activity>
105         </RepeatUntil>
106       </elseActivity>
107     </IfElse>
108   </RcvResHandler>
```

Figure C.13: Operator Peer - SMoL - Part 3

```
110 <RcvResHandler>
111   <variables>
112     <variable name="rcvEvt" />
113     <variable name="temperature" />
114     <variable name="bloodPressure" />
115   </variables>
116   <RcvEvt>
117     <result variable="rcvEvt" />
118     <eventName> <literal>"bloodPressureMonitoring"</literal> </eventName>
119   </RcvEvt>
120
121   <IfElse>
122     <guard> <literal>peopleInTrouble.contains(rcvEvt.caller)</literal> </guard>
123     <activity> <Empty /> <!-- ignore --> </activity>
124     <elseActivity>
125       <RepeatUntil>
126         <guard><literal>(bloodPressure &lt; 9) || (bloodPressure &gt; 12)
127         || (temperature &lt; 37) || (temperature &gt; 38)</literal></guard>
128         <activity>
129           <Sequence>
130             <Invoke>
131               <result variable="temperature" />
132               <targetId variable="rcvEvt"part="caller"/>
133               <opName>
134               <literal>"get_temperature"</literal>
135               </opName>
136               <input>
137               <literal>new Serializable[]{}</literal>
138               </input>
139               <returnResult>true</returnResult>
140             </Invoke>
```

Figure C.14: Operator Peer - SMoL - Part 4

```
141                     <Invoke>
142                       <result variable="bloodPressure" />
143                       <targetId variable="rcvEvt"part="caller"/>
144                       <opName>
145                       <literal>"get_blood_pressure"</literal>
146                       </opName>
147                       <input>
148                       <literal>new Serializable[]{}</literal>
149                       </input>
150                       <returnResult>true</returnResult>
151                     </Invoke>
152                   </Sequence>
153                 </activity>
154               </RepeatUntil>
155             </elseActivity>
156           </IfElse>
157         </RcvResHandler>
158       </InfoHandler>
159     </Sequence>
160   </Process>
```

Figure C.15: Operator Peer - SMoL - Part 5

C.3 Translated Codes

C.3.1 Equipment

Equipment Peer

```
1  package com.SequiTel.equipment;
2
3  import org.smepp.extendFramework.ISMEPPPeerClass;
4  import org.smepp.api.PeerManager;
5  import org.smepp.extendFramework.SMEPPProcessManager;
6  import org.smepp.api.exceptions.SMEPPException;
7  import org.smepp.extendFramework.IThreadedCommand;
8  import org.smepp.extendFramework.LocalVariablesDeclaration;
9  import org.smepp.extendFramework.SMoL;
10 import org.smepp.datatypes.gm.GroupDescription;
11 import org.smepp.datatypes.smm.PeerServiceId;
12 import org.smepp.datatypes.api.Credentials;
13 import org.smepp.datatypes.api.ServiceGrounding;
14 import org.smepp.datatypes.gm.GroupId;
15 import org.smepp.smm.xml.ContractType;
16
17 /**
18   @author SMoLTranslator
19 **/
20 public class EquipmentPeer implements ISMEPPPeerClass
21 {
22   private PeerManager peerManager;
23   private SMEPPProcessManager processManager = null;
24   private GroupDescription groupSequiTelDescription =
25     new GroupDescription("SequiTel-Operators",
26         securityInfoSequiTel, "SequiTel-Operators␣group" );
27   private Object securityInfoSequiTel = null /* opaque */;
28   private PeerServiceId serviceSequiTelGroup;
29   private Credentials myCredentials = null /* opaque */;
30   private ServiceGrounding monitoringServiceGrounding = null /* opaque */;
31   private GroupId groupSequiTel;
32   private GroupId[] groupsSequiTel;
33   private GroupDescription groupFriendsDescription =
34     new GroupDescription("SequiTel-JohnSMITH",
35         securityInfoFriends, "SequiTel-JohnSMITH␣group" );
36   private GroupId[] groupsFriends;
37   private PeerServiceId serviceFriendsGroup;
38   private ContractType monitoringServiceContract = null /* opaque */;
39   private Object securityInfoFriends = null /* opaque */;
40   private GroupId groupFriends;
41
42   /**
43     The translated SMoL behavior
44     @throws SMEPPException
45   **/
46   public static void main(String[] args) throws Exception
47   {
48     new EquipmentPeer();
49   }
```

Figure C.16: Equipment Peer- Translated - Part 1

```
51    /**
52      The translated SMoL behavior
53      @throws SMEPPException
54    **/
55    public EquipmentPeer() throws SMEPPException
56    {
57      this.processManager = new SMEPPProcessManager();
58
59      peerManager = PeerManager.newPeer(myCredentials, null);
60      if ( mustBeStopped() ) return;
61
62      groupsSequiTel = peerManager.getGroups(groupSequiTelDescription,
63                myCredentials);
64      if ( mustBeStopped() ) return;
65
66      if((groupsSequiTel == null) || (groupsSequiTel.length == 0))
67      {
68        groupSequiTel = peerManager.createGroup(groupSequiTelDescription,
69                  myCredentials);
70      }
71      else
72      {
73        groupSequiTel = groupsSequiTel[0];
74        if ( mustBeStopped() ) return;
75
76        peerManager.joinGroup(groupSequiTel);
77      }
78      if ( mustBeStopped() ) return;
79
80      groupsFriends = peerManager.getGroups(groupFriendsDescription,
81                myCredentials);
82      if ( mustBeStopped() ) return;
83
84      if((groupsFriends == null) || (groupsFriends.length == 0))
85      {
86        groupFriends = peerManager.createGroup(groupFriendsDescription,
87                  myCredentials);
88      }
89      else
90      {
91        groupFriends = groupsFriends[0];
92        if ( mustBeStopped() ) return;
93
94        peerManager.joinGroup(groupFriends);
95      }
96      if ( mustBeStopped() ) return;
97
98      serviceSequiTelGroup = peerManager.publish(groupSequiTel,
99              monitoringServiceContract,
100             monitoringServiceGrounding,null,null);
101     if ( mustBeStopped() ) return;
102
103     serviceFriendsGroup = peerManager.publish(groupFriends,
104             monitoringServiceContract,
105             monitoringServiceGrounding,null,null);
106     if ( mustBeStopped() ) return;
107
108     SMoL.waitFor( this, "P1Y" );
109   }
```

Figure C.17: Equipment Peer - Translated - Part 2

```
111     /*
112       The following methods are always implemented in the same way
113     */
114     /**
115       @return true if the service must be stopped else false
116     **/
117     public boolean mustBeStopped()
118     {
119       return this.processManager.mustBeStopped();
120     }
121
122     /**
123       Stop the service execution
124     **/
125     public void stop()
126     {
127       this.processManager.stop();
128     }
129
130     /**
131       @see ISMEPPPeerClass
132     **/
133     public void addInnerThreadedCommand(IThreadedCommand cmd)
134     {
135       this.processManager.addInnerThreadedCommand(cmd);
136     }
137
138   }
```

Figure C.18: Equipment Peer - Translated - Part 3

Equipment Monitoring

```
1  package com.SequiTel.equipment.monitoring;
2
3  import org.smepp.extendFramework.ISMEPPServiceClass;
4  import org.smepp.api.ServiceManager;
5  import org.smepp.extendFramework.SMEPPProcessManager;
6  import java.io.Serializable;
7  import org.smepp.api.exceptions.SMEPPException;
8  import org.smepp.extendFramework.IThreadedCommand;
9  import org.smepp.extendFramework.LocalVariablesDeclaration;
10 import org.smepp.extendFramework.SMoLFlowBranch;
11 import org.smepp.extendFramework.SMoLIH;
12 import org.smepp.extendFramework.SMoLIHBranch;
13 import org.smepp.extendFramework.SMoLIHMainCommand;
14 import org.smepp.extendFramework.SMoL;
15 import org.smepp.api.exceptions.CUSTOM.applicationTermination;
16 import org.smepp.extendFramework.SMoLIHRcvMsgBranch;
17 import org.smepp.extendFramework.SMoLFlow;
18 import org.smepp.datatypes.em.EventInfo;
19 import org.smepp.api.misc.ReceivedMessage;
20
21 /**
22   @author SMoLTranslator
23 **/
24 public class EquipmentMonitoring implements ISMEPPServiceClass
25 {
26   private ServiceManager serviceManager;
27   private SMEPPProcessManager processManager = null;
28
29   /**
30     @param svc the API service manager instance
31     @param constrValue values which permit to initialize this object
32   **/
33   public EquipmentMonitoring(ServiceManager svc, Serializable constrValue)
34   {
35     serviceManager = svc;
36   }
37
38   /**
39     The translated SMoL behavior
40     @throws SMEPPException
41   **/
42   public void service() throws SMEPPException
43   {
44     new SMoLFlow( this,
45       new SMoLFlowBranch[]{
46         new SMoLFlowBranch(  )
47         {
48           public void command() throws SMEPPException
49           {
50             new SMoLIH( this,
```

Figure C.19: Equipment Monitoring - Translated - Part 1

```
51                  new SMoLIHMainCommand(  )
52                  {
53                    public void command() throws SMEPPException
54                    {
55                      SMoL.waitFor( this, "P1Y" );
56                      if ( mustBeStopped() ) return;
57
58                      throw new applicationTermination(  );
59                    }
60                  }
61                  , new SMoLIHBranch[]{
62                    new SMoLIHRcvMsgBranch(  )
63                    {
64                      ReceivedMessage rcvMsg;
65                      Serializable[] temperature;
66
67                      protected int getTimeout()
68                      {
69                        return -1;
70                      }
71                      public void receiveMessage(int roundRobinTimeout)
72                                    throws SMEPPException
73                      {
74                        rcvMsg = SMoL.receiveMessage(this, serviceManager,
75                                "get_temperature", new Class[]{Integer.class},
76                                roundRobinTimeout);
77                      }
78                      public void command() throws SMEPPException
79                      {
80                        // TODO opaque value(s)
81                        temperature = null /* opaque */;
82                        if ( mustBeStopped() ) return;
83
84                        serviceManager.reply(rcvMsg.callerId,
85                                "get_temperature", temperature, null);
86                      }
87                    }
88                    ,
89                    new SMoLIHRcvMsgBranch(  )
90                    {
91                      ReceivedMessage rcvMsg;
92                      Serializable[] bloodPressure;
93
94                      protected int getTimeout()
95                      {
96                        return -1;
97                      }
98                      public void receiveMessage(int roundRobinTimeout)
99                                    throws SMEPPException
100                     {
101                       rcvMsg = SMoL.receiveMessage(this, serviceManager,
102                               "get_blood_pressure", new Class[]{Integer.class},
103                               roundRobinTimeout);
104                     }
105                     public void command() throws SMEPPException
106                     {
107                       // TODO opaque value(s)
108                       bloodPressure = null /* opaque */;
109                       if ( mustBeStopped() ) return;
110
111                       serviceManager.reply(rcvMsg.callerId,
112                               "get_blood_pressure", bloodPressure, null);
113                     }
```

Figure C.20: Equipment Monitoring - Translated - Part 2

```
114                   }
115                 }
116               );
117             }
118           }
119           ,
120           new SMoLFlowBranch(  )
121           {
122             EventInfo bloodPressure = null /* opaque */;
123             EventInfo temperature = null /* opaque */;
124 
125             public void command() throws SMEPPException
126             {
127               // TODO opaque value(s)
128               while(true)
129               {
130                 if ( mustBeStopped() ) return;
131 
132                 if((temperature < 37) || (temperature > 38))
133                 {
134                   serviceManager.event("temperatureMonitoring", temperature);
135                 }
136                 if ( mustBeStopped() ) return;
137 
138                 if((bloodPressure < 9) || (bloodPressure > 12))
139                 {
140                   serviceManager.event("bloodPressureMonitoring", bloodPressure);
141                 }
142                 if ( mustBeStopped() ) return;
143 
144                 SMoL.waitFor( this, "P0DT0H0M30S" );
145               }
146             }
147           }
148         }
149     );
150   }
151 
152   /*
153     The following methods are always implemented in the same way
154   */
155   /**
156     @return true if the service must be stopped else false
157   **/
158   public boolean mustBeStopped()
159   {
160     return this.processManager.mustBeStopped();
161   }
162 
163   /**
164     Stop the service execution
165   **/
166   public void stop()
167   {
168     this.processManager.stop();
169   }
```

Figure C.21: Equipment Monitoring - Translated - Part 3

```
171    /**
172      @see Runnable#run()
173    **/
174    public void run()
175    {
176      try
177      {
178        this.processManager = new SMEPPProcessManager();
179        this.service();
180      }
181      catch ( SMEPPException e )
182      {
183        e.printStackTrace();
184      }
185    }
186
187    /**
188      @see ISMEPPServiceClass
189    **/
190    public void addInnerThreadedCommand(IThreadedCommand cmd)
191    {
192      this.processManager.addInnerThreadedCommand(cmd);
193    }
194
195  }
```

Figure C.22: Equipment Monitoring - Translated - Part 4

C.3.2 Operator

Operator Peer

```
1  package com.SequiTel.operator;
2
3  import org.smepp.extendFramework.ISMEPPPeerClass;
4  import org.smepp.api.PeerManager;
5  import org.smepp.extendFramework.SMEPPProcessManager;
6  import org.smepp.api.exceptions.SMEPPException;
7  import org.smepp.extendFramework.IThreadedCommand;
8  import org.smepp.extendFramework.LocalVariablesDeclaration;
9  import org.smepp.extendFramework.SMoLIH;
10 import org.smepp.extendFramework.SMoLIHBranch;
11 import org.smepp.extendFramework.SMoLIHMainCommand;
12 import org.smepp.extendFramework.SMoL;
13 import org.smepp.extendFramework.SMoLIHRcvEvtBranch;
14 import org.smepp.datatypes.gm.GroupId;
15 import org.smepp.datatypes.gm.GroupDescription;
16 import org.smepp.datatypes.api.Credentials;
17 import java.io.Serializable;
18 import org.smepp.api.misc.ReceivedEvent;
19
20 /**
21   @author SMoLTranslator
22 **/
23 public class OperatorPeer implements ISMEPPPeerClass
24 {
25   private PeerManager peerManager;
26   private SMEPPProcessManager processManager = null;
27   private GroupId groupSequiTel;
28   private GroupId[] groupsSequiTel;
29   private Object peopleInTrouble;
30   private GroupDescription groupSequiTelDescription =
31                  new GroupDescription("SequiTel-Operators",
32                  securityInfoSequiTel, "SequiTel-Operators␣group" );
33   private Object securityInfoSequiTel = null /* opaque */;
34   private Credentials myCredentials = null /* opaque */;
35
36   /**
37     The translated SMoL behavior
38     @throws SMEPPException
39   **/
40   public static void main(String[] args) throws Exception
41   {
42     new OperatorPeer();
43   }
44
45   /**
46     The translated SMoL behavior
47     @throws SMEPPException
48   **/
49   public OperatorPeer() throws SMEPPException
50   {
51     this.processManager = new SMEPPProcessManager();
```

Figure C.23: Operator Peer - Translated - Part 1

```
53      peerManager = PeerManager.newPeer(myCredentials, null);
54      if ( mustBeStopped() ) return;
55
56      groupsSequiTel = peerManager.getGroups(groupSequiTelDescription,
57                      myCredentials);
58      if ( mustBeStopped() ) return;
59
60      if((groupsSequiTel == null) || (groupsSequiTel.length == 0))
61      {
62        groupSequiTel = peerManager.createGroup(groupSequiTelDescription,
63                      myCredentials);
64      }
65      else
66      {
67        groupSequiTel = groupsSequiTel[0];
68        if ( mustBeStopped() ) return;
69
70        peerManager.joinGroup(groupSequiTel);
71      }
72      if ( mustBeStopped() ) return;
73
74      peerManager.subscribe("temperatureMonitoring");
75      if ( mustBeStopped() ) return;
76
77      peerManager.subscribe("bloodPressureMonitoring");
78      if ( mustBeStopped() ) return;
79
80      new SMoLIH( this,
81        new SMoLIHMainCommand(  )
82        {
83          public void command() throws SMEPPException
84          {
85            SMoL.waitFor( this, "P1Y" );
86          }
87        }
88        , new SMoLIHBranch[]{
89          new SMoLIHRcvEvtBranch(  )
90          {
91            Serializable bloodPressure;
92            ReceivedEvent rcvEvt;
93            Serializable temperature;
94
95            protected int getTimeout()
96            {
97              return -1;
98            }
99            public void receiveEvent(int roundRobinTimeout) throws SMEPPException
100           {
101             rcvEvt = SMoL.receiveEvent(this, peerManager, "temperatureMonitoring",
102                         roundRobinTimeout);
103           }
```

Figure C.24: Operator Peer - Translated - Part 2

```
104             public void command() throws SMEPPException
105             {
106               if(peopleInTrouble.contains(rcvEvt.caller))
107               {
108                 ;
109               }
110               else
111               {
112                 do
113                 {
114                   if ( mustBeStopped() ) return;
115 
116                   temperature = SMoL.invokeRes(this, peerManager, rcvEvt.caller,
117                           "get_temperature", new Serializable[]{});
118                   if ( mustBeStopped() ) return;
119 
120                   bloodPressure = SMoL.invokeRes(this, peerManager, rcvEvt.caller,
121                           "get_blood_pressure", new Serializable[]{});
122                 } while((bloodPressure < 9) || (bloodPressure > 12) ||
123                   (temperature < 37) || (temperature > 38));
124               }
125             }
126           }
127           ,
128           new SMoLIHRcvEvtBranch( )
129           {
130             Serializable bloodPressure;
131             ReceivedEvent rcvEvt;
132             Serializable temperature;
133 
134             protected int getTimeout()
135             {
136               return -1;
137             }
138             public void receiveEvent(int roundRobinTimeout) throws SMEPPException
139             {
140               rcvEvt = SMoL.receiveEvent(this, peerManager, "bloodPressureMonitoring",
141                           roundRobinTimeout);
142             }
143             public void command() throws SMEPPException
144             {
145               if(peopleInTrouble.contains(rcvEvt.caller))
146               {
147                 ;
148               }
149               else
150               {
151                 do
152                 {
153                   if ( mustBeStopped() ) return;
154 
155                   temperature = SMoL.invokeRes(this, peerManager, rcvEvt.caller,
156                           "get_temperature", new Serializable[]{});
157                   if ( mustBeStopped() ) return;
158 
159                   bloodPressure = SMoL.invokeRes(this, peerManager, rcvEvt.caller,
160                           "get_blood_pressure", new Serializable[]{});
161                 } while((bloodPressure < 9) || (bloodPressure > 12) ||
162                   (temperature < 37) || (temperature > 38));
163               }
164             }
```

Figure C.25: Operator Peer - Translated - Part 3

```
165            }
166          }
167        );
168      }
169
170      /*
171        The following methods are always implemented in the same way
172      */
173      /**
174        @return true if the service must be stopped else false
175      **/
176      public boolean mustBeStopped()
177      {
178        return this.processManager.mustBeStopped();
179      }
180
181      /**
182        Stop the service execution
183      **/
184      public void stop()
185      {
186        this.processManager.stop();
187      }
188
189      /**
190        @see ISMEPPPeerClass
191      **/
192      public void addInnerThreadedCommand(IThreadedCommand cmd)
193      {
194        this.processManager.addInnerThreadedCommand(cmd);
195      }
196
197    }
```

Figure C.26: Operator Peer - Translated - Part 4

C.4 Implemented Codes

C.4.1 Equipment

Equipment Peer

```
1  package com.SequiTel.equipment;
2
3  import org.smepp.extendFramework.ISMEPPPeerClass;
4  import org.smepp.api.PeerManager;
5  import org.smepp.extendFramework.SMEPPProcessManager;
6  import org.smepp.api.exceptions.SMEPPException;
7  import org.smepp.extendFramework.IThreadedCommand;
8  import org.smepp.extendFramework.SMoL;
9  import org.smepp.datatypes.gm.GroupDescription;
10 import org.smepp.datatypes.gm.SecurityInformation;
11 import org.smepp.datatypes.smm.PeerServiceId;
12 import org.smepp.datatypes.api.Credentials;
13 import org.smepp.datatypes.api.ServiceGrounding;
14 import org.smepp.datatypes.gm.GroupId;
15 import org.smepp.smm.xml.ContractType;
16 import java.util.HashMap;
17 import java.util.logging.LogManager;
18
19 /**
20   @author SMoLTranslator
21 **/
22 public class EquipmentPeerImpl implements ISMEPPPeerClass
23 {
24   private PeerManager peerManager;
25   private SMEPPProcessManager processManager = null;
26   private Credentials myCredentials = new Credentials("");
27
28   private SecurityInformation securityInfoSequiTel = new SecurityInformation(0);
29   private GroupDescription groupSequiTelDescription =
30                            new GroupDescription("SequiTel-Operators",
31                              securityInfoSequiTel, "SequiTel-Operators␣group" );
32   private GroupId groupSequiTel;
33   private GroupId[] groupsSequiTel;
34
35   private SecurityInformation securityInfoFriends = new SecurityInformation(0);
36   private GroupDescription groupFriendsDescription =
37                            new GroupDescription("SequiTel-JohnSMITH",
38                              securityInfoFriends, "SequiTel-JohnSMITH␣group" );
39   private GroupId[] groupsFriends;
40   private GroupId groupFriends;
41
42   private ContractType monitoringServiceContract =
43     org.smepp.smm.xml.ContractLoader.loadXMLFromFile(
44         "src/com/SequiTel/equipment/monitoring.xml"
45     );
46   private ServiceGrounding monitoringServiceGrounding;
47   private PeerServiceId serviceSequiTelGroup;
48   private PeerServiceId serviceFriendsGroup;
```

Figure C.27: Equipment Peer - Implemented - Part 1

```
51    /**
52      The translated SMoL behavior
53      @throws SMEPPException
54    **/
55    public static void main(String[] args) throws Exception
56    {
57      new EquipmentPeerImpl();
58    }
59
60    /**
61      The translated SMoL behavior
62      @throws SMEPPException
63    **/
64    public EquipmentPeerImpl() throws SMEPPException
65    {
66      this.processManager = new SMEPPProcessManager();
67      LogManager.getLogManager().reset();
68
69      try
70      {
71        monitoringServiceGrounding = new org.smepp.api.SMEPPServiceGrounding(
72                com.SequiTel.equipment.EquipmentMonitoringImpl.class);
73        System.out.println("SequiTel:␣creating␣peer");
74        peerManager = PeerManager.newPeer(myCredentials,
75                "../SMEPP_TRANSLATOR/library/smepp_compiled/res/peer2.config");
76        System.out.println("SequiTel:␣created␣peer");
77      }
78      catch (Exception e)
79      {
80        throw new RuntimeException("Error␣while␣initializing␣peer", e);
81      }
82
83      groupsSequiTel = peerManager.getGroups(groupSequiTelDescription,myCredentials);
84      if((groupsSequiTel == null) || (groupsSequiTel.length == 0))
85      {
86        groupSequiTel = peerManager.createGroup(groupSequiTelDescription,myCredentials);
87      }
88      else
89      {
90        groupSequiTel = groupsSequiTel[0];
91        peerManager.joinGroup(groupSequiTel);
92      }
93      System.out.println("SequiTel:␣member␣of␣SequiTel␣group␣identified␣by␣"
94               + groupSequiTel);
95
96      groupsFriends = peerManager.getGroups(groupFriendsDescription, myCredentials);
97      if((groupsFriends == null) || (groupsFriends.length == 0))
98      {
99        groupFriends = peerManager.createGroup(groupFriendsDescription, myCredentials);
100     }
101     else
102     {
103       groupFriends = groupsFriends[0];
104       peerManager.joinGroup(groupFriends);
105     }
```

Figure C.28: Equipment Peer - Implemented - Part 2

```
106       System.out.println("SequiTel:␣member␣of␣friends␣group␣identified␣by␣" +
107                 groupFriends);
108
109       serviceSequiTelGroup = peerManager.publish(groupSequiTel,
110                         monitoringServiceContract,
111                         monitoringServiceGrounding, null, new HashMap());
112       System.out.println(
113           "SequiTel:␣service␣published␣in␣the␣SequiTel␣group,␣its␣id␣:␣" +
114           serviceSequiTelGroup + "␣group␣service␣identifier␣:␣" +
115           serviceSequiTelGroup.getGroupServiceId());
116
117       serviceFriendsGroup = peerManager.publish(groupFriends,
118                         monitoringServiceContract,
119                         monitoringServiceGrounding, null, new HashMap());
120       System.out.println(
121           "SequiTel:␣service␣published␣in␣the␣friends␣group,␣its␣id␣:␣" +
122           serviceFriendsGroup + "␣group␣service␣identifier␣:␣" +
123           serviceFriendsGroup.getGroupServiceId());
124
125       SMoL.waitFor( this, "P1Y" );
126     }
127
128     /*
129       The following methods are always implemented in the same way
130     */
131     /**
132       @return true if the service must be stopped else false
133     **/
134     public boolean mustBeStopped()
135     {
136       return this.processManager.mustBeStopped();
137     }
138
139     /**
140       Stop the service execution
141     **/
142     public void stop()
143     {
144       this.processManager.stop();
145     }
146
147     /**
148       @see ISMEPPPeerClass
149     **/
150     public void addInnerThreadedCommand(IThreadedCommand cmd)
151     {
152       this.processManager.addInnerThreadedCommand(cmd);
153     }
154
155   }
```

Figure C.29: Equipment Peer - Implemented - Part 3

Equipment Monitoring

```
1  package com.SequiTel.equipment;
2
3  import org.smepp.api.ServiceManager;
4  import java.io.Serializable;
5  import org.smepp.api.exceptions.SMEPPException;
6  import org.smepp.extendFramework.*;
7  import org.smepp.datatypes.em.EventInfo;
8  import org.smepp.api.misc.ReceivedMessage;
9
10 /**
11   @author SMoLTranslator
12 **/
13 public class EquipmentMonitoringImpl implements ISMEPPServiceClass
14 {
15   private ServiceManager serviceManager;
16   private SMEPPProcessManager processManager = null;
17
18   /**
19     @param svc the API service manager instance
20     @param constrValue values which permit to initialize this object
21   **/
22   public EquipmentMonitoringImpl(ServiceManager svc, Serializable constrValue)
23   {
24     serviceManager = svc;
25   }
26
27   /**
28     The translated SMoL behavior
29     @throws SMEPPException
30   **/
31   public void service() throws SMEPPException
32   {
33     System.out.println("SequiTel␣:␣service␣started");
34     new SMoLFlow( this,
35       new SMoLFlowBranch[]{
36         new SMoLFlowBranch(  )
37         {
38           public void command() throws SMEPPException
39           {
40             new SMoLIH( this,
41               new SMoLIHMainCommand(  )
42               {
43                 public void command() throws SMEPPException
44                 {
45                   SMoL.waitFor( this, "P1Y" );
46
47                   throw new ApplicationTermination(  );
48                 }
49               },
```

Figure C.30: Equipment Monitoring - Implemented - Part 1

```
50                  new SMoLIHBranch[]{
51                    new SMoLIHRcvMsgBranch(  )
52                    {
53                      ReceivedMessage rcvMsg;
54                      float temperature;
55
56                      protected int getTimeout()
57                      {
58                        return -1;
59                      }
60                      public void receiveMessage(int roundRobinTimeout)
61                        throws SMEPPException
62                      {
63                        rcvMsg = SMoL.receiveMessage(this, serviceManager,
64                                      "get_temperature",
65                                      new Class[]{}, roundRobinTimeout);
66                      }
67                      public void command() throws SMEPPException
68                      {
69                        temperature = get_temperature();
70                        System.out.println("SequiTel␣:␣My␣temperature␣:␣" + temperature);
71                        serviceManager.reply(rcvMsg.caller, "get_temperature",
72                                  new Serializable[]{temperature}, null);
73                      }
74                    },
75                    new SMoLIHRcvMsgBranch(  )
76                    {
77                      ReceivedMessage rcvMsg;
78                      float bloodPressure;
79
80                      protected int getTimeout()
81                      {
82                        return -1;
83                      }
84                      public void receiveMessage(int roundRobinTimeout)
85                        throws SMEPPException
86                      {
87                        rcvMsg = SMoL.receiveMessage(this, serviceManager,
88                                      "get_blood_pressure",
89                                      new Class[]{}, roundRobinTimeout);
90                      }
91                      public void command() throws SMEPPException
92                      {
93                        bloodPressure = get_blood_pressure();
94                        System.out.println("SequiTel␣:␣My␣blood␣pressure␣:␣" +
95                              bloodPressure);
96                        serviceManager.reply(rcvMsg.caller, "get_blood_pressure",
97                                  new Serializable[]{bloodPressure}, null);
98                      }
99                    }
100                 }
101               );
102             }
103           },
```

Figure C.31: Equipment Monitoring - Implemented - Part 2

```
104         new SMoLFlowBranch(  )
105         {
106           float bloodPressure;
107           float temperature;
108
109           public void command() throws SMEPPException
110           {
111             while(true)
112             {
113               bloodPressure = get_blood_pressure();
114               temperature = get_temperature();
115
116               if((new Float(temperature) < 37) || (new Float(temperature) > 38))
117               {
118                 System.out.println("SequiTel␣:␣ALERT␣!!!␣My␣temperature␣:␣"
119                           + temperature);
120                 serviceManager.event("temperatureMonitoring",
121                           new EventInfo("JOHN␣SMITH") );
122               }
123
124               if((new Float(bloodPressure) < 9) || (new Float(bloodPressure) > 12))
125               {
126                 System.out.println("SequiTel␣:␣ALERT␣!!!␣My␣blood␣pressure␣:␣"
127                           + bloodPressure);
128                 serviceManager.event("bloodPressureMonitoring",
129                           new EventInfo("JOHN␣SMITH") );
130               }
131
132               SMoL.waitFor( this, "P0DT0H1M0S" );
133             }
134           }
135         }
136       }
137     );
138   }
```

Figure C.32: Equipment Monitoring - Implemented - Part 3

```
140     public float get_blood_pressure()
141     {
142       return (float) (Math.random() * 10);
143     }
144
145     public float get_temperature()
146     {
147       return (float) (Math.random() * 30);
148     }
149
150     /*
151       The following methods are always implemented in the same way
152     */
153     /**
154       @return true if the service must be stopped else false
155     **/
156     public boolean mustBeStopped()
157     {
158       return this.processManager.mustBeStopped();
159     }
160
161     /**
162       Stop the service execution
163     **/
164     public void stop()
165     {
166       this.processManager.stop();
167     }
168
169     /**
170       @see Runnable#run()
171     **/
172     public void run()
173     {
174       try
175       {
176         this.processManager = new SMEPPProcessManager();
177         this.service();
178       }
179       catch ( SMEPPException e )
180       {
181         e.printStackTrace();
182       }
183     }
184
185     /**
186       @see ISMEPPServiceClass
187     **/
188     public void addInnerThreadedCommand(IThreadedCommand cmd)
189     {
190       this.processManager.addInnerThreadedCommand(cmd);
191     }
192
193   }
```

Figure C.33: Equipment Monitoring - Implemented - Part 4

C.4.2 Operator

Operator Peer

```
1  package com.SequiTel.operator;
2
3  import org.smepp.extendFramework.ISMEPPPeerClass;
4  import org.smepp.api.PeerManager;
5  import org.smepp.extendFramework.SMEPPProcessManager;
6  import org.smepp.api.exceptions.SMEPPException;
7  import org.smepp.extendFramework.IThreadedCommand;
8  import org.smepp.extendFramework.SMoLIH;
9  import org.smepp.extendFramework.SMoLIHBranch;
10 import org.smepp.extendFramework.SMoLIHMainCommand;
11 import org.smepp.extendFramework.SMoL;
12 import org.smepp.extendFramework.SMoLIHRcvEvtBranch;
13 import org.smepp.datatypes.gm.GroupId;
14 import org.smepp.datatypes.gm.GroupDescription;
15 import org.smepp.datatypes.gm.SecurityInformation;
16 import org.smepp.datatypes.api.Credentials;
17 import org.smepp.datatypes.api.EntityId;
18
19 import java.io.Serializable;
20 import java.util.logging.LogManager;
21
22 import org.smepp.api.misc.ReceivedEvent;
23
24 /**
25   @author SMoLTranslator
26 **/
27 public class OperatorPeerImpl implements ISMEPPPeerClass
28 {
29   private PeerManager peerManager;
30   private SMEPPProcessManager processManager = null;
31   private GroupId groupSequiTel;
32   private GroupId[] groupsSequiTel;
33   private SecurityInformation securityInfoSequiTel = new SecurityInformation(0);
34   private GroupDescription groupSequiTelDescription =
35                             new GroupDescription("SequiTel-Operators",
36                                         securityInfoSequiTel,
37                                         "SequiTel-Operators␣group" );
38   private Credentials myCredentials = new Credentials("");
39   private java.util.List<EntityId> peopleInTrouble =
40             new java.util.LinkedList<EntityId>();
41
42
43   /**
44     The translated SMoL behavior
45     @throws SMEPPException
46   **/
47   public static void main(String[] args) throws Exception
48   {
49     new OperatorPeerImpl();
50   }
```

Figure C.34: Operator Peer - Implemented - Part 1

```
52     /**
53       The translated SMoL behavior
54       @throws SMEPPException
55     **/
56     public OperatorPeerImpl() throws SMEPPException
57     {
58       this.processManager = new SMEPPProcessManager();
59       LogManager.getLogManager().reset();
60
61       try
62       {
63         System.out.println("SequiTel:␣creating␣peer");
64         peerManager = PeerManager.newPeer(myCredentials,
65                 "../SMEPP_TRANSLATOR/library/smepp_compiled/res/peer1.config");
66         System.out.println("SequiTel:␣created␣peer");
67       }
68       catch (Exception e)
69       {
70         throw new RuntimeException("Error␣while␣initializing␣peer", e);
71       }
72
73       groupsSequiTel = peerManager.getGroups(groupSequiTelDescription,myCredentials);
74       if((groupsSequiTel == null) || (groupsSequiTel.length == 0))
75       {
76         groupSequiTel = peerManager.createGroup(groupSequiTelDescription,myCredentials);
77       }
78       else
79       {
80         groupSequiTel = groupsSequiTel[0];
81         if ( mustBeStopped() ) return;
82
83         peerManager.joinGroup(groupSequiTel);
84       }
85
86       System.out.println("SequiTel␣:␣JOIN␣GROUP␣" + groupSequiTel);
87
88       peerManager.subscribe("temperatureMonitoring");
89       peerManager.subscribe("bloodPressureMonitoring");
90
91       new SMoLIH( this,
92         new SMoLIHMainCommand(  )
93         {
94           public void command() throws SMEPPException
95           {
96             System.out.println("SequiTel␣:␣begin␣execution");
97             SMoL.waitFor( this, "P1Y" );
98           }
99         }
100        , new SMoLIHBranch[]{
101          new SMoLIHRcvEvtBranch(  )
102          {
103            float temperature;
104            float bloodPressure;
105            ReceivedEvent rcvEvt;
106
107            protected int getTimeout()
108            {
109              return -1;
110            }
```

Figure C.35: Operator Peer - Implemented - Part 2

```
111             public void receiveEvent(int roundRobinTimeout) throws SMEPPException
112             {
113               rcvEvt = SMoL.receiveEvent(this, peerManager, "temperatureMonitoring",
114                               roundRobinTimeout);
115             }
116             public void command() throws SMEPPException
117             {
118               if(peopleInTrouble.contains(rcvEvt.caller))
119               {
120                 System.out.println("SequiTel␣:␣ALERT␣RECEIVED␣FROM␣:␣" +
121                           rcvEvt.caller + "␣IGNORE␣");
122                 ;
123               }
124               else
125               {
126                 System.out.println("SequiTel␣:␣NEW␣ALERT␣RECEIVED␣FROM␣:␣" +
127                           rcvEvt.caller + "␣FROM␣" + rcvEvt.eventInfo);
128                 peopleInTrouble.add(rcvEvt.caller);
129                 do
130                 {
131                   temperature = new Float( SMoL.invokeRes(this, peerManager,
132                                 rcvEvt.caller, "get_temperature",
133                                 new Serializable[]{}).toString() );
134                   bloodPressure = new Float( SMoL.invokeRes(this, peerManager,
135                                 rcvEvt.caller, "get_blood_pressure",
136                                 new Serializable[]{}).toString() );
137                 } while((bloodPressure < 9) || (bloodPressure > 12) ||
138                     (temperature < 37) || (temperature > 38));
139               }
140
141               System.out.println("SequiTel␣:␣END␣OF␣ALERT␣:␣" + rcvEvt.eventInfo);
142               peopleInTrouble.remove(rcvEvt.caller);
143             }
144           }
145           ,
146           new SMoLIHRcvEvtBranch(  )
147           {
148             ReceivedEvent rcvEvt;
149             float temperature;
150             float bloodPressure;
151
152             protected int getTimeout()
153             {
154               return -1;
155             }
156             public void receiveEvent(int roundRobinTimeout) throws SMEPPException
157             {
158               rcvEvt = SMoL.receiveEvent(this, peerManager, "bloodPressureMonitoring",
159                               roundRobinTimeout);
160             }
```

Figure C.36: Operator Peer - Implemented - Part 3

```
161            public void command() throws SMEPPException
162            {
163              if(peopleInTrouble.contains(rcvEvt.caller))
164              {
165                System.out.println("SequiTel␣:␣ALERT␣RECEIVED␣FROM␣:␣"
166                           + rcvEvt.caller + "␣IGNORE␣");
167                ;
168              }
169              else
170              {
171                System.out.println("SequiTel␣:␣NEW␣ALERT␣RECEIVED␣FROM␣:␣"
172                           + rcvEvt.caller + "␣FROM␣" + rcvEvt.eventInfo);
173                peopleInTrouble.add(rcvEvt.caller);
174                do
175                {
176                  temperature = new Float( SMoL.invokeRes(this, peerManager,
177                              rcvEvt.caller, "get_temperature",
178                              new Serializable[]{}).toString() );
179                  bloodPressure = new Float( SMoL.invokeRes(this, peerManager,
180                              rcvEvt.caller, "get_blood_pressure",
181                              new Serializable[]{}).toString() );
182                } while((bloodPressure < 9) || (bloodPressure > 12) ||
183                    (temperature < 37) || (temperature > 38));
184              }
185
186              System.out.println("SequiTel␣:␣END␣OF␣ALERT␣:␣" + rcvEvt.eventInfo);
187              peopleInTrouble.remove(rcvEvt.caller);
188            }
189          }
190        }
191      );
192    }
193
194    /*
195      The following methods are always implemented in the same way
196    */4
197    /**
198      @return true if the service must be stopped else false
199    **/
200    public boolean mustBeStopped()
201    {
202      return this.processManager.mustBeStopped();
203    }
204
205    /**
206      Stop the service execution
207    **/
208    public void stop()
209    {
210      this.processManager.stop();
211    }
212
213    /**
214      @see ISMEPPPeerClass
215    **/
216    public void addInnerThreadedCommand(IThreadedCommand cmd)
217    {
218      this.processManager.addInnerThreadedCommand(cmd);
219    }
220
221  }
```

Figure C.37: Operator Peer - Implemented - Part 4

Appendix D

SMoL Codes Translation

D.1 Expressions Management

Figure D.1: Variables Declaration

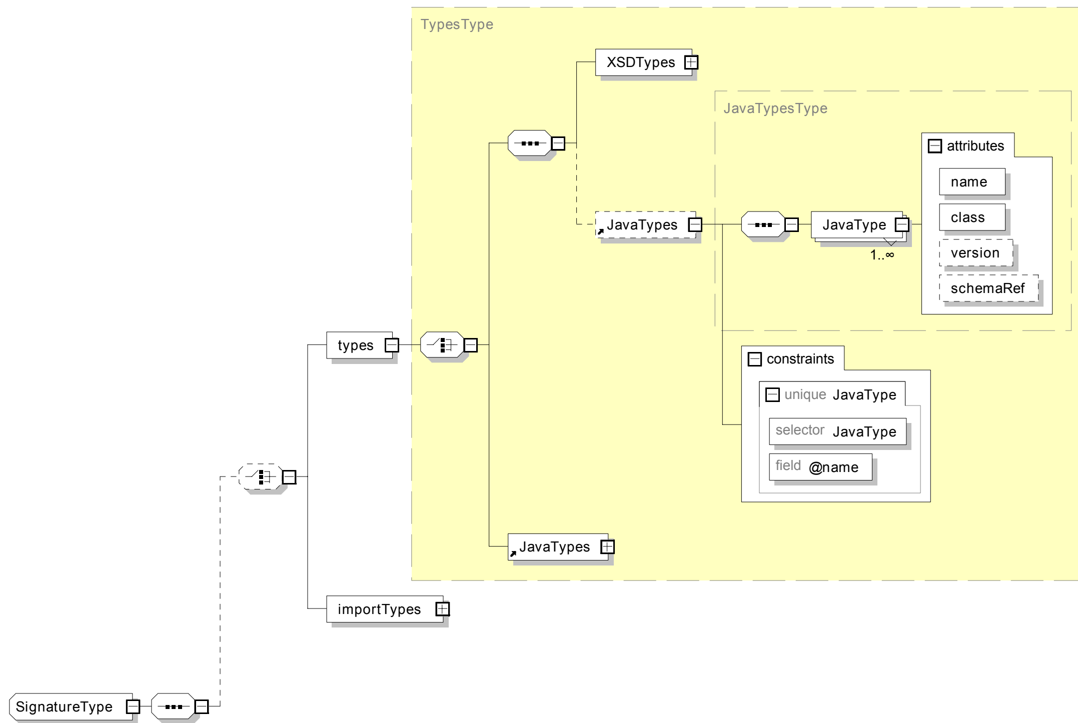

Figure D.2: Types Declaration

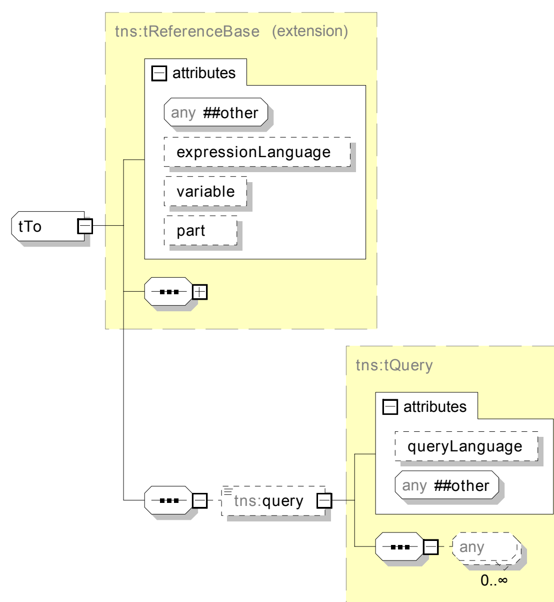

Figure D.3: SMoL Left Expression

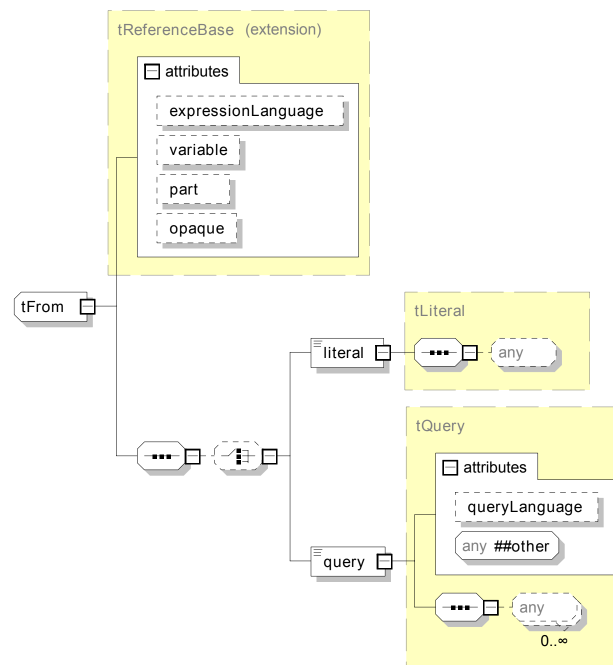

Figure D.4: SMoL Right Expression

D.2 Exceptions Manager

```
try {
      mainCommand
}
catch(ExceptionType nameObject){
      command1
}
catch(ExceptionType2 nameObject){
      command2
}
```

Figure D.5: Java try-catch

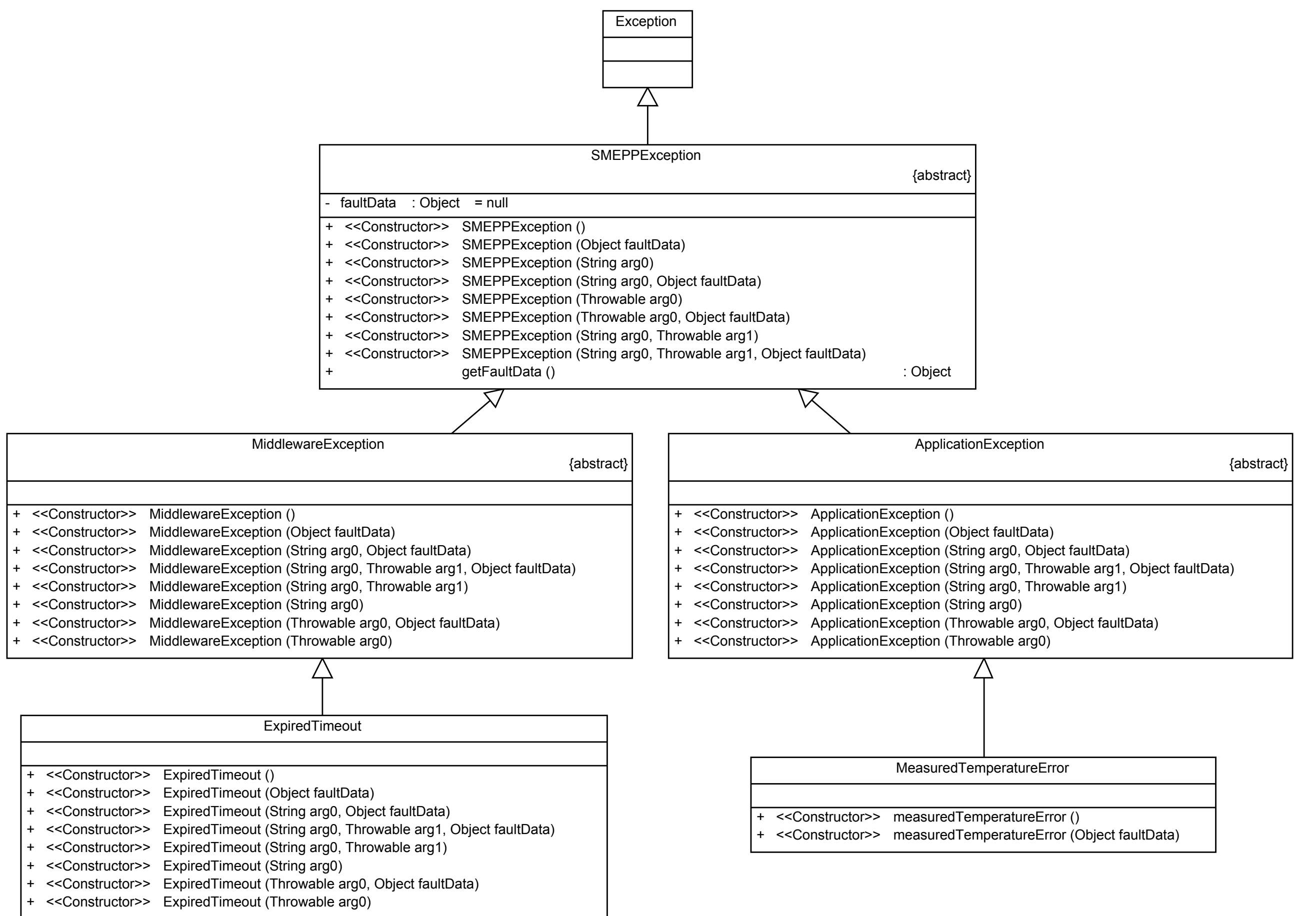

Figure D.6: Java Exception Hiearchy

```
<FaultHandler>
            <Throw>
                    <faultName>measuredTemperatureError</faultName>
                    <faultDatavariable="measuredTemperature"/>
            </Throw>
            <Catch>
                    <faultName>measuredTemperatureError</faultName>
                    <faultDatavariable="uncorrectMeasure"/>
                    <Empty/>
            </Catch>
            <CatchAll>
                    <actualFaultvariable="fault"/>
                    <Empty/>
            </CatchAll>
</FaultHandler>
```

Figure D.7: Exception Usage Example

```
try {
      throw new measuredTemperatureError (measuredTemperature)
}
catch (measuredTemperatureError exception27940) {
      uncorrectMeasure = exception27940.getFaultData ( ) ;
      ;
}
catch (SMEPPException exception21345) {
      fault = exception21345;
      ;
}
```

Figure D.8: Translation of Exception Commands

```
package org.smepp.api.exceptions.CUSTOM;
import org.smepp.api.exceptions.ApplicationException;
public class measuredTemperatureError extends ApplicationException {
             public measuredTemperatureError ( ) {
             }
             public measuredTemperatureError ( Object faultData ) {
                    this.faultData = faultData;
             }
}
```

Figure D.9: Translated SMoL Exception

D.3 Structured Command

D.3.1 SMoL Flow

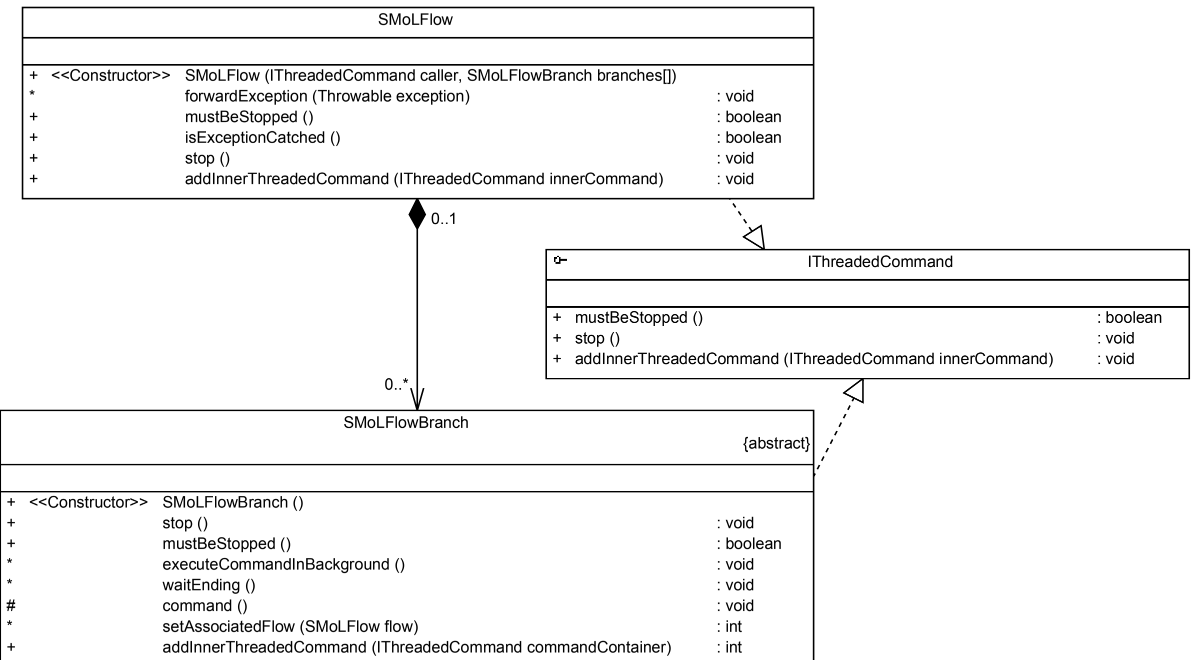

Figure D.10: SMoL Flow - Class Diagram

```
1  /*  <Flow>
2      <Empty />
3      <Empty />
4    </Flow>  */
5  new SMoLFlow( this, new SMoLFlowBranch[]{
6      new SMoLFlowBranch(  ) {
7        public void command() throws SMEPPException { ; }
8      } ,
9      new SMoLFlowBranch(  ) {
10       public void command() throws SMEPPException { ; }
11     }
12   } );
```

Figure D.11: Translation of A SMoL Flow

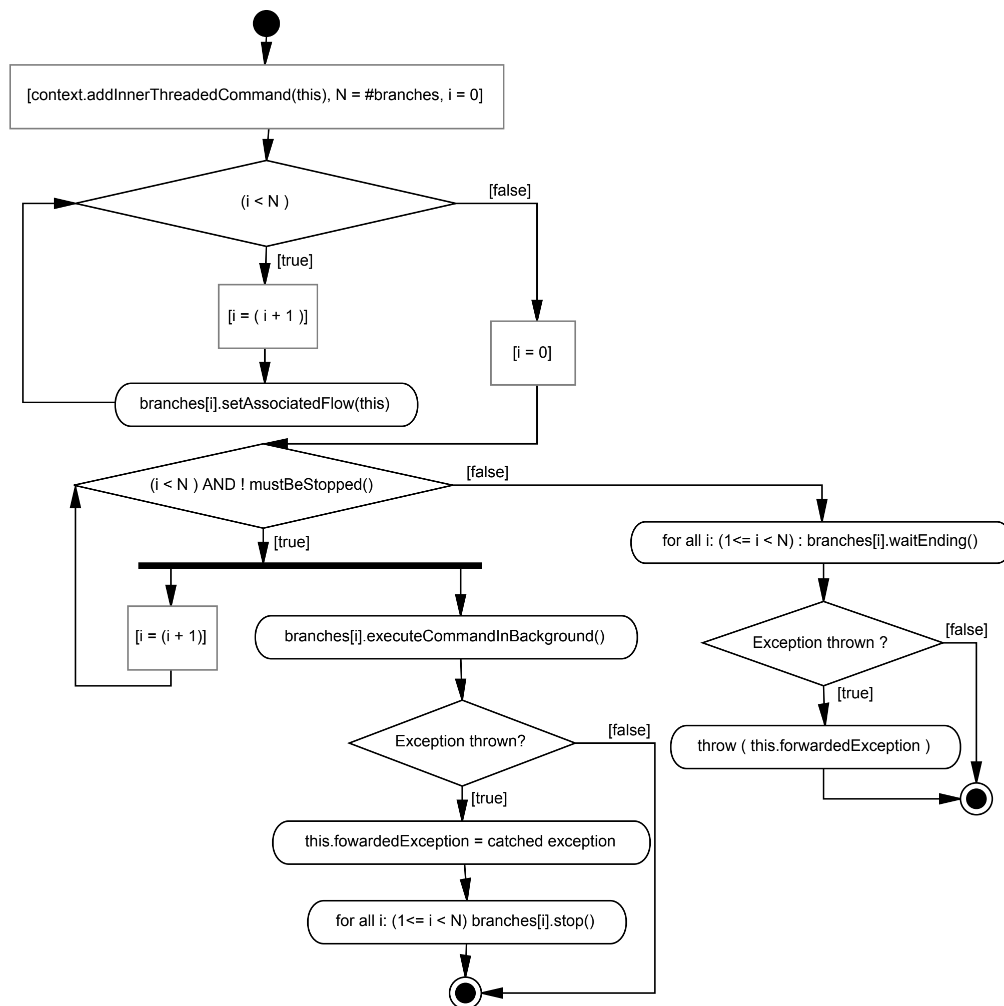

Figure D.12: SMoL Flow - Implementation

D.3.2 SMoL Pick

Figure D.13: SMoL Pick - Class Diagram

```
1  /*  <Pick>
2      <RcvMsgHandler>
3        <variables>
4          <variable name="rcvMsg" />
5          <variable name="opName" />
6          <variable name="input" />
7          <variable name="time" />
8        </variables>
9        <RcvMsg>
10         <result variable="rcvMsg" />
11         <operationName variable="opName" />
12         <inputType variable="input" />
13         <timeout variable="time" />
14       </RcvMsg>
15       <Empty />
16     </RcvMsgHandler>
17
18     <RcvResHandler>
19       <variables>
20         <variable name="timeout" />
21         <variable name="id" />
22         <variable name="opName" />
23         <variable name="result" />
24       </variables>
25       <RcvRes>
26         <result variable="result" />
27         <id variable="id" />
28         <opName variable="opName" />
29         <timeout variable="timeout" />
30       </RcvRes>
31       <Empty />
32     </RcvResHandler>
33
34     <RcvEvtHandler>
35       <variables>
36         <variable name="eventName" />
37         <variable name="result" />
38         <variable name="timeout" />
39         <variable name="groupId" />
40       </variables>
41       <RcvEvt>
42         <result variable="result" />
43         <eventName variable="eventName" />
44         <timeout variable="timeout" />
45         <groupId variable="groupId" />
46       </RcvEvt>
47       <Empty />
48     </RcvEvtHandler>
49
50     <TimedHandler>
51       <variables>
52         <variable name="forValue" >
53           <init> <literal>"PODTOHOM5S"</literal> </init>
54         </variable>
55       </variables>
56       <Wait>
57         <for variable="forValue" />
58         <repeatEvery variable="repeatEvery" />
59       </Wait>
60       <Empty />
61     </TimedHandler>
62   </Pick> */
```

Figure D.14: Translation of A SMoL Pick - Part 1

```
63 new SMoLPick( this,
64   new SMoLPickBranch[]{
65     new SMoLPickRcvMsgBranch(  ) {
66       ReceivedMessage rcvMsg;
67       int time;
68       Class[] input;
69       String opName;
70
71       protected int getTimeout() { return time; }
72       public void receiveMessage(int roundRobinTimeout) throws SMEPPException
73       {
74         rcvMsg = SMoL.receiveMessage(this, serviceManager, opName, input, .
75                     roundRobinTimeout);
76       }
77       public void command() throws SMEPPException {  ; }
78     } ,
79     new SMoLPickRcvResBranch(  )
80     {
81       Object id;
82       Serializable result;
83       String opName;
84       int timeout;
85
86       protected int getTimeout() { return timeout; }
87       public void receiveResponse(int roundRobinTimeout) throws SMEPPException
88       {
89         result = SMoL.receiveResponse(this, serviceManager, id, opName,
90                     roundRobinTimeout);
91       }
92       public void command() throws SMEPPException {  ; }
93     } ,
94     new SMoLPickRcvEvtBranch(  )
95     {
96       Object groupId;
97       ReceivedEvent result;
98       String eventName;
99       int timeout;
100
101      protected int getTimeout() { return timeout; }
102      public void receiveEvent(int roundRobinTimeout) throws SMEPPException
103      {
104        result = SMoL.receiveEvent(this, serviceManager, eventName,
105                   roundRobinTimeout);
106      }
107      public void command() throws SMEPPException {  ; }
108    } ,
109    new SMoLPickAlarmBranch(  )
110    {
111      String forValue = "P0DT0H0M5S";
112      protected String getForParameter() { return forValue; }
113      protected String getUntilParameter() { return null; }
114      public void command() throws SMEPPException {  ; }
115    }
116  }
117 );
```

Figure D.15: Translation of A SMoL Pick - Part 2

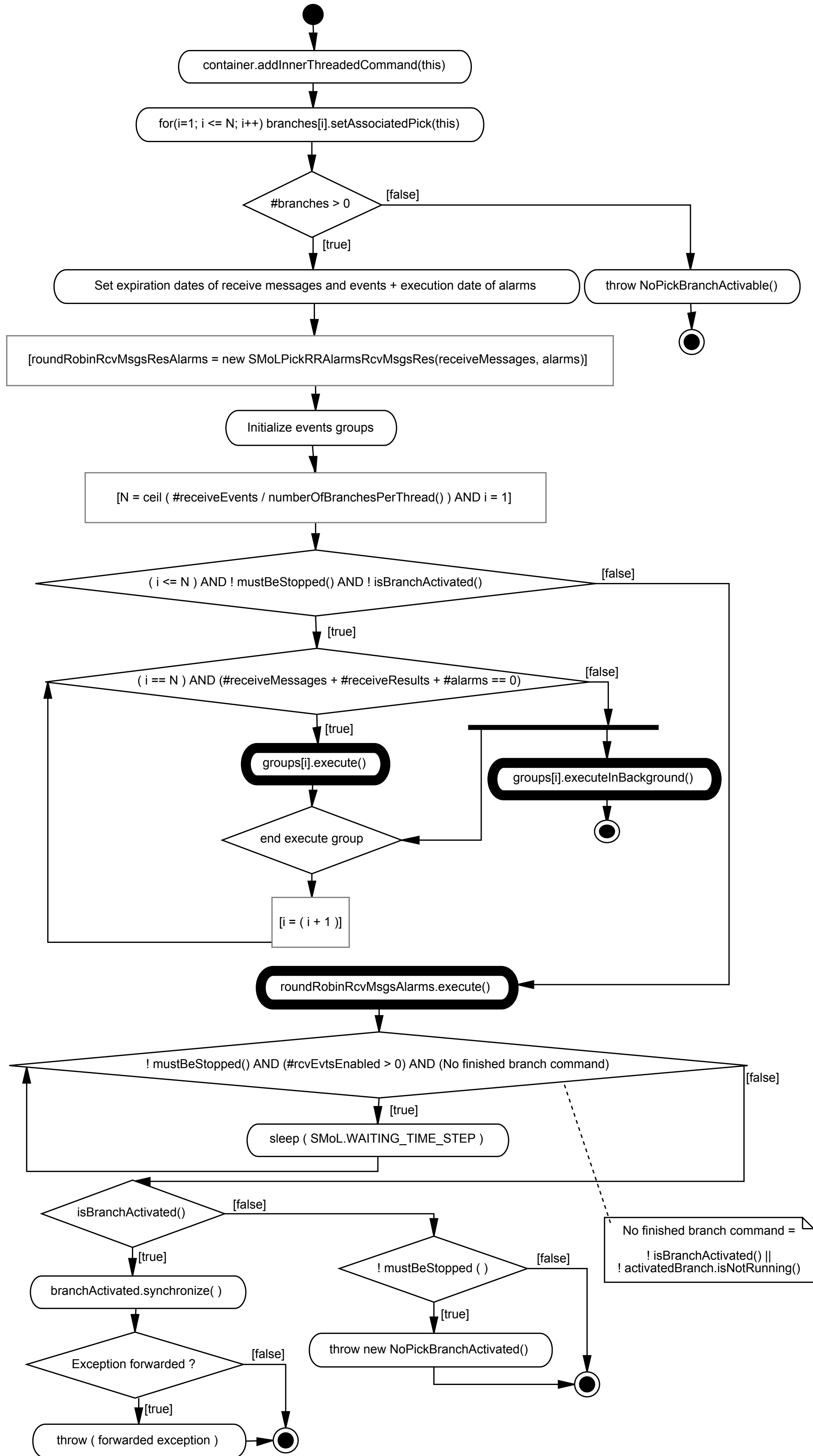

Figure D.16: SMoL Pick - Behaviour

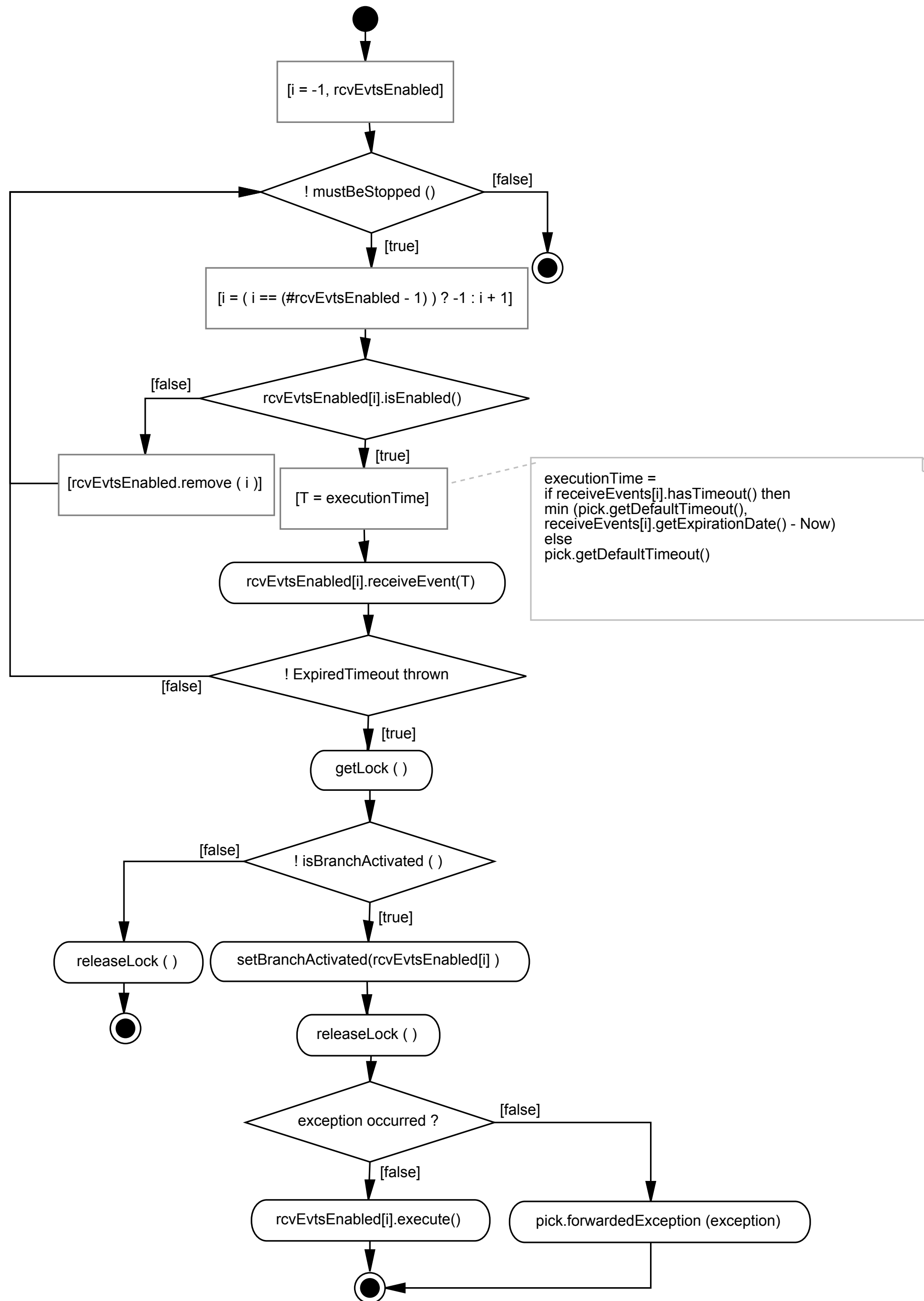

Figure D.17: SMoL Pick - Events Round Robin

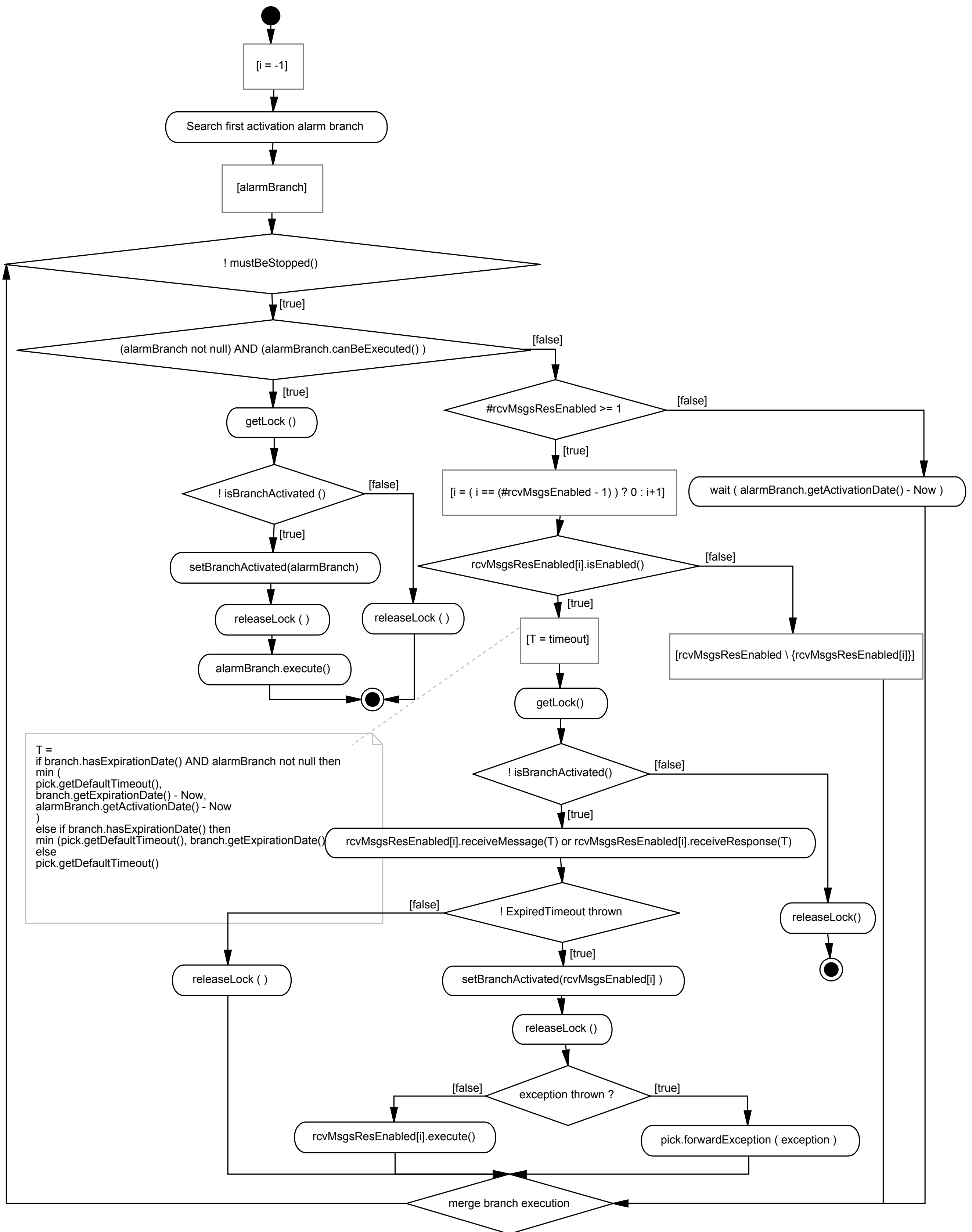

Figure D.18: SMoL Pick - Receive Messages, Response and Alarms Round Robin

D.3.3 SMoL Information Handler

SMoLIH

+ <<Constructor>> SMoLIH (IThreadedCommand caller, SMoLIHMainCommand mainCommand, SMoLIHBranch branches[])
+ numberOfRcvEvtsPerThread () : int
+ getDefaultTimeout () : long
* forwardException (Throwable fault) : void
- createSMoLIHBranchesGroup (SMoLIHRcvEvtBranch events[], int N) : SMoLIHRcvEvtsGroup[]
+ getMainCommand () : SMoLIHMainCommand

mainCommand

1..1

0..*

SMoLIHMainCommand {abstract}

+ <<Constructor>> SMoLIHMainCommand ()
# command () : void
* executeInBackground () : void
* execute () : void
* isRunning () : boolean
* waitEnding () : void
+ setAssociatedInformationHandler (SMoLIH IH) : void

SMoLIHBranch {abstract}

+ <<Constructor>> SMoLIHBranch ()
# command () : void
* executeCommand () : void
* executeCommandInBackground () : void
* waitEnding () : void
* isEnabled () : boolean
+ setAssociatedInformationHandler (SMoLIH IH) : void

IThreadedCommand

+ mustBeStopped () : boolean
+ stop () : void
+ addInnerThreadedCommand (IThreadedCommand command) : void

SMoLIHRcvEvtBranch {abstract}

+ <<Constructor>> SMoLIHRcvEvtBranch ()
* setAssociatedInformationHandler (SMoLIH informationHandler) : void
* getExpirationDate () : java.util.Date
# hasTimeout () : boolean
# getTimeout () : int
+ receiveEvent (int roundRobinTimeout, long timeout) : void

1..*

SMoLIHRcvMsgBranch {abstract}

+ <<Constructor>> SMoLIHRcvMsgBranch ()
* hasTimeout () : boolean
* getExpirationDate () : java.util.Date
* receiveMessage (long timeout) : void

0..*

1..1

SMoLIHRRAlarmsRcvMsgsRes

+ <<Constructor>> SMoLIHRRAlarmsRcvMsgsRes (SMoLIH IH, SMoLIHBranch rcvMsgsRes[], SMoLIHAlarmBranch alarms[])
* execute () : void
- getNextAlarmBranchActivation () : SMoLIHAlarmBranch
- countBranchesEnabled () : int
- noRcvMsgEnabled () : boolean
+ mustBeStopped () : boolean

1..1

1..1

1..1

0..*

SMoLIHRcvEvtsGroup

+ <<Constructor>> SMoLIHRcvEvtsGroup (SMoLIHRcvEvtBranch events[], SMoLIH ih)
+ execute () : void
+ executeInBackground () : void
+ mustBeStopped () : boolean
- countBranchesEnabled () : int

SMoLIHAlarmBranch {abstract}

+ <<Constructor>> SMoLIHAlarmBranch (String forTime, String untilTime, String repeatEveryTime)
* isRepeatEvery () : boolean
+ getActivationDate () : Date
+ canBeExecuted () : boolean
+ getForParameter () : String
+ getUntilPararameter () : String
+ getRepeatEveryParameter () : String

0..*

SMoLIHRcvResBranch {abstract}

+ <<Constructor>> SMoLIHRcvResBranch ()
* setAssociatedInformationHandler (SMoLIH informationHandler) : void
* getExpirationDate () : java.util.Date
* isEnabled () : boolean
# hasTimeout () : boolean
# getTimeout () : int
+ receiveResponse (int roundRobinTimeout) : void

Figure D.19: SMoL Information Handler - Class Diagram

```
1  /*   <InfoHandler>
2        <Empty />
3
4        <RcvMsgHandler>
5          <variables>
6            <variable name="rcvMsg" />
7            <variable name="opName" />
8            <variable name="input" />
9            <variable name="time" />
10         </variables>
11         <RcvMsg>
12           <result variable="rcvMsg" />
13           <operationName variable="opName" />
14           <inputType variable="input" />
15           <timeout variable="time" />
16         </RcvMsg>
17         <Empty />
18       </RcvMsgHandler>
19       <RcvResHandler>
20         <variables>
21           <variable name="timeout" />
22           <variable name="id" />
23           <variable name="opName" />
24           <variable name="result" />
25         </variables>
26         <RcvRes>
27           <result variable="result" />
28           <id variable="id" />
29           <opName variable="opName" />
30           <timeout variable="timeout" />
31         </RcvRes>
32         <Empty />
33       </RcvResHandler>
34       <RcvEvtHandler>
35         <variables>
36           <variable name="rcvEvt" />
37           <variable name="event" />
38           <variable name="timeout" />
39         </variables>
40         <RcvEvt>
41           <result variable="rcvEvt" />
42           <eventName variable="event" />
43           <timeout variable="timeout" />
44         </RcvEvt>
45         <Empty />
46       </RcvEvtHandler>
47       <RepTimedHandler>
48         <variables>
49           <variable name="forValue" >
50             <init> <literal>"P0DT0H0M5S"</literal> </init>
51           </variable>
52           <variable name="repeatEvery">
53             <init> <literal>"P0DT0H0M1S"</literal> </init>
54           </variable>
55         </variables>
56         <RepWait>
57           <for variable="forValue" />
58           <repeatEvery variable="repeatEvery" />
59         </RepWait>
60         <Empty />
61       </RepTimedHandler>
62     </InfoHandler> */
```

Figure D.20: Translation of a SMoL Information Handler - Part 1

```
63  new SMoLIH( this,
64    new SMoLIHMainCommand(  )
65    {
66      public void command() throws SMEPPException  {  ; }
67    } ,
68    new SMoLIHBranch[]{
69      new SMoLIHRcvMsgBranch(  )
70      {
71        ReceivedMessage rcvMsg;
72        int time;
73        Class[] input;
74        String opName;
75
76        protected int getTimeout() { return time; }
77        public void receiveMessage(int roundRobinTimeout) throws SMEPPException
78        {
79          rcvMsg = SMoL.receiveMessage(this, serviceManager, opName, input,
80                          roundRobinTimeout);
81        }
82        public void command() throws SMEPPException { ; }
83      },
84      new SMoLIHRcvResBranch(  )
85      {
86        Object id;
87        Serializable result;
88        String opName;
89        int timeout;
90
91        protected int getTimeout() { return timeout; }
92        public void receiveResponse(int roundRobinTimeout) throws SMEPPException
93        {
94          result = SMoL.receiveResponse(this, serviceManager, id, opName,
95                          roundRobinTimeout);
96        }
97        public void command() throws SMEPPException { ; }
98      },
99      new SMoLIHRcvEvtBranch(  )
100     {
101       String event;
102       ReceivedEvent rcvEvt;
103       int timeout;
104
105       protected int getTimeout() {  return timeout; }
106       public void receiveEvent(int roundRobinTimeout) throws SMEPPException
107       {
108         rcvEvt = SMoL.receiveEvent(this, serviceManager, event,
109                        roundRobinTimeout);
110       }
111       public void command() throws SMEPPException { ; }
112     },
113     new SMoLIHAlarmBranch(  )
114     {
115       String forValue = "P0DT0H0M5S";
116       String repeatEvery = "P0DT0H0M1S";
117
118       protected String getForParameter() { return forValue; }
119       protected String getUntilParameter() { return null; }
120       protected String getRepeatEveryParameter() { return repeatEvery; }
121       protected void command()  throws SMEPPException { ; }
122     }
123   }
124 );
```

Figure D.21: Translation of a SMoL Information Handler - Part 2

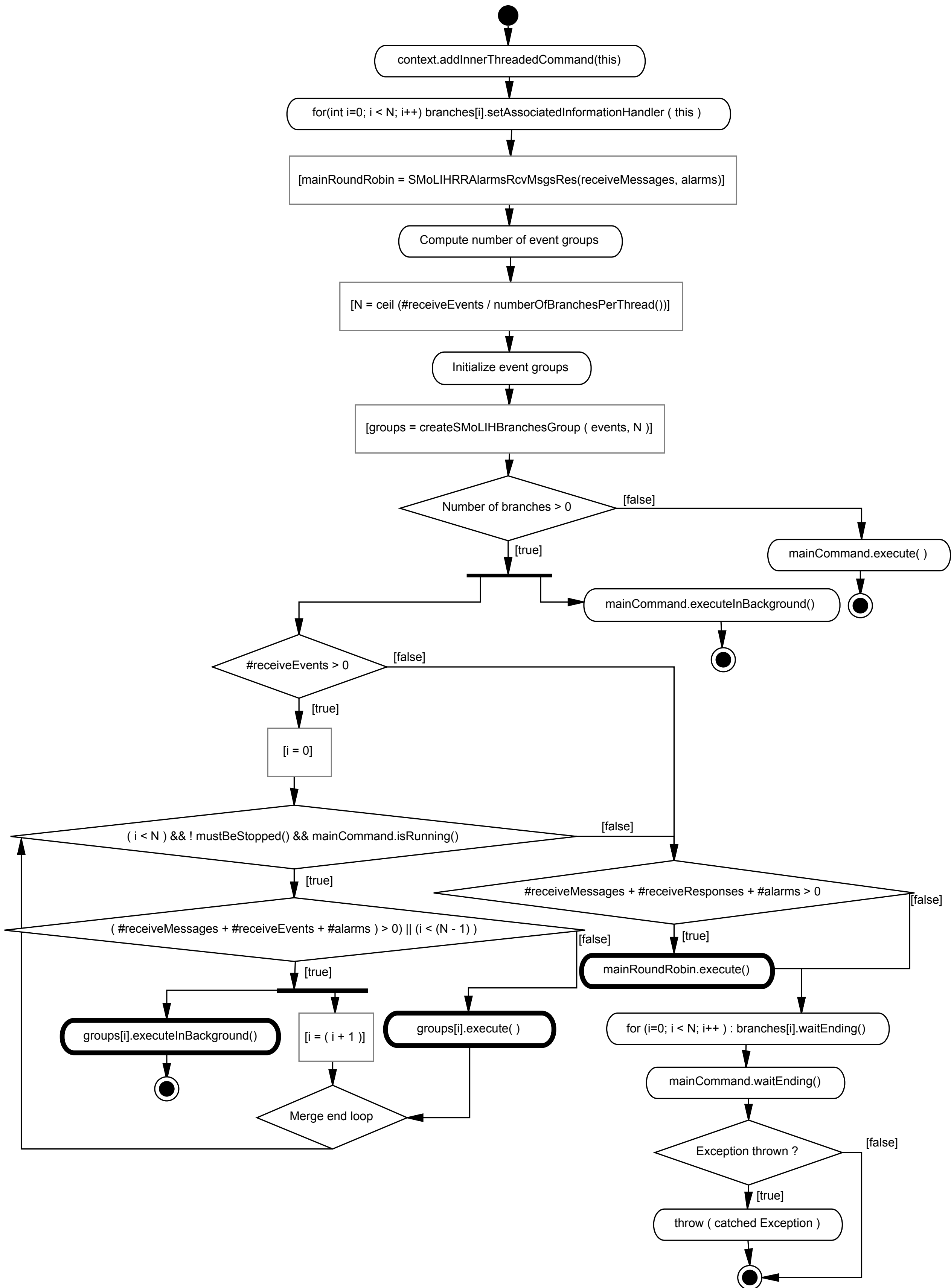

Figure D.22: SMoL Information Handler - Behaviour

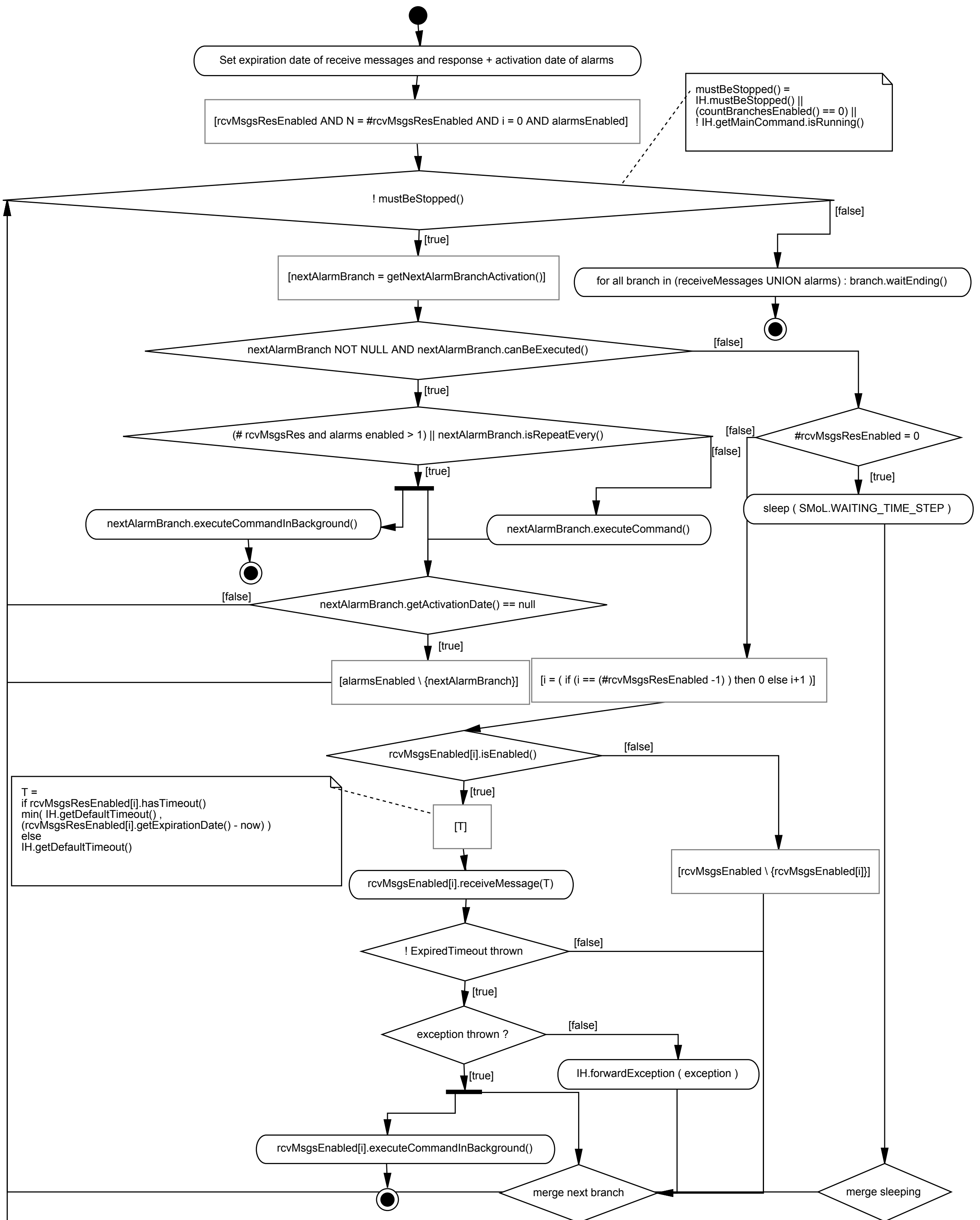

Figure D.23: SMoL Information Handler - ReceiveMessage, ReceiveEvent and alarms round robin

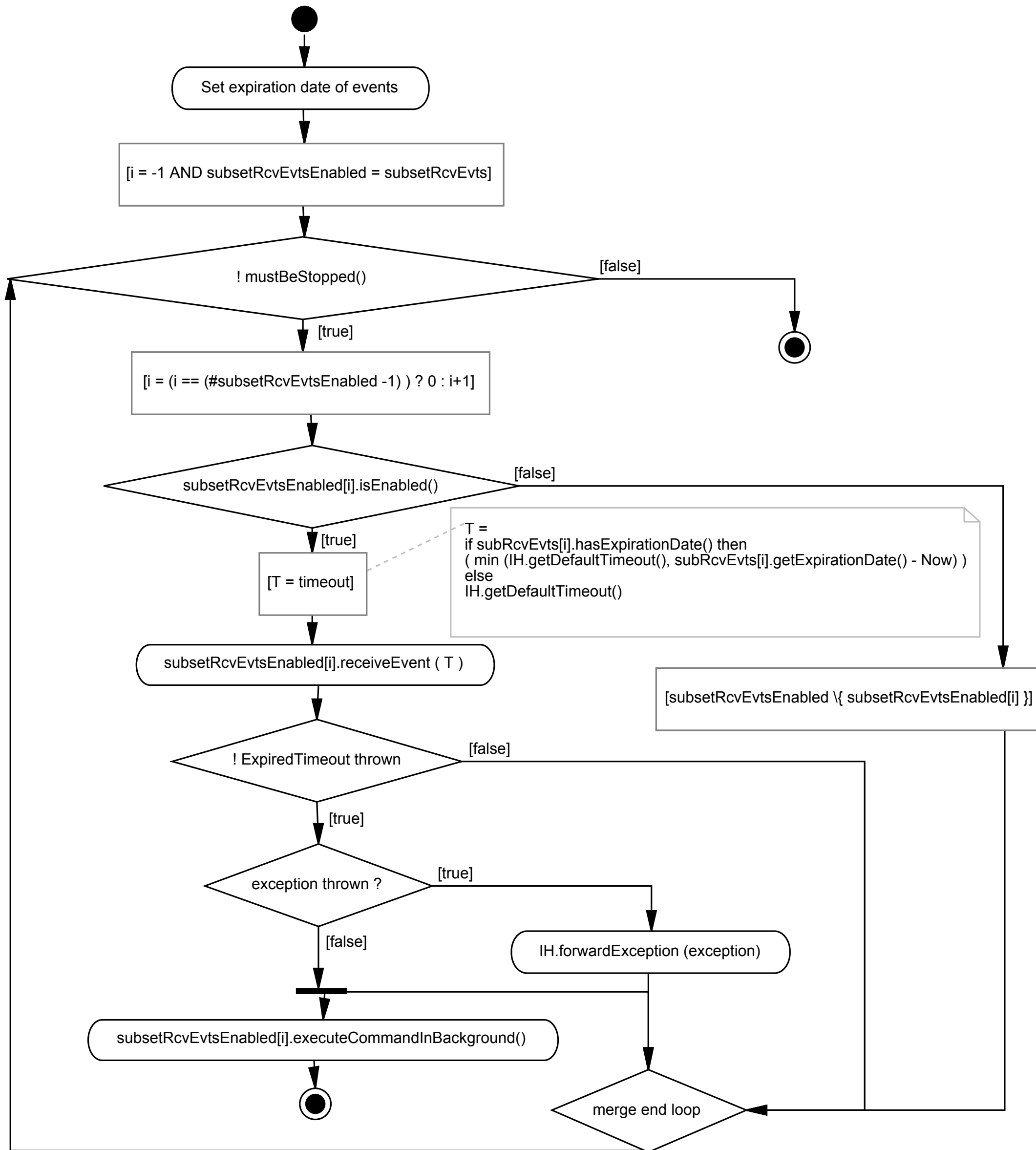

Figure D.24: SMoL Information Handler - ReceiveEvent Round Robin

Appendix E

Translator Architecture

Figure E.1: Common Translator Classes - Part 1

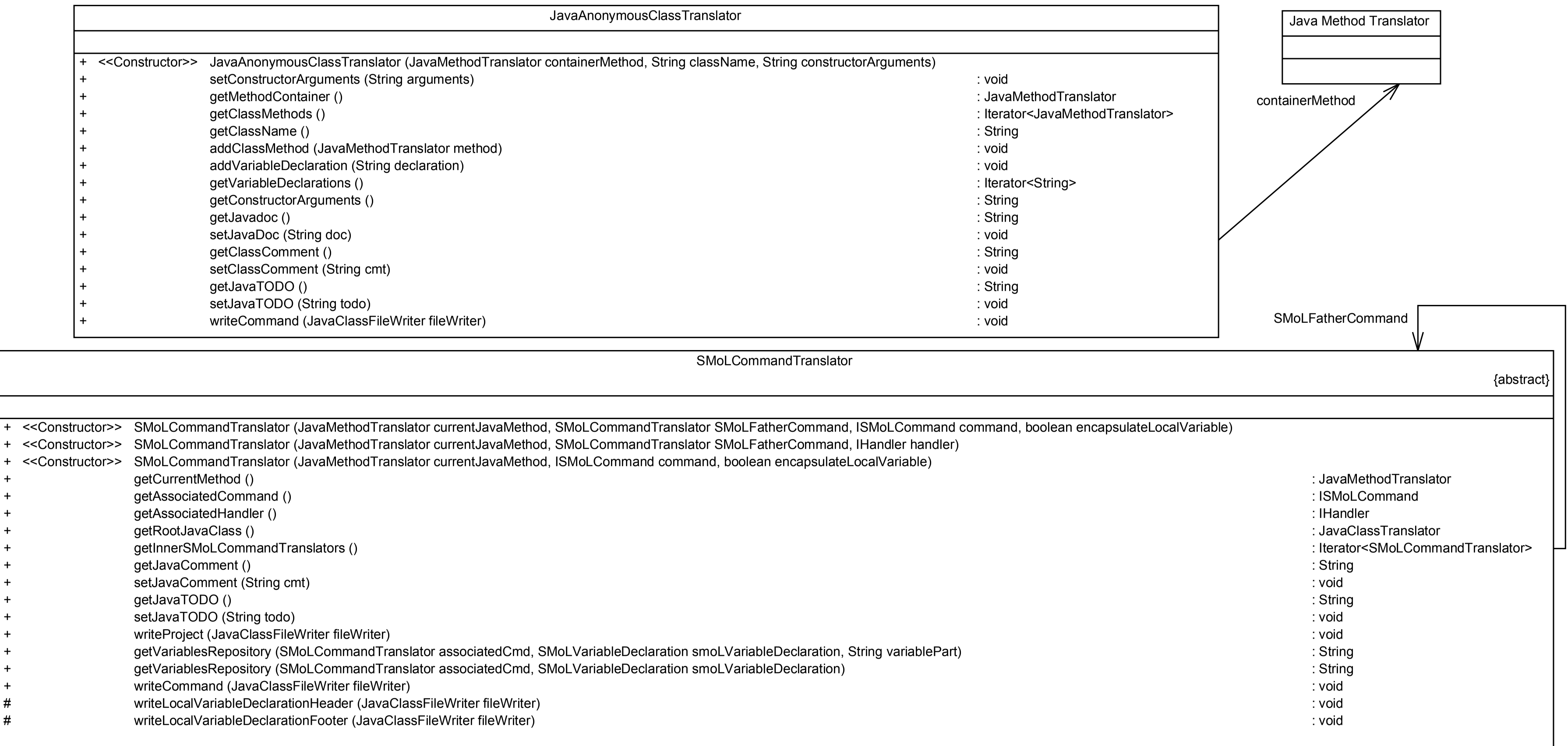

Figure E.2: Common Translator Classes - Part 2

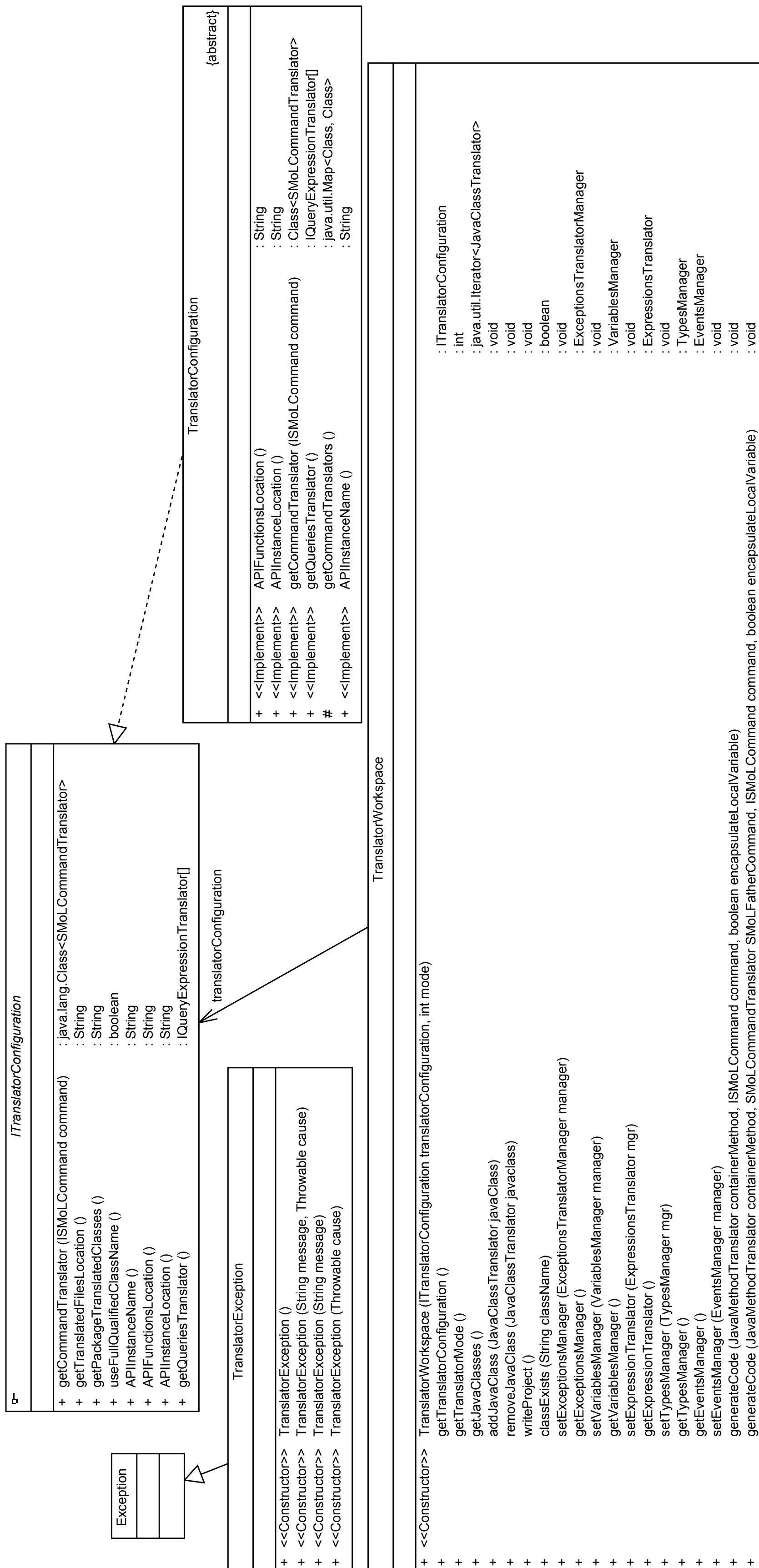

Figure E.3: Common Translator Classes - Part 3

E.1 Peers Translator

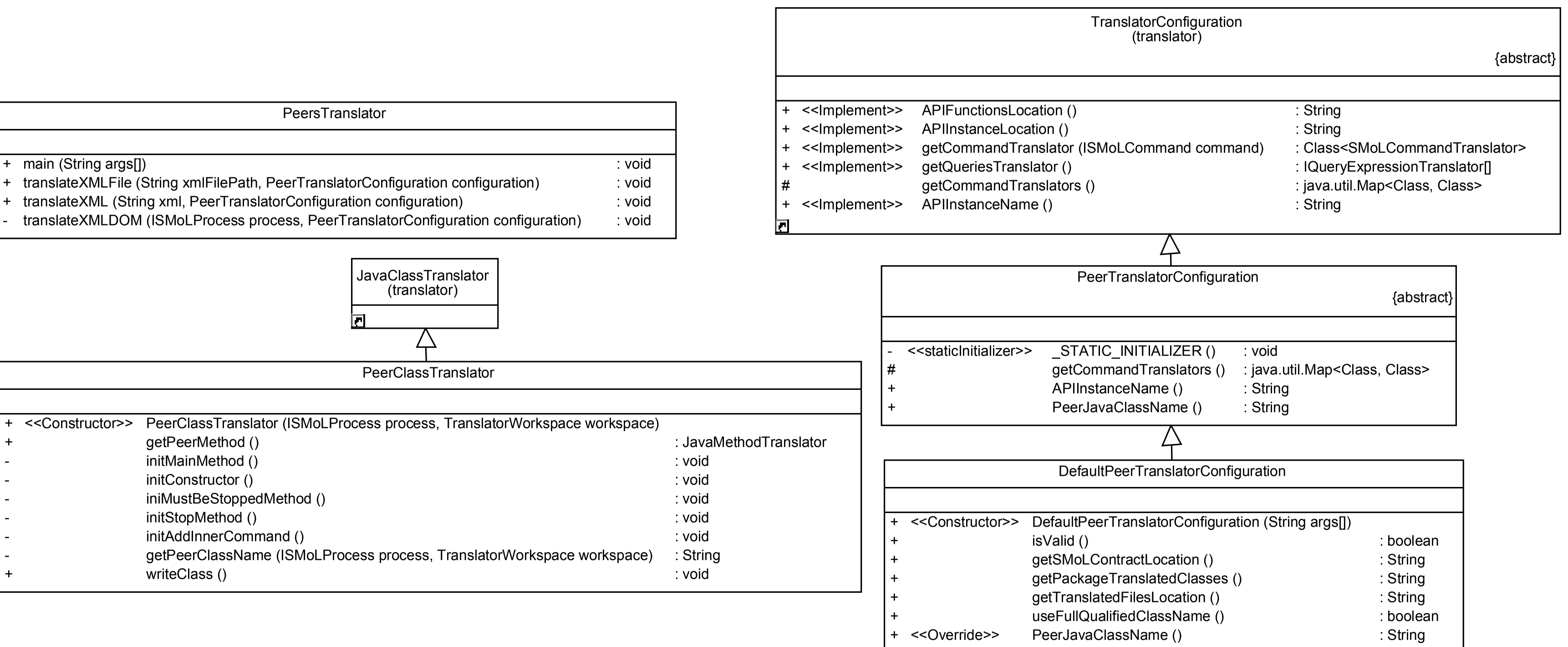

Figure E.4: Peers Translator

E.2 Services Translator

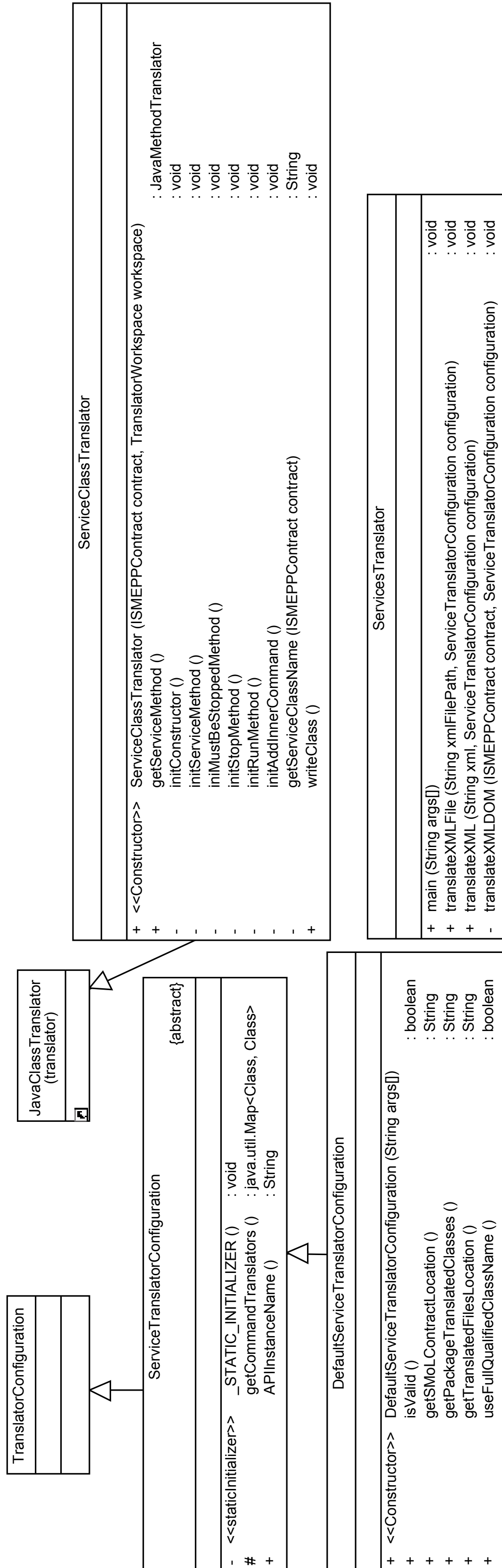

Figure E.5: Services Translator

E.3 Events Manager

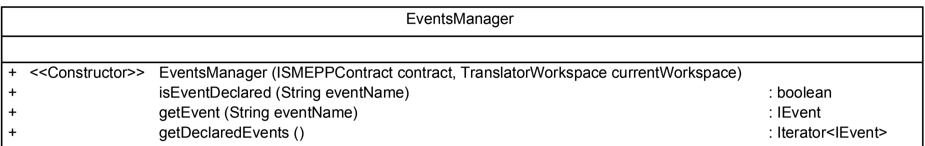

Figure E.6: Events Manager

E.4 Exceptions Manager

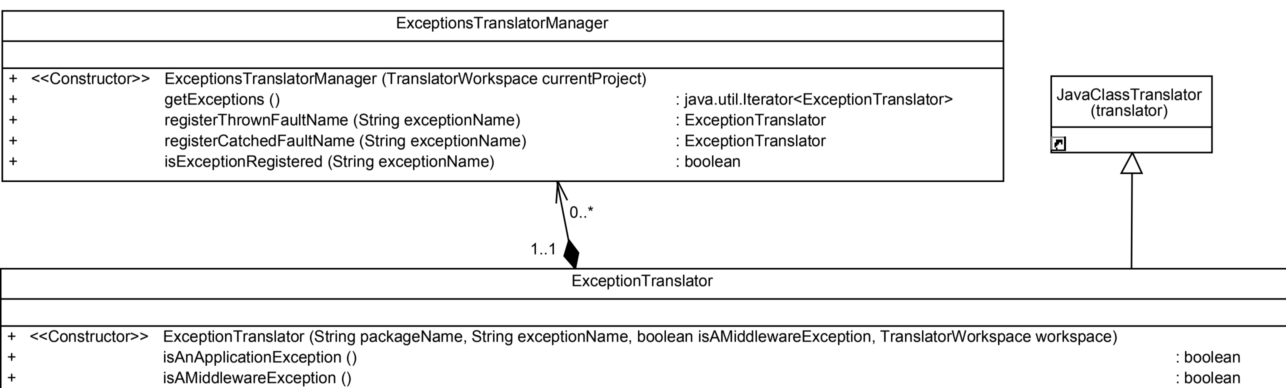

Figure E.7: Exceptions Manager

E.5 Expressions Manager

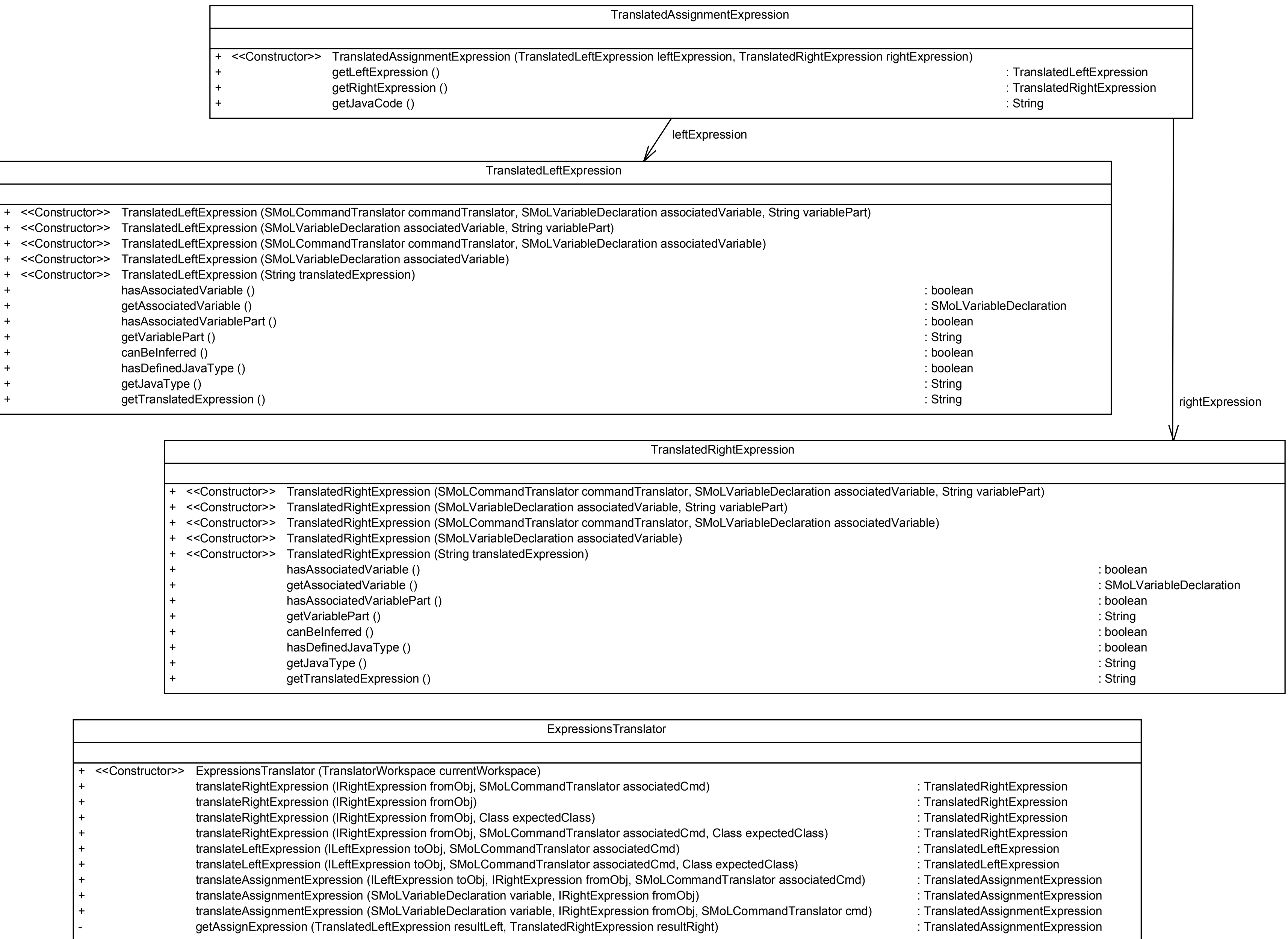

Figure E.8: Expressions Manager - Part 1

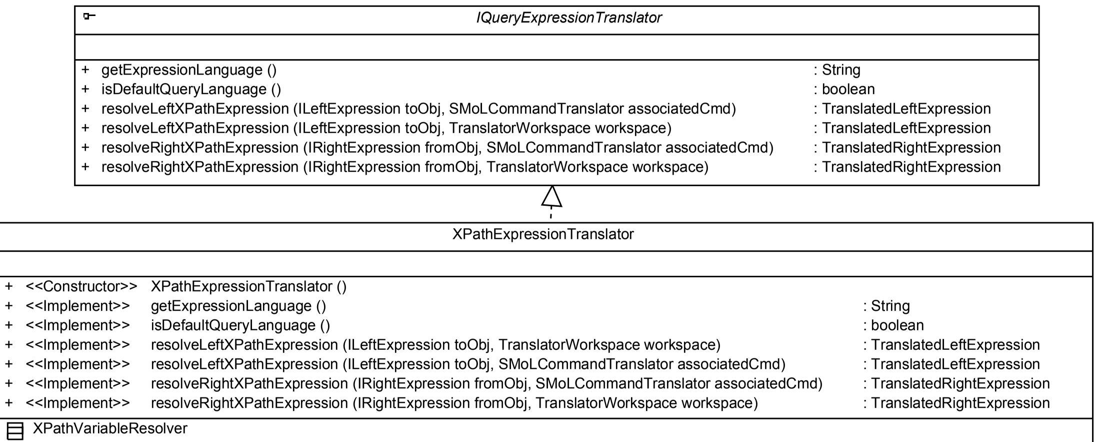

Figure E.9: Expressions Manager - Part 2

E.6 Types Manager

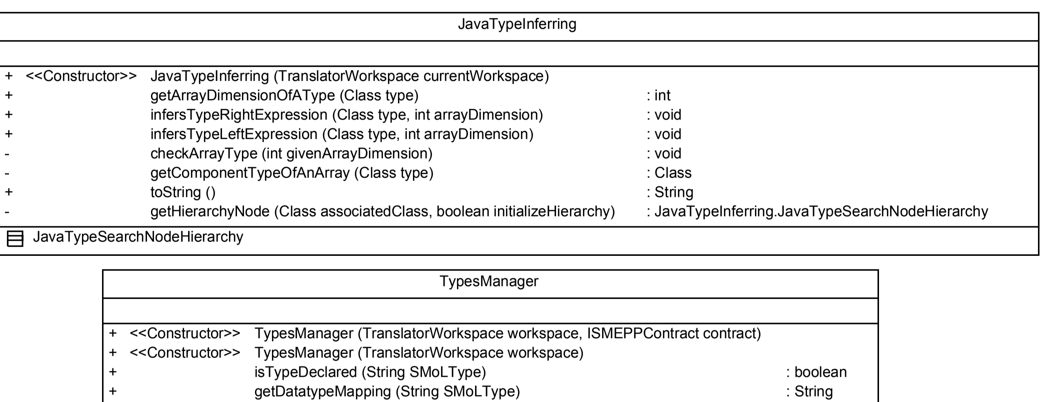

Figure E.10: Types Manager

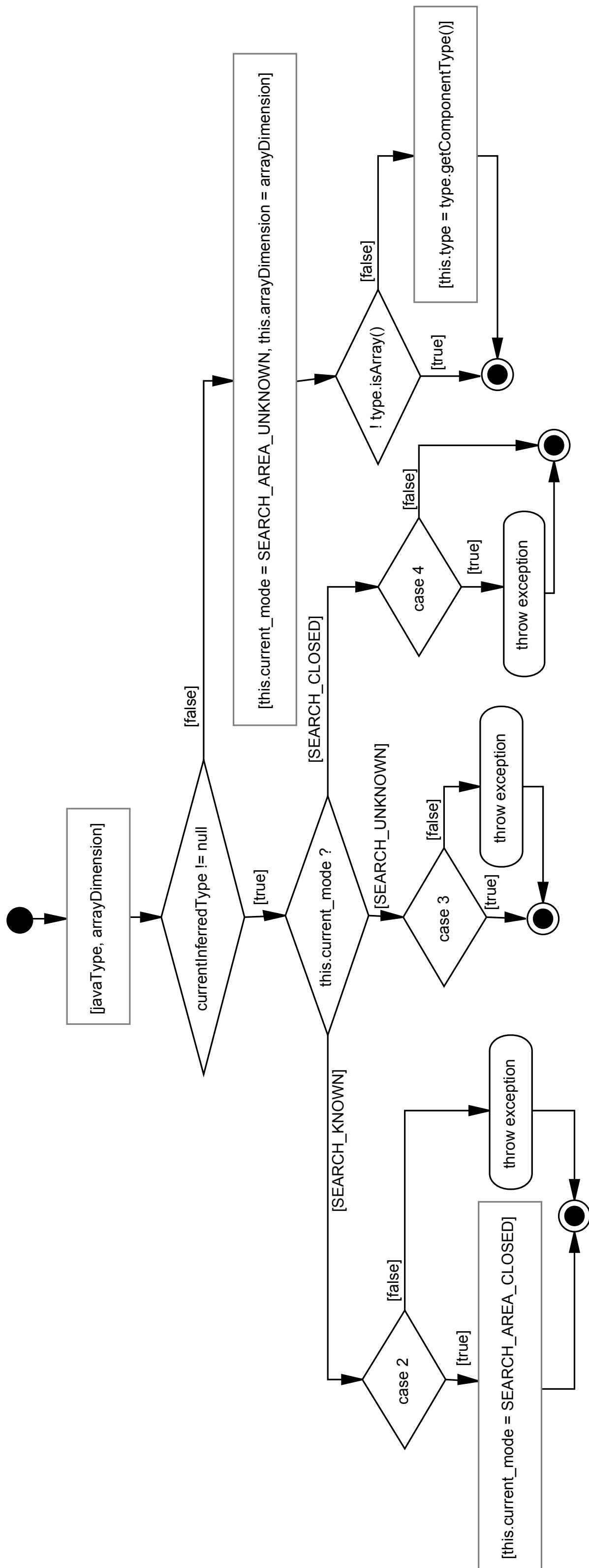

Figure E.11: Type Inferring - Variable used in A Right Expression

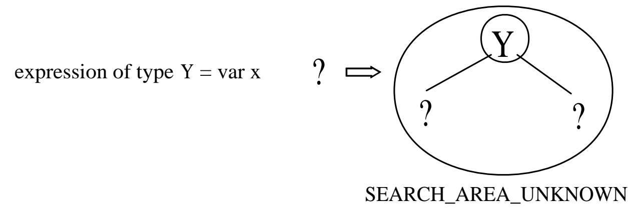

Figure E.12: Type Inferring - Variable used in A Right Expression - Case 1

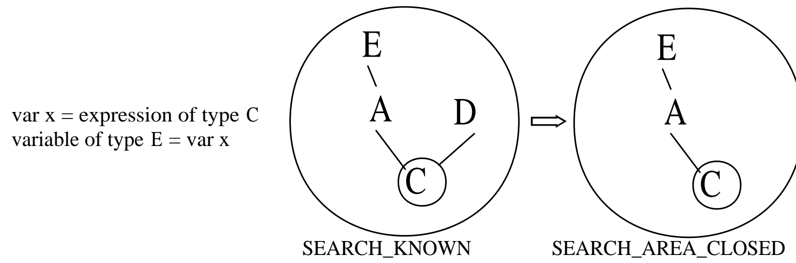

Figure E.13: Type Inferring - Variable used in A Right Expression - Case 2

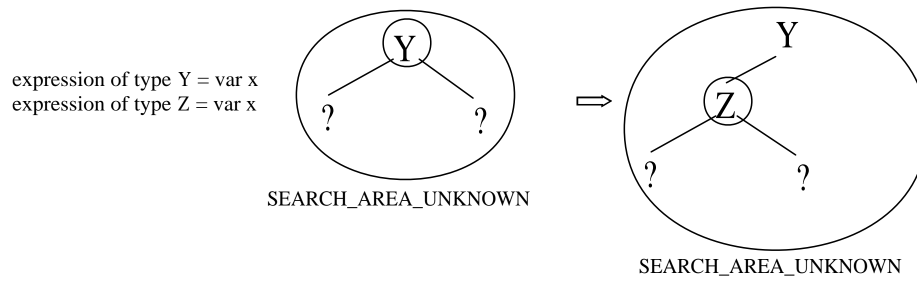

Figure E.14: Type Inferring - Variable used in A Right Expression - Case 3

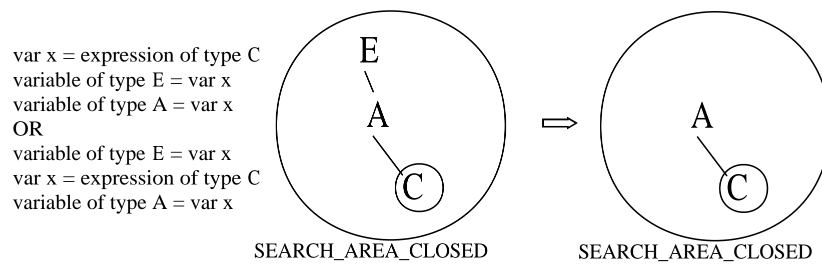

Figure E.15: Type Inferring - Variable used in A Right Expression - Case 4

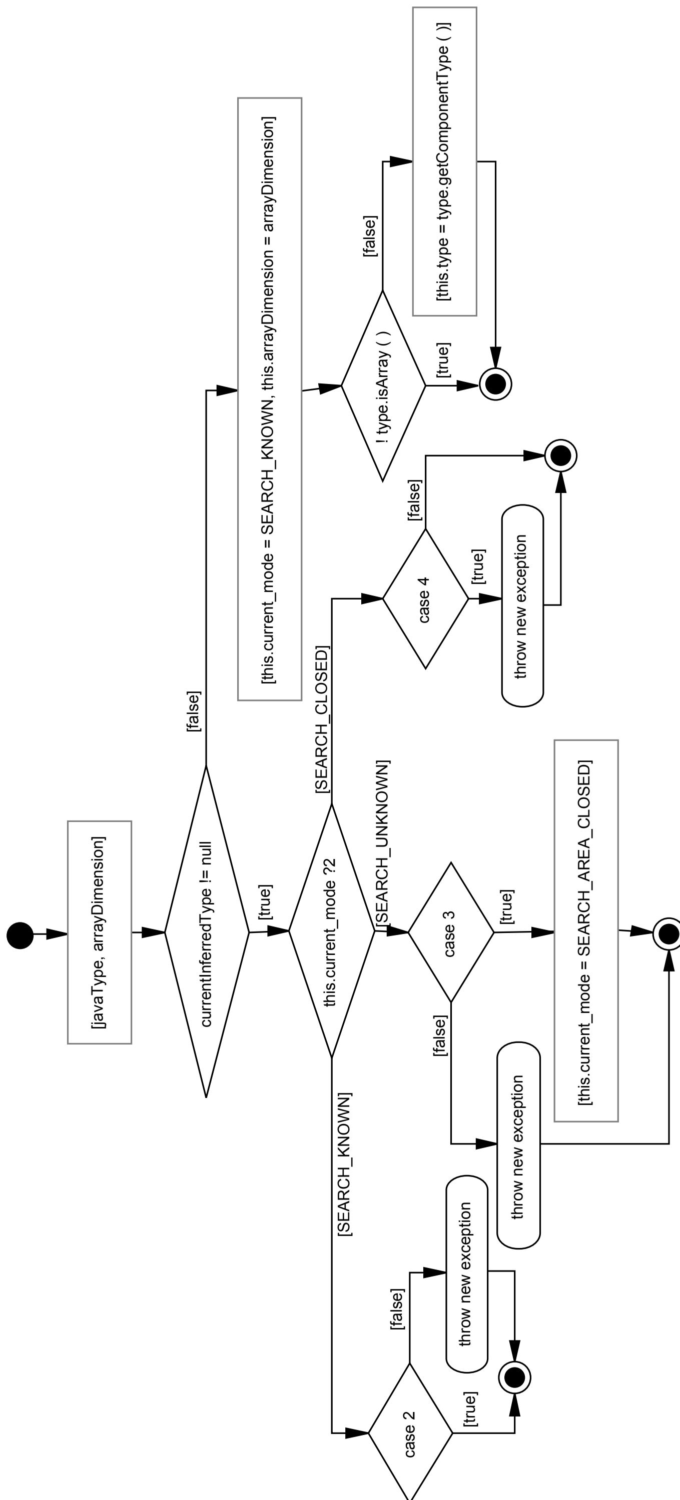

Figure E.16: Type Inferring - Variable Used in A Left Expression

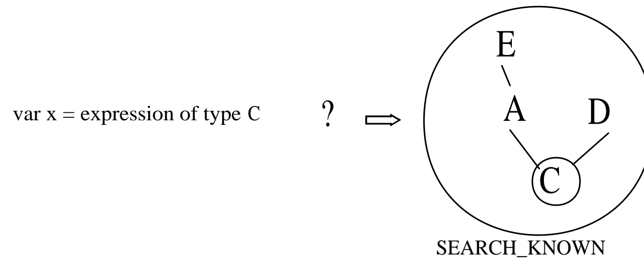

Figure E.17: Type Inferring - Variable Used in A Left Expression - Case 1

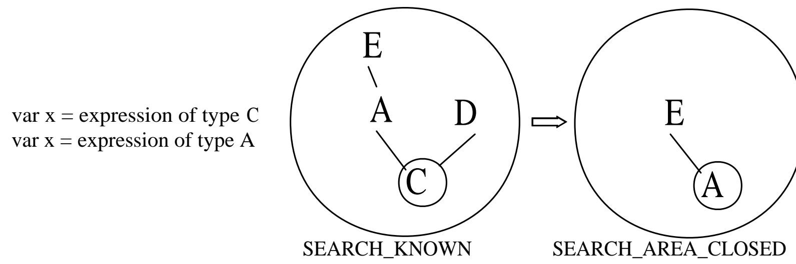

Figure E.18: Type Inferring - Variable Used in A Left Expression - Case 2

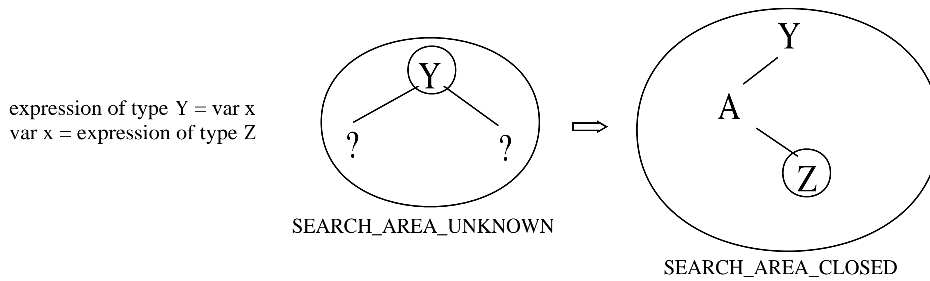

Figure E.19: Type Inferring - Variable Used in A Left Expression - Case 3

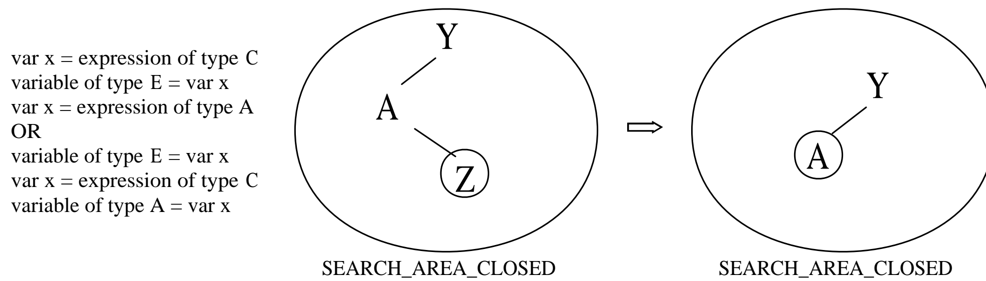

Figure E.20: Type Inferring - Variable Used in A Left Expression - Case 4

E.7 Variables Manager

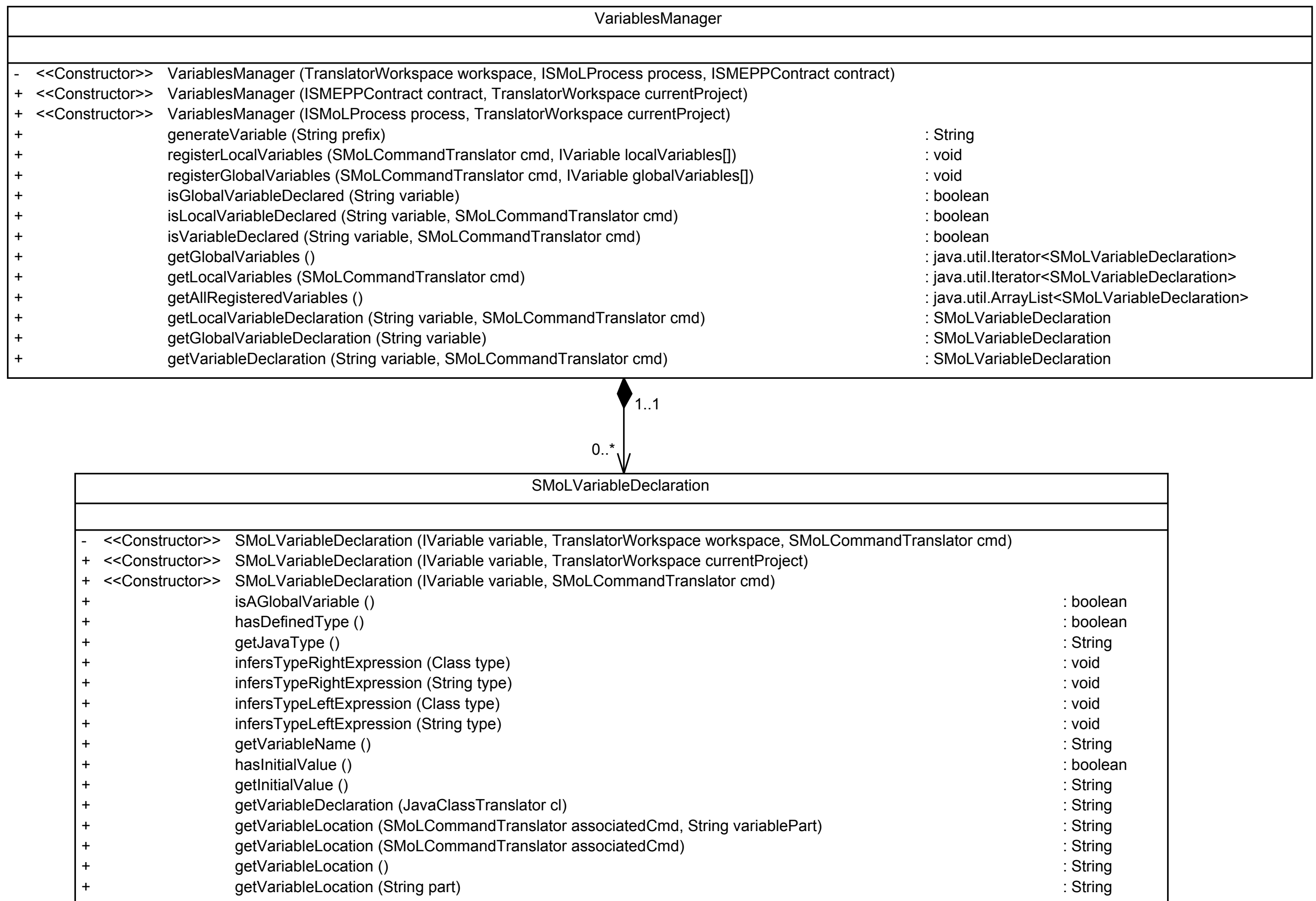

Figure E.21: Variables Manager